Neural Computing and Applications (2023) 35:12145–12157
https://doi.org/10.1007/s00521-023-08346-x

ORIGINAL ARTICLE

Circuit design completion using graph neural networks

Anwar Said[1] · Mudassir Shabbir[1] · Brian Broll[1] · Waseem Abbas[2] · Peter Völgyesi[1] · Xenofon Koutsoukos[1]

Received: 2 September 2022 / Accepted: 26 January 2023 / Published online: 17 February 2023

Abstract

Electronic design automation tools are widely used in circuit design and greatly assist designers in handling the complexities and challenges of circuit design and evaluation. There have been numerous recent developments in using machine learning tools, particularly graph neural networks (GNNs), to address circuit design problems. These techniques take advantage of the natural representation of a circuit as a graph. In this study, we propose using state-of-the-art GNNs to solve a key circuit design issue. Specifically, we are interested in addressing the *circuit completion problem (CCP)*, where the goal is to determine the missing components and their connections in a partially designed or evaluated circuit. We provide a novel two-step solution to this problem: First, we formulate missing component identification as a graph classification task in the graph-based representation of partial circuit, and second, we treat the placement and connectivity of the newly (predicted) component as a link completion problem. We propose a novel graph learning framework called feature-enhanced graph isomorphism network that combines both GNNs and graph descriptors in an end-to-end fashion to extract expressive graph representations. We also present three new circuit datasets to implement and test our solutions. Our extensive experiments demonstrate that the proposed framework is an effective and generalizable solution to the CCP problem.

Keywords Circuit completion · Graph neural networks · Graph classification · Link prediction · Circuit design suggestion

1 Introduction

Many modern Integrated Development Environments (IDE) propose recommendations to help human software developers and designers in the form of auto-completion [1–4]. This automated source code generation not only leads to a better design experience but also improves the productivity and efficiency of the development process. Lately, the machine learning community has developed innovative models and tools, for instance, GPT-3, Co-PILOT, and Code-whisperer, that generate these suggestions with near-perfect accuracy and have been very helpful to software designers [5–7]. However, we observe that similar or equivalent features for design suggestions are largely missing from the modern circuit design domain. While the design and synthesis of the circuit of electrical components are based on well-defined goals and specifications, we observe that privacy concerns may prohibit one from sharing such specification-related information in a meaningful way. Furthermore, it may be computationally impractical for a circuit design software to offer auto-complete suggestions based on a complete specification.

✉ Mudassir Shabbir
mudassir.shabbir@vanderbilt.edu

Anwar Said
anwar.said@vanderbilt.edu

Brian Broll
brian.broll@vanderbilt.edu

Waseem Abbas
waseem.abbas@utdallas.edu

Peter Völgyesi
peter.volgyesi@vanderbilt.edu

Xenofon Koutsoukos
xenofon.koutsoukos@vanderbilt.edu

[1] Department of Computer Science, Vanderbilt University, Nashville 37235, TN, USA

[2] Department of Systems Engineering, University of Texas at Dallas, Texas 75080, TX, USA

Therefore, it is an interesting and challenging problem from an algorithmic perspective to generate relevant suggestions to complete a circuit design based solely on the partial circuit design at hand. Motivated by the successful application of machine learning models for automated source code generation, we propose formulating and studying the partial circuit completion problem using graph machine learning.

A pertinent application of the proposed work is the reverse engineering (RE) of electric circuits, which refers broadly to the retrieval of circuit design, layout, netlist, and/or functionality through testing and inspection [8]. RE is often practiced to detect malicious alterations and identify and replace obsolete hardware, e.g., [9–11]. One of the goals in RE is to use the available circuit information and utilize it to characterize/determine an unknown (or missing) component in the remaining circuit. RE is a cumbersome process due to circuit complexity and limited information, and as such various techniques are employed to execute it, albeit with limitations [8, 12]. Electronic design automation (EDA) facilitates the rapid design and development of electronic circuits while reducing design errors and defects. However, as an unexpected consequence, it also permits the growth of the counterfeit electronic industry and the implantation of hardware Trojans in forged copies [13]. For example, consider the incident entitled *The Big Hack* [14], which occurred in 2018 and involved inserting unauthorized microchips into products made by a company supplying servers to Apple, Amazon, and even the US government [15].

Similarly, the proposed approach in the paper can potentially be applied to diagnose faulty components or defective interconnections between components. It can be augmented with the existing circuit fault diagnosis and testing techniques to identify the fault/defect. Traditionally, a process called physical failure analysis is performed to locate the defect [16, 17]. However, manufacturing intricacies, the nature of defects, and scalability issues pose several challenges [18, 19]. ML-based circuit completion can be a helpful tool for this purpose; for instance, the setup can be utilized to identify a component's defective/missing interconnections with other components in the circuit. The contributions of the current work are outlined below:

- We propose a novel two-step solution to the circuit completion problem: First, we formulate missing component identification as a graph classification task in the graph-based representation of partial circuit, and second, we treat the placement and connectivity of the newly (predicted) component as a link completion problem.
- We introduce a new graph neural network (GNN) architecture called the feature-enhanced graph isomorphism network (FEGIN) that combines the ability of static graph descriptors to capture rich topological information about the graph with the message-passing capabilities of GNNs to learn powerful expressive representations for solving the circuit completion problem (CCP).
- To help with the training and testing of future graph learning approaches for the circuit completion problem, we present three rich datasets of circuit netlists: *LTSpice examples*, *LTSpice demos* and *Kicad Github*.
- We perform an extensive evaluation of fifteen different graph learning approaches to solve the circuit completion problem. The evaluation models include nine GNNs and six graph descriptors that include statistical, spectral, as well as distance-based embeddings.

The rest of the paper is organized as follows: Sect. 2 summarizes the relevant related work. Section 3 formulates the circuit completion problem (CCP) and discusses the main challenges. Section 4 presents the graph machine learning-based solution of the CCP. Section 6 provides the details of our experiments along with the results. Finally, Sect. 7 concludes the paper.

2 Related work

The size of integrated circuits (IC) and printed circuit boards (PCB) has increased exponentially over the past decade, posing a challenge to the circuit design flow's scalability. To enhance the design, geometry, and packaging of complex ICs and PCBs, machine learning for electronic design automation (EDA) has largely been studied. Due to the extremely large space, machine learning for EDA is becoming one of the most popular areas of interest and has shown promising results [20, 21]. These machine learning methods span nearly all of the stages in EDA design, starting from the architecture design to the final testing phase. A number of machine learning methods have been explored in EDA, especially for physical design [22]. These methods include the use of conventional machine learning methods such as K-nearest-neighbors, neural networks, support vector machines, and random forest. Moreover, the last years have also seen a surge in deep learning approaches for EDA such as convolutional neural networks, recurrent neural networks, reinforcement learning and graph neural networks (GNNs), to name a few [23, 24].

Graphs play a crucial role in EDA design since many EDA objects, such as netlists and layouts, are naturally represented by graphs. Many EDA problems such as layer assignment [25], partitioning [26], multiple pattern layout decomposition [27, 28] and testability analysis [29] have been explored with graph processing [30]. These approaches mainly base on the conventional graph-based approaches. However, a number of approaches have been introduced more recently to study GNNs in EDA design [31–33]. Having the graphical representation of an IC or PCB, GNNs allow to learn the entire structure of the given object which open up a number of downstream machine learning tasks [34]. GNN is a powerful paradigm that has shown excellent results in solving many problems, such as protein folding [35], predicting estimated time of arrival [36] and antibiotic discovery [37], to name a few.

GNNs have found extensive applications in EDA design. Starting from the logical synthesis, GNNs have been studied and applied in almost every EDA's stage [23, 38]. The authors in [39] propose a GNN-based architecture to predict technological mapping done by logic synthesis. Similarly, [30] introduce a GNNs-based framework to predict observation point candidate in a netlist that maximizes fault coverage and minimizes observation point number. The authors adapted multi-stage classification with GNNs and fully connected layers and mapped the problem as a node classification task. For the floorplanning, Google researchers recently presented a more comprehensive approach [40]. This approach uses edge-based GNNs under the umbrella of deep reinforcement learning to automatically generate chip floorplans. In the placement stage, where the design gate locations are mapped to the chip layout, the authors in [41] introduced a graph attention framework to predict components' placement nets. This approach represents a netlist as an attributed directed graph and maps the problem as a regression task. It uses fan-in, number of cells, fan-out sizes, and areas as node attributes, while edge features are computed through graph clustering. Similarly, in [42], reinforcement learning is used to find optimal placement parameters. The authors in [43] and GraphSAGE [44] were leveraged to accelerate and optimize the placement. Some other approaches that use GNNs in EDA include [23, 45–49].

3 Preliminaries and problem formulation

A circuit design schematic for an IC or a PCB is represented by a *netlist*. A netlist is usually a text file where each line represents an electronic component and all its connections, along with any auxiliary information. Each component has a unique ID and a type, e.g., capacitor, resistor, inductor, voltage source, etc. The types of circuit components can be equivalently represented by integers from 1 to k. One can define the circuit completion problem as follows:

Problem 3.1 (circuit completion problem) Let v_0, v_1, be the netlists of two circuit schematics where v_0 is an original circuit while v_1 is (a counterfeit or partial copy) of v_0 that is missing a component x and all its connections that are present in v_0. The circuit completion problem is to predict the type of component x and all its missing connections where $x \in \{1, 2, 3, \ldots, k\}$.

We illustrate the component classification problem in Fig. 1. We observe that there are several challenges in the rigorous formulation and solution of circuit completion problem. Firstly, the problem implicitly assumes a well-defined criterion for *validity* of a circuit that is required to check the correctness of a solution to the problem. While there are some trivial checks, there are no well-defined criteria that software or even a human expert can use to validate a circuit netlist. Secondly, even if a circuit is valid, it may not be one that one is interested in. There may exist another component $y \neq x$, and a corresponding circuit netlist v_0' such that v_0' is also a valid completion of v_1 with component y, i.e., the circuit completion problem on v_1 also has a solution y. This points out the fact that the mapping from a partial design to a missing component type is not a function. As long as both v_0, v_0' are two netlists of interesting designs (where 'interesting' may be defined by application), it may be acceptable to predict either of them as a completion of the partial netlist v_1. But it may be that v_0' is a trivial completion that is of no interest to the particular application one is studying. We observe that, in the presence of these inconsistencies, it makes even more sense to consider a data-driven solution instead of a more deterministic algorithm. As a first step, we strip off any auxiliary information in netlists and translate each netlist to an undirected graph $G = (V, E, \phi)$, where set of vertices V represent the components in the netlist, and the connection between components is denoted by an edge in E between the corresponding vertices in the graph. Further, each vertex v has an integer type, say $\phi(v)$, from $\{1, 2, 3, \ldots, k\}$. Due to this natural translation from netlists to graphs, we have the following version of the circuit completion problem:

Problem 3.2 (circuit completion problem on graphs) Let $G = (V, E, \phi)$, be a graph of a netlist with $\phi(v) \in \{1, 2, 3, \ldots, k\}$. Let G' be a graph obtained by removing an arbitrary vertex $u \in V$ from G. Compute the value of $\phi(u)$ from G'.

There is also a secondary problem that arises in the completion of partial circuit designs. Once we know the type of component that is missing from the circuit, we also want to find out where on the circuit schematic this missing component lies. This question about the geometry or the placement of the component can be posed in terms of its

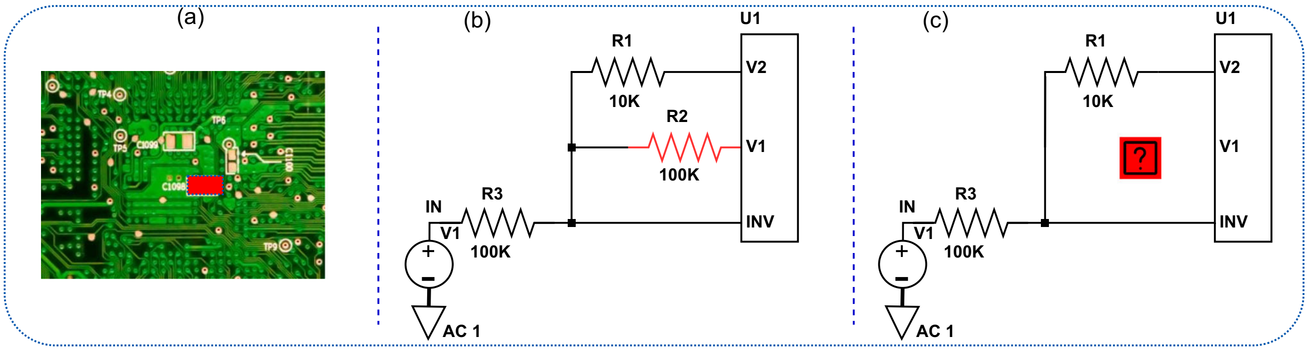

Fig. 1 Illustration of the missing component classification problem. **a** Circuit with a missing component shown as red rectangle, **b** schematic representation of the netlist, and **c** schematic netlist representation with a missing component

connections or links to other existing components. In graph terminology, we have the following problem:

Problem 3.3 (circuit link completion problem on graphs) Let $G = (V, E, \phi)$, be a graph of a netlist with $\phi(v) \in [k]$ where $[k] := \{1, 2, 3, \ldots, k\}$. Let G' be a graph obtained by removing an arbitrary vertex $u \in V$ from G. Given the value of $\phi(u)$ and the G', compute the set of neighbors of u in G.

Here, the set of neighbors of a node u is defined as the following set: $\{v \in V : (u, v) \in E\}$. We propose data-driven solutions to both of these problems. Formally, we are going to learn a map, $\hat{\phi} : G \to \{1, 2, 3, \ldots, k\}$, from a family graphs G to the set of first k integers. This map predicts the type of a missing component based on the topology and types of known components in a graph. A second map, $\hat{\xi} : G, [k] \to 2^V$, takes input a graph G, and a missing component type $\phi \in [k]$ and returns a subset of vertices (power set of vertices V is denoted by 2^V) that are connections of the missing component in graph G. We provide a visual illustration of this problem in Fig. 2.

To build a framework that identifies missing components and their connections in an incomplete circuit, we use two ingredients that are built upon the networked representations of electric circuits. Firstly, to predict the missing component, we leverage the graph neural networks framework that learns graph structure in a graph classification setting. Secondly, we adapted a GNN-based link prediction approach to predict links/edges for the missing component. We provide a visual illustration of the proposed architecture in Fig. 3.

Remark 1 We note, that graph topology alone may not be enough to solve either of the two circuit completion problems, as there may be multiple ways to complete a partial circuit. Therefore, one expects the proposed method to fail at times. However, we show through extensive experiments that on real-world datasets, the results of graph machine learning methods are reliable enough to aid a human expert in the design synthesis and evaluation process.

4 Circuit completion through graph learning

In this section, we outline our proposed data-driven solution to Problems 3.2 and 3.3. *Graph classification* is a well-known problem in graph machine learning, in which a machine learning model is trained to predict a label for an input graph from among two or more classes. Since we need to learn to predict the missing component type for a circuit graph and the number of possible component types

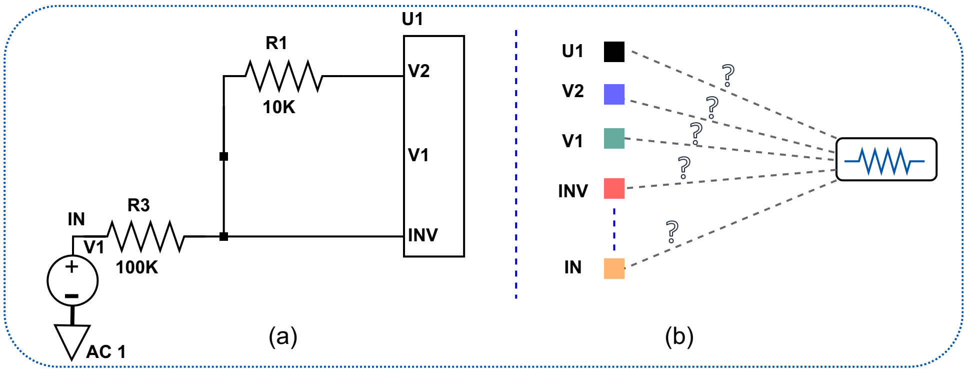

Fig. 2 Illustration of the links insertion problem. **a** Indicates a schematic netlist representation, and **b** shows a new component with possible links to be predicted

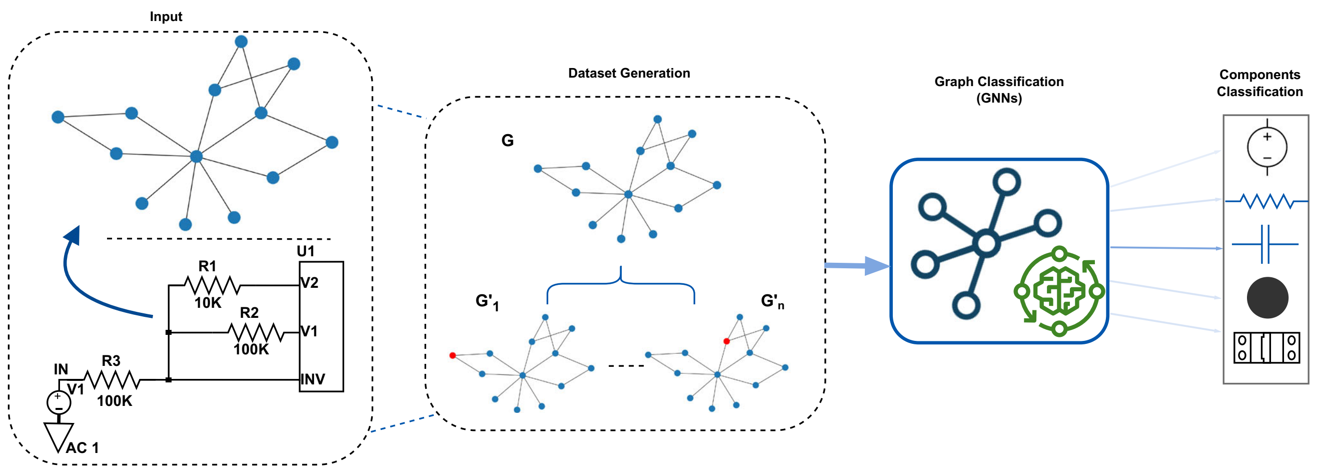

Fig. 3 Architecture of the proposed component classification task

is limited, Problem 3.2 can be naturally cast as a graph classification.

Formally, given a family of n graph, G that represents partial circuits with the missing component type for each $G \in G$ given as a vector $Y \in [k]^n$, we will train a graph classification model to learn the map $\hat{\phi}$ with the objective of minimizing $\sum_{G \in G} 1 - \mathbb{1}_{Y(G)=\hat{\phi}(G)}$, where $\mathbb{1}_{Y(G)=\hat{\phi}(G)}$ is an indicator function to check whether predicted label $\hat{\phi}(G)$ of a graph is equal to the expected component type $Y(G)$. With the standard assumption that the family G is a true representative of all circuit designs, this reduces our problem to choosing a suitable graph machine learning model and optimizing a machine learning model on the given datasets.

Our proposed methodology mainly consists of two ingredients to solve Problems 3.2 and 3.3. In this section, we first present the data collection process and then the details of each solution for both problems.

4.1 Dataset collection and preprocessing

In order to deploy any learning-based techniques, the first steps usually involve the collection of relevant datasets. For the circuit-completion problem, there are no readily available datasets to use. Therefore, we collected three circuit datasets from three different sources to design a solution for this problem. Firstly, we collected example circuits from *LTSpice*[1] and we named this dataset as *LtSpice examples*. Secondly, we collected demo circuits from LTSpice and named it *LTSpice demos*, and third, we scrapped user-created circuits from Github. We name this dataset as *Kicad Github*. For the third dataset, we first discovered KiCad[2] files on Github and then converted them to netlists. Since there is no command line tool available for this task, we implement a custom crawler that collects open source circuits from GitHub. The collected datasets are made publicly available to benefit the research community.[3] The number of valid netlists in train-test sets of *Ltspice examples* is 1505 : 377. Similarly, the train-test ratio of *Ltspice demos* dataset is 235 : 59 and 553 : 139 in *Kicad Github* dataset. We further preprocess these datasets separately for both component classification and link prediction tasks. We present those details in the forthcoming section.

A netlist, v, results in multiple graphs for training or testing purposes of the graph classification because we can remove a node u from G and label the resulting graph as $\phi(u)$. We can have as many instances as there are types of components in a netlist. For our experiments, we only use the five most common component types in a given dataset.

For the link completion task, we opt for an approach that is similar to batching in graph learning domain, where we stack the adjacency matrices in diagonal blocks to represent the entire dataset as a single huge graph. Let $\boldsymbol{A}_i \in [0,1]^{N_i \times N_i}$ be the adjacency matrix of graph G_i with N_i nodes. Then, the resulting graph will have $N := \sum_{1 \le j \le n} N_j$ nodes. The stacked adjacency matrix $\boldsymbol{A}$ with dimensions, $N \times N$, and the corresponding vector with concatenated node features are defined as:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & & \\ & \ddots & \\ & & \boldsymbol{A}_n \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 \\ \vdots \\ \boldsymbol{X}_n \end{bmatrix}$$

Link prediction methods usually require a single giant graph as an input to learn the network structure. Therefore,

[1] https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html.

[2] https://www.kicad.org/.

[3] https://github.com/symbench/spice-datasets.

this approach of stacking all graphs in the training dataset allows us to use off-the-shelf GNN methods for prediction with very little extra computational overhead. The large adjacency matrix can also be stored in a compressed form using sparse data structures.

4.2 Component prediction using graph representations

In this section, we provide the details of our proposed solution to Problem 3.2. To design a data-driven solution, we have access to dataset of graphs generated and labeled with the real-world netlists as outlined in the previous section. Given a family of graphs, $G = \{G_1, G_2, \ldots, G_N\}$ and a corresponding list of class labels $Y = \{y_1, y_2, \ldots, y_N\}, y_i \in [k]$, the problem of learning a function from G to Y is the well-known graph classification problem. Once we have a graph dataset, $G = \{G_1, G_2, \ldots, G_N\}$, Problem 3.2 can be naturally translated into a graph classification problem. Graph classification is usually solved using two types of approaches. Firstly, we have methods that use deep learning (DL) and are known as GNNs. Secondly, there are methods that extract a fixed-dimensional embedding for each graph, which is then used as input to a classifier to learn the required mapping. The latter methods are known as graph descriptors [50, 51]. In GNN methods, vector representations are learned for nodes and/or graphs through a message-passing mechanism that involves aggregating graph neighborhood information at each node to update a node's state iteratively. This process is repeated l times to bring together l-hops neighborhood information to learn a feature vector for a node. Formally, the information at layer l of a node v for such a GNN model can be represented as below:

$$a_v^{(l)} = \text{AGGREGATE}^{(l)}(\{h_u^{(l-1)} : u \in N(v)\}) \tag{1}$$

$$h_v^l = \text{COMBINE}^{(l)}(h_v^{(l-1)}, a_v^{(l)}), \tag{2}$$

where h_0 is initialized with the input node features X. The AGGREGATE and COMBINE functions play a crucial role in any GNN architecture, so they are chosen carefully. A number of aggregation methods have been proposed such as MEAN, MAX, SUM and LSTM for the aggregation. The COMBINE function could be a concatenation of the node features followed by a linear transformation. Finally, to perform the graph classification task, one could aggregate the learned node embeddings to get a feature vector for the entire graph. This process of node feature aggregation is known as the READOUT operation in the graph learning domain. Given the learned embeddings of the final layer $h_v^{(L)}$, we can obtain graph-level embeddings as follows.

$$h_G = \text{READOUT}(\{h_v^{(l)} | v \in G\}) \tag{3}$$

GNNs leverage node features along with the graph structure to learn graph embeddings and enjoy state-of-the-art results in many applications in far-reaching domains like bioinformatics, social networks, and many more.

In this work, we introduce a novel GNN architecture that combines graph descriptors and graph convolutional neural networks to perform component identification tasks. In addition, we implemented the SEAL framework [52] to predict connections between the components. In the following, we present the details of a few GNN baselines, followed by graph descriptors that we use in our experimental analysis, and then present the proposed framework that is built upon these techniques.

4.2.1 Graph neural networks

Graph convolutional network (GCN) GCN is the most effective and a seminal GNN model introduced by [32]. This work introduces convolutions on graph signals through message-passing mechanisms where the information from neighboring nodes is aggregated using a weighted average function. This work allows for a more general and efficient framework to implement convolutions on graph-structured data. It also proposes a renormalization trick with the degree matrix to solve the vanishing or exploding gradient problem. The overall architecture is defined as follows.

$$h^{(l)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}h^{(l-1)}W^{(l-1)}) \tag{4}$$

where $h^{(l)}$ is the latent representation of the $l-th$ layer, $\tilde{D}$ is the degree matrix, $\tilde{A}$ is the adjacency matrix with added self-loops, and $\sigma(.)$ is the activation function. GCN showed exceptional results on node classification and link prediction tasks and attracted a large body of researchers to the graph learning domain.

GraphSAGE [44] This work introduces three aggregation functions: mean aggregator, LSTM and Max pooling for the neighborhood aggregation. A general architecture of this framework is presented in Eq. 1.

Graph attention network (GAT) [53]. Attention mechanism is the salient distinguishing feature of GAT-based GNN model that in the neighborhood aggregation, the method assigns non-equal trainable importance weights. This framework allows for different attention weights for the node neighbors, and thus, each neighbor contributes differently to the aggregation. Neighbors that are more important to a node contribute more to the aggregation compared to the less important nodes. With the sum aggregation operator, GAT's architecture can be defined as follows.

$$h_v^{(l)} = \sigma\left(\sum_{u \in N_v} \alpha_{vu}^{(l-1)} W^{(l-1)} h_u^{(l-1)}\right) \tag{5}$$

where α_{vu} is the attention score for node u to v, and W is the weight matrix. This work also employs a multi-head attention mechanism and found it beneficial in the simulations.

Graph isomorphism network (GIN) [54]. GIN employs a multilayer perceptron in the aggregation along with a scalar that distinguishes the representation of a node's previous layer from the representations of its neighbors. We define GIN's architecture as follows.

$$h_v^l = \mathrm{MLP}^{(l)}\left(\left(1 + \epsilon^{(l)}\right).h_v^{(l-1)} + \sum_{u \in N_v} h_u^{(l-1)}\right) \tag{6}$$

Nested graph neural networks (NGNNs) [55] NGNNs are a recently proposed GNN framework that uses two levels of GNNs: a base GNN and an outer GNN. The core idea of NGNNs is to use rooted subgraphs (subgraphs with local h-hops) for each node and apply a base GNN to each rooted subgraph independently. These local node representations are further aggregated with a subgraph pooling layer to get final representations for the root nodes. Then, an outer GNNs is trained to learn the representations for the entire graph.

4.2.2 Graph descriptors

Graph descriptors are found more effective on graphs with little information at the node level [56]. These methods extract feature representations from graphs in a single shot, and then, off-the-shelf ML models are used for the downstream ML task. We present the schematic diagram of a general graph descriptor framework for the graph classification task in Fig. 4. In this work, we applied six graph descriptors to test their performance on our data. In the forthcoming section, we briefly discuss those methods.

Family of spectral graph distances (FGSD) [57] FGSD is a graph representation method that is based on the spectral pair-wise distances among nodes. It defines a generalized spectral distance function on graphs to compute their corresponding feature representations. The distance function uses the sum of the squared difference of the eigenvectors scaled by a function of the eigenvalue to compute a distance between two nodes. These distances are further stacked in a multiset and computed histogram representation.

Network Laplacian spectral descriptor (NetLSD) [50] NetLSD is a recently proposed simple graph descriptor that is based on the properties of the graph Laplacian spectrum. In particular, it inherits heat and wave kernel properties to extract the corresponding graph representation. The core idea behind NetLSD is to consider the heat diffusion process on the graph. By solving the heat equation, NetLSD computes the trace of the heat kernel matrix at different time scales and considers it the representation of the graph.

Distributed statistical graph descriptor (DGSD) [51] DGSD is a recently proposed statistical graph descriptor that is based on pair-wise node distances. It encodes nodes' proximity information by defining a distance function based on simple graph statistics such as node degree, common neighborhood, and their mutual connectivity.

NetSimile [58] NetSimile is a simple graph representation method that considers four social theories to encode the graph's structure: social capital, balance, structural holes, and social exchange. These theories encapsulate several local and global-level properties of the network, such as node proximity, information flow, transitivity, and reciprocity.

Weisfeiler-Lehman Kernel (WL) [59] WL is a well-known graph kernel that is based on the WL-test of isomorphism on graphs [60]. At each iteration, it augments node labels by the set of neighboring node labels and creates a new compressed label for that node. Finally, it constructs a histogram from the final node labels to get the desired graph representation.

Shortest Path Kernel [61] Shortest path kernel defined over a graph G encapsulates shortest path distances among all pair of nodes. It first constructs a shortest path graph S from G whose vertices equal to those in G and its edges are the subset of G's edges labeled by the number of shortest path distances among the endpoints. Based on the shortest path distances in S, it then defines a graph kernel that extracts the desired representation from the graph.

4.3 Predicting connections using GNNs

In Problem 3.3, we are given a graph G with nodes types, as well as a distinguished node v, and the goal of the problem is to predict the neighborhood of v in G. This problem can be further simplified by considering the question whether v, u should be an edge or not for each $u \in G$. This connection test problem in a circuit, given a component v, can be cast as a link prediction task where the aim is to predict whether two nodes in a graph are likely to have an edge based on the rest of graph connectivity. Link prediction is a well-studied problem with numerous applications, including knowledge graph completion,

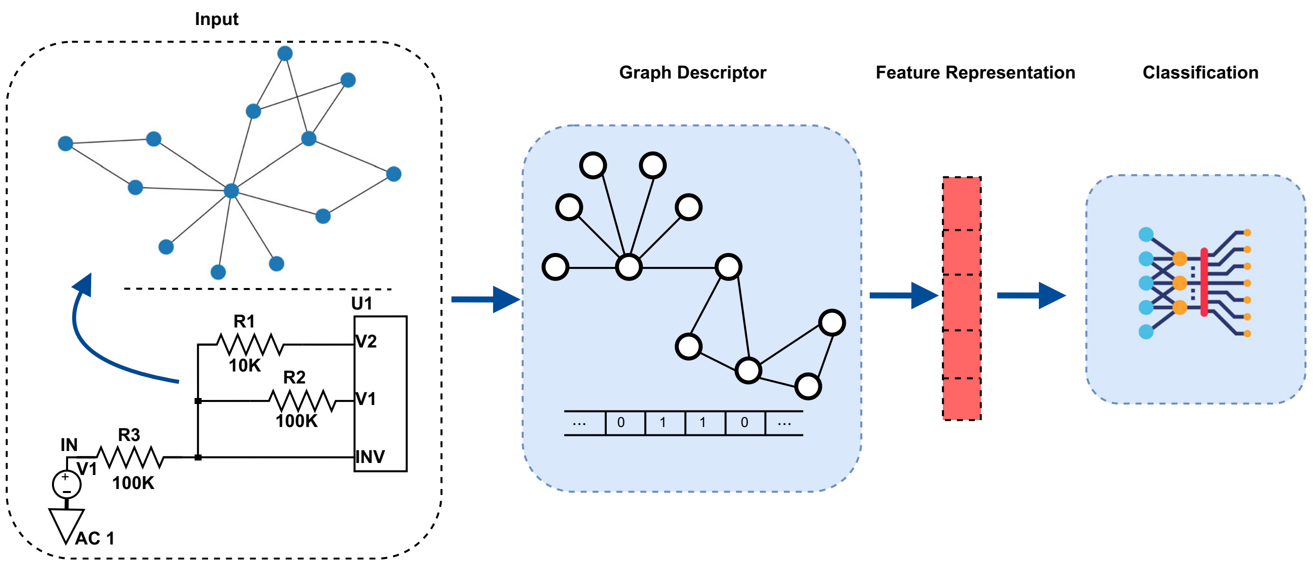

Fig. 4 A schematic visualization of graph descriptor framework

friend recommendation, and gene interactions in biological networks. We define the link insertion problem as a link prediction problem as follows.

One category of solutions to the link prediction problem is known as the "heuristic methods," where the likelihood of links is computed through different node similarity measures such as Jaccard Similarity, Katz Index, Common Neighbors, and PageRank. A second class of link prediction approach is based on machine learning, where the structure of the graph is learned and then the likelihood of links is predicted. The learning-based approaches are more robust as they are computationally efficient and can be applied in an inductive setting. Typically, these methods take a single graph as an input and learn the structure of the network in an end-to-end fashion. And then, the trained model is used for the desired link prediction task.

We propose to select the SEAL (subgraphs, embeddings and attributes for link prediction) framework that is based on the learning approach.

The idea is to enclose subgraphs around the links present in a graph (positive class instances) to extract training data. For a pair (x, y), the *enclosing subgraph* is a subgraph that contains h-hop neighborhood for both x and y nodes. In practice, the idea of enclosing subgraphs works quite well for the link prediction task. Moreover, this work also proposes a node labeling approach denoted as "DRNL" (Double-Radius Node Labeling) to label nodes within the subgraphs. The purpose of using DRNL is to mark nodes' different roles to preserve structural information. We adapt SEAL for the link prediction task on circuit graphs and train it with DRNL one-hot encoding concatenated with the original component node features. We present the schematic diagram for the SEAL framework in Fig. 5.

5 Feature-enhanced graph isomorphism network (FEGIN)

In addition to training and evaluating the graph machine learning models that are discussed in the previous section, we also propose a novel graph neural network-based framework called feature-enhanced graph isomorphism network (FEGIN). In this section, we provide details of this novel model.

As outlined in the previous sections, GNNs encode graph structure using message-passing mechanisms to iteratively update the features vector at each node followed by a weighted custom aggregation function. Various GNN models differ based on the choice of aggregation functions and the implementation mechanism of message passing. For instance, GCN [32] uses mean aggregation function, GAT [53] uses attention mechanism during the message passing, and GIN [54] uses MLP to aggregate node features. In a graph classification setting, this node-level information is then aggregated through a pooling mechanism to construct the desired graph-level feature representations. The main objective of the graph pooling mechanism is to collapse all nodes' features into a fixed dimensional feature vector that is independent of the size of the input graph. We note that in general GNNs tend to perform better on dense graphs as compared to sparse graphs. This might be due to the fact that message-passing mechanism is more effective in dense graphs as relative pairwise distances among nodes are smaller. On the other hand, the fewer edges in a sparse network, such as the one representing circuit designs, may prevent GNNs from exploiting the full potential of the otherwise quite effective message-passing mechanism. Moreover, the graph pooling

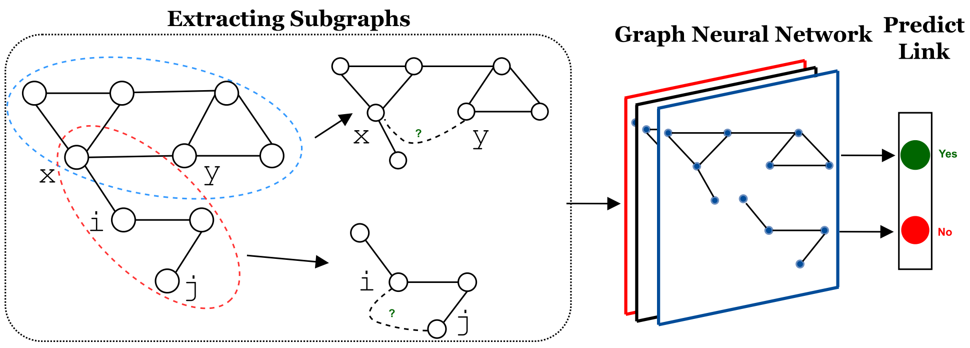

Fig. 5 Link prediction (SEAL) framework. (Left) enclosing subgraphs are extracted for each target link, and (right), a GNN model to learn subgraph representations and perform link prediction

operators that compress the learned node-level features to fixed dimensional latent space may result in further potential loss of information. This significantly affects the performance of the graph classification method [62] on circuit-based graph inputs with mostly constant node degrees.

To address these limitations, we propose a feature-enhanced graph isomorphism network (FEGIN) that combines the latent representation of a graph embedding method with a message-passing mechanism such as GIN to accommodate for any loss of information in sparse networks. In addition, we use global sort pooling [63] followed by a multi-layer perceptron (MLP) to further stabilize the pooling mechanism. As illustrated in Fig. 6, the proposed framework consists of three main components: a) *a message-passing network* such as GIN, (b) *a fixed-dimensional graph embedding* such as NetLSD descriptor, and (c) *sort pooling layer* with MLP. Sects. 4.2.1 and 4.2.2, respectively, provide details on graph isomorphism network (GIN) and network Laplacian spectral descriptor (NetLSD) that we use for message-passing mechanism and graph embedding of FEGIN, respectively. Given a graph as an input, the NetLSD embedding is computed as follows:

$$H_t = e^{-Lt} = e^{\Phi(-\Lambda t)\Phi^\top} = \Phi e^{-\Lambda t}\Phi^\top. \tag{7}$$

where L is the graph Laplacian matrix, t is the time, Φ is the matrix of mutually orthogonal eigenvectors, and Λ is a diagonal matrix of the eigenvalues of L. Here H_t defines the heat kernel over the graph at time t where ij^{th} entry indicates the amount of heat transferred from node i to node j. NetLSD embedding is then defined over H_t at different scales as: $h'(G) = \{tr(H_t)\}_{t>0}$. Similarly, we use GIN, equation 6 to extract node embeddings h_v. Once we have the embeddings matrix $H^{n\times d}$ using GIN, we apply sort pooling operator [63] which provides us graph-level embedding $h(G)$ for the given graph G. The key idea of sort pooling is to choose the top k rows of H after row-wise sorting based on the last feature channel in decreasing order and flatten them into a single feature vector. Here, k is a hyper-parameter that may need to tuned for the particular application at hand and available computational resources but customarily, it is chosen in such a way that it covers 60% of the nodes of the entire training dataset. This allows for a consistent and refined WL ordering to be imposed on the graph vertices, which makes it possible to apply traditional learning models to the sorted representations. Finally, we concatenate $h(G)$ and $h'(G)$ and apply a two-layered MLP to get the final embedding of the graph. We note that we compute $h'(G)$ only once for the entire

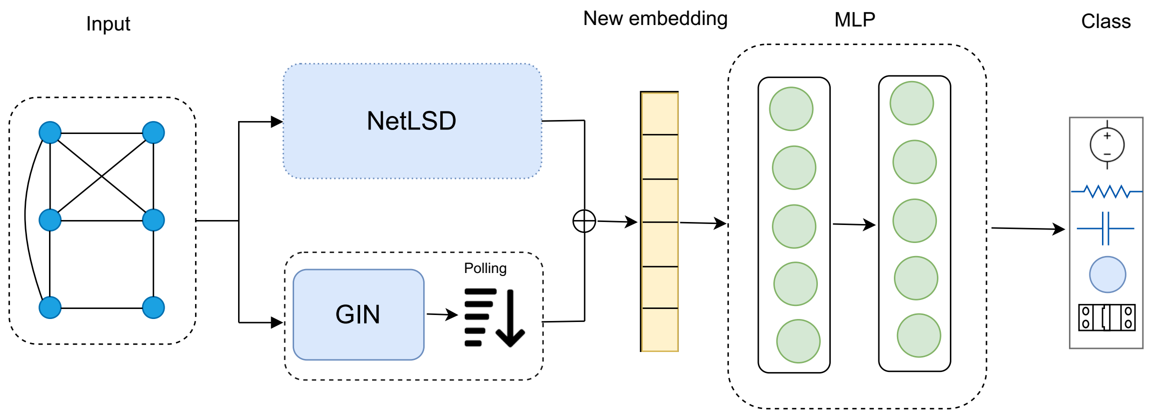

Fig. 6 Illustration of the proposed FEGIN framework

dataset and then use them as per need in EFGIN's architecture.

6 Experimental results

We perform extensive experiments to evaluate the performance of the proposed framework on three netlist datasets. The results show that the proposed framework is an effective and well-suited framework for the detection of counterfeit designs. It achieves satisfactory results on both the component classification and link prediction tasks. For evaluation, we use F1 for graph classification and AUC for link prediction. We repeat all the experiments 10 times and report the average evaluation metrics along with the standard deviations. We made the source code[4] and datasets[5] publicly available to foster reproducibility of the results. In the forthcoming sections, we present the details of dataset preprocessing, models, and experimental results.

6.1 Component classification task

Datasets We use three electric circuit datasets: *Ltspice examples*, *Ltspice demos* and *Kicad Github* in our experimentation. The details of which are presented in Sect. 4.1. We initially split each netlist dataset with 80:20 train-test ratio, transform each of the netlists to graphs, and then generate the desired train and test sets separately for the classification. Because of the extreme class imbalance, we remove classes having less than 300 instances for the *Ltspice examples* dataset, 100 instances for the *Ltspice demos* and 50 instances for the *Kicad Github* dataset. We chose these different thresholds due to the difference in the number of instances for each class in each dataset. We present the component details of these datasets in Table 1. Each dataset has 5 classes with at least 50 training samples per class. We further remove graphs with sizes <5 and greater than 25 from each dataset for the link prediction task (Table 4).

Models We evaluate the proposed framework against GNNs and graph descriptors on component classification task. GNN baselines include graph convolutional networks (GCN) [32], GraphSAGE [44], graph isomorphism network (GIN) [54], graph attention network (GAT) and four versions of these methods with nested graph neural networks (NGNNs) [55]. We also tested graph descriptors including NetLSD [50], DGSD [51], FGSD [57], NetSimile [58] and two kernels WL [59], and shortest path [61] to further showcase the performance without using node attributes. We describe these models in detail in Sect. 4.2.

Experimental Setup We run all the experiments on an 11th Gen Intel Corei9 machine with 64 GB of RAM and a GeForce GTX 1660 Ti GPU. For the GNNs experiments, we use the source code made publicly available by [55] which contains the implementations of all the baselines as well. $80:20$ train-test split was used to split the initial circuit data, and then, the transformation of both train and test sets was separately performed for constructing the classification datasets. We use the same GNNs architectures with 4 layers, 128 batch size, learning rate of 0.001 and train with 100 number of epochs. Hidden dimensions were set to 32 for each layer. For NGNNs, we set the subgraph height (h) to 2 and use $h+1$ message passing layers throughout the experiments. Finally, each model was tested 10 times, and the average weighted-F1 scores on test sets along with the standard deviations are reported. For the graph descriptors, we use a random forest classifier with 500 number of estimators for each descriptor.

Results We report the classification results in Table 2 for GNNs methods and in Table 3 for the graph descriptors. We observe from the results that FEGIN shows excellent results on all three datasets and outperforms all other models. The second-best results on these datasets were achieved by the GIN model. From Table 3, we observed that, compared to GNNs, the performance of the graph descriptors is quite low. Among the models in graph descriptors, NetSimile-based embeddings achieve the best results. Based on the results presented, we conclude that using a graph classification framework with graph neural networks, specifically the proposed feature-enhanced graph isomorphism network (FEGIN) shows a promising approach as a solution to the circuit completion problem.

6.2 Link prediction task

In the previous section, we presented experimental results of a graph classification-based solution to predicting a missing component problem. The second part of the problem is to find the optimal placement or connections from the predicted component to all other components in a given circuit. This is formulated as a link prediction task. In the following, we present the details of the experimental setup and the obtained results.

Models We consider a recently proposed linked prediction model titled SEAL (subgraphs, embeddings and attributes for link prediction) [52]. SEAL is a well-known link prediction model that shows excellent results on many benchmark datasets and has been widely used for link prediction tasks. We present the details of the SEAL model in Sect. 6.2.

[4] https://github.com/Anwar-Said/Circuit-Completion-Using-GNNs.

[5] https://github.com/symbench/spice-datasets.

Table 1 Datasets' class-wise stats for component classification task

	Ltspice examples		Ltspice demos		Kicad Github	
Component class	Train	Test	Train	Test	Train	Test
Voltage_source	1238	303	142	38	N/A	N/A
Sub_element	1224	300	142	38	N/A	N/A
Junction-node	1243	305	142	38	254	72
Resistor	1107	272	142	38	135	38
Behavioral cap	835	189	124	37	114	32
UnifDistRCLine	N/A	N/A	N/A	N/A	142	43
JunFETrans	N/A	N/A	N/A	N/A	86	25

Table 2 Test performance of eight GNNs against the proposed method on three circuit datasets. mean±std of 10 runs of each model are shown

Dataset	GCN	GIN	GAT	GraphSAGE	NGCN	NGIN	NGAT	NGSAGE	FEGIN
Ltspice examples	79.5 ± 0.01	92.7 ± 0.005	78.1 ± 0.04	78.6 ± 0.03	81.9 ± 1.04	88.9 ± 0.01	82.7 ± 0.03	82.2 ± 0.03	*93.1* ± 0.010
Ltspice demos	61.8 ± 0.01	87.9 ± 0.02	58.5 ± 0.03	59.0 ± 0.04	52.6 ± 0.08	82.4 ± 0.03	53.8 ± 0.09	53.7 ± 0.06	*90.0* ± 0.016
Kicad Github	51.7 ± 0.03	66.1 ± 0.02	56.7 ± 0.04	46.6 ± 0.02	58.1 ± 0.01	62.5 ± 0.02	56.8 ± 0.03	57.0 ± 0.02	*67.4* ± 0.01

Italic indicates the best results for each dataset

Table 3 Classification results comparison of graph descriptors and kernel methods (F1)

Dataset	DGSD	NetLSD	WL	FGSD	NetSimile	Shortest path
Ltspice examples	77.63	77.22	77.21	48.88	*85.11*	77.21
Ltspice demos	57.60	52.07	48.27	33.03	*70.86*	48.26
Kicad Github	47.89	47.77	49.35	51.52	*59.27*	49.35

Table 4 Dataset stats for link prediction task

Dataset	Train	Test	# of classes	# of nodes	# of edges
Ltspice examples	901	217	5	13.52	17.71
Ltspice demos	82	25	5	16.47	22.71
Kicad Github	112	29	5	11.69	11.43

Experimental setup We adapted publicly available Pytorch Geometric's implementation for the SEAL model. Similar to the component classification split, we chose the same 80:20 split and construct one training graph from the whole training set. We set batch size of 1, learning rate $1e^{(-4)}$, number of hops to 2, and train the model with 50 number of epochs. Throughout the experiments, we use one-hot encoding of the node labeling generated through Double Radius Node Labeling (DRNL) and concatenate them with one-hot encoding of the components' type. Apart from these parameters, we adapted the similar architecture and experimental setup presented in [52]. In the testing phase, we remove one vertex from each test graph and predict its connections. We repeat this process 10 times and report the average AUC.

Results We report the link prediction results in Table 5. We note that the implementation based on SEAL model obtained results up to 75% on Ltspice examples and Kicad Github datasets. The results on Ltspice LTSpice demo dataset are lower, which may be due to the limited number of training instances in this dataset. Although link prediction on such a sparse dataset is a challenging task, SEAL obtained encouraging results indicating that the proposed framework is effective in learning the netlist structures and can be used for the placement of new components.

7 Conclusion

In this work, we introduced a graph learning framework for the circuit completion problem by mapping the problem of missing components as a graph classification task and presenting a novel approach to find the solution. Further,

Table 5 Link prediction results with SEAL framework (AUC)

Dataset	SEAL
Ltspice examples	74.74
Ltspice demos	61.39
Kicad Github	75.09

we formulated a link prediction task to find the links of the missing component and proposed a link prediction solution. Finally, we conducted extensive experiments to evaluate the performance of the proposed framework. The results demonstrate that our solution is robust and offers a suitable approach to solving the inherently complex and challenging circuit completion problem.

Acknowledgements This work is supported in part by DARPA through contract number FA8750-20-C-0537. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor.

Data availability statement The datasets generated during and/or analyzed during the current study are available in the spice-datasets repository, https://github.com/symbench/spice-datasets.

Code availability The source code and preprocessed data for the experiments are available in the Circuit-Completion-Using-GNNs repository, https://github.com/Anwar-Said/Circuit-Completion-Using-GNNs

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Cai F, de Rijke M (2016) A survey of query auto completion in information retrieval. Found Trends Inf Retr 10:273–363
2. Foster SP, Griswold WG, Lerner S (2012) Witchdoctor: Ide support for real-time auto-completion of refactorings. In: 2012 34th international conference on software engineering (ICSE), pp 222–232
3. Moran K, Vásquez ML, Bernal-Cárdenas C, Poshyvanyk D (2015) Auto-completing bug reports for android applications. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering
4. Rahman MM, Yeasmin S, Roy CK (2014) Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In: 2014 software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering (CSMR-WCRE), pp 194–203
5. Sobania D, Briesch M, Rothlauf F (2022) Choose your programming copilot: a comparison of the program synthesis performance of Github copilot and genetic programming. In: Proceedings of the genetic and evolutionary computation conference
6. Floridi L, Chiriatti M (2020) Gpt-3: Its nature, scope, limits, and consequences. Mind Mach 30(4):681–694
7. Amazon: Introducing Amazon CodeWhisperer, the ML-powered coding companion (2022) https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/. Accessed: 2022-08-02
8. Botero UJ, Wilson R, Lu H, Rahman MT, Mallaiyan MA, Ganji F, Asadizanjani N, Tehranipoor MM, Woodard D, Forte D (2021) Hardware trust and assurance through reverse engineering: a survey and outlook from image analysis and machine learning perspectives. ArXiv arXiv:2002.04210
9. Bao C, Forte D, Srivastava A (2016) On reverse engineering-based hardware trojan detection. IEEE Trans Comput Aided Des Integr Circuits Syst 35:49–57
10. Botero UJ, Tehranipoor MM, Forte D (2019) Upgrade/downgrade: efficient and secure legacy electronic system replacement. IEEE Des Test 36:14–22
11. Grand JA (2014) Printed circuit board deconstruction techniques. In: WOOT
12. Elnaggar R, Chakrabarty K (2018) Machine learning for hardware security: opportunities and risks. J Electron Test 34:183–201
13. Tehranipoor MM, Guin U, Forte D (2015) Counterfeit integrated circuits. Springer, Cham, pp 15–36
14. Robertson J, Riley M (2018) The big hack: Amazon, apple, supermicro, and the Chinese Government. Bloomberg Businessweek
15. Botero UJ, Wilson R, Lu H, Rahman MT, Mallaiyan MA, Ganji F, Asadizanjani N, Tehranipoor MM, Woodard DL, Forte D (2020) Hardware trust and assurance through reverse engineering. Association for Computing Machinery, New York
16. Zhao L, Goh S, Chan Y, Yeoh B, Hu H, Thor M, Tan A, Lam J (2018) Prediction of electrical and physical failure analysis success using artificial neural networks. In: 2018 IEEE international symposium on the physical and failure analysis of integrated circuits (IPFA), pp. 1–5. IEEE
17. Ye F, Zhang Z, Chakrabarty K, Gu X (2013) Board-level functional fault diagnosis using artificial neural networks, support-vector machines, and weighted-majority voting. IEEE Trans Comput Aided Des Integr Circuits Syst 32(5):723–736
18. Pradhan M, Bhattacharya BB (2021) A survey of digital circuit testing in the light of machine learning. Wiley Interdiscip Rev Data Mining Knowl Discov 11(1):1360
19. Ivanova M, Petkov N (2021) Machine learning for in-circuit testing of printed circuit board assembly. In: 2021 4th artificial intelligence and cloud computing conference, pp 221–228
20. Huang Z, Wang Q, Chen Y, Jiang X (2020) A survey on machine learning against hardware Trojan attacks: recent advances and challenges. IEEE Access 8:10796–10826
21. Acampora G, Schiattarella R (2021) Deep neural networks for quantum circuit mapping. Neural Comput Appl 33(20):13723–13743
22. Kahng AB (2018) New directions for learning-based ic design tools and methodologies. In: 2018 23rd Asia and South pacific design automation conference (ASP-DAC), pp 405–410. IEEE
23. Lopera DS, Servadei L, Kiprit GN, Hazra S, Wille R, Ecker W (2021) A survey of graph neural networks for electronic design automation. In: 2021 ACM/IEEE 3rd workshop on machine learning for CAD (MLCAD), pp. 1–6. IEEE
24. Said A, Hassan S-U, Abbas W, Shabbir M (2021) Netki: a kirchhoff index based statistical graph embedding in nearly linear time. Neurocomputing 433:108–118
25. Lee T-H, Wang T-C (2008) Congestion-constrained layer assignment for via minimization in global routing. IEEE Trans Comput Aided Des Integr Circuits Syst 27(9):1643–1656
26. Selvakkumaran N, Karypis G (2006) Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. IEEE Trans Comput Aided Des Integr Circuits Syst 25(3):504–517
27. Yu B, Yuan K, Ding D, Pan DZ (2015) Layout decomposition for triple patterning lithography. IEEE Trans Comput Aided Des Integr Circuits Syst 34(3):433–446

28. Moreno-García CF, Elyan E, Jayne C (2019) New trends on digitisation of complex engineering drawings. Neural Comput Appl 31(6):1695–1712
29. Cheng K-T, Lin C-J (1995) Timing-driven test point insertion for full-scan and partial-scan bist. In: Proceedings of 1995 IEEE international test conference (ITC), pp 506–514. IEEE
30. Ma Y, Ren H, Khailany B, Sikka H, Luo L, Natarajan K, Yu B (2019) High performance graph convolutional networks with applications in testability analysis. In: Proceedings of the 56th annual design automation conference 2019, pp 1–6
31. Ma Y, He Z, Li W, Zhang L, Yu B (2020) Understanding graphs in EDA: from shallow to deep learning. In: Proceedings of the 2020 international symposium on physical design, pp 119–126
32. Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907
33. Khailany B, Ren H, Dai S, Godil S, Keller B, Kirby R, Klinefelter A, Venkatesan R, Zhang Y, Catanzaro B et al (2020) Accelerating chip design with machine learning. IEEE Micro 40(6):23–32
34. Hamilton WL (2020) Graph representation learning. Synth Lect Artif Intell Mach Learn 14(3):1–159
35. Jumper J, Evans R, Pritzel A, Green T, Figurnov M, Ronneberger O, Tunyasuvunakool K, Bates R, Žídek A, Potapenko A et al (2021) Highly accurate protein structure prediction with alphafold. Nature 596(7873):583–589
36. Derrow-Pinion A, She J, Wong D, Lange O, Hester T, Perez L, Nunkesser M, Lee S, Guo X, Wiltshire B, et al. (2021) Eta prediction with graph neural networks in google maps. In: Proceedings of the 30th ACM international conference on information & knowledge management, pp 3767–3776
37. Stokes JM, Yang K, Swanson K, Jin W, Cubillos-Ruiz A, Donghia NM, MacNair CR, French S, Carfrae LA, Bloom-Ackermann Z et al (2020) A deep learning approach to antibiotic discovery. Cell 180(4):688–702
38. Dey M, Mia SM, Sarkar N, Bhattacharya A, Roy S, Malakar S, Sarkar R (2021) A two-stage CNN-based hand-drawn electrical and electronic circuit component recognition system. Neural Comput Appl 33(20):13367–13390
39. Ustun E, Deng C, Pal D, Li Z, Zhang Z (2020) Accurate operation delay prediction for fpga hls using graph neural networks. In: Proceedings of the 39th international conference on computer-aided design, pp 1–9
40. Mirhoseini A, Goldie A, Yazgan M, Jiang JW, Songhori E, Wang S, Lee Y-J, Johnson E, Pathak O, Nazi A et al (2021) A graph placement methodology for fast chip design. Nature 594(7862):207–212
41. Xie Z, Liang R, Xu X, Hu J, Duan Y, Chen Y (2021) Net 2: A graph attention network method customized for pre-placement net length estimation. In: 2021 26th Asia and South Pacific design automation conference (ASP-DAC), pp 671–677. IEEE
42. Agnesina A, Chang K, Lim SK (2020) Vlsi placement parameter optimization using deep reinforcement learning. In: Proceedings of the 39th international conference on computer-aided design, pp 1–9
43. Lu Y-C, Pentapati S, Lim SK (2020) Vlsi placement optimization using graph neural networks. In: 34th advances in neural information processing systems (NeurIPS) workshop on ML for systems
44. Hamilton W, Ying Z, Leskovec J (2017) Inductive representation learning on large graphs. Adv Neural Inf Process Syst **30**
45. Ren H, Kokai GF, Turner WJ, Ku T-S (2020) Paragraph: layout parasitics and device parameter prediction using graph neural networks. In: 2020 57th ACM/IEEE design automation conference (DAC), pp 1–6. IEEE
46. Zhang G, He H, Katabi D (2019) Circuit-GNN: Graph neural networks for distributed circuit design. In: International conference on machine learning, pp 7364–7373. PMLR
47. Wang H, Wang K, Yang J, Shen L, Sun N, Lee H-S, Han S (2020) Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In: 2020 57th ACM/IEEE design automation conference (DAC), pp 1–6. IEEE
48. Li Y, Lin Y, Madhusudan M, Sharma A, Xu W, Sapatnekar SS, Harjani R, Hu J (2020) A customized graph neural network model for guiding analog IC placement. In: 2020 IEEE/ACM international conference on computer aided design (ICCAD), pp 1–9. IEEE
49. Abualigah L, Shehab M, Alshinwan M, Alabool H (2020) Salp swarm algorithm: a comprehensive survey. Neural Comput Appl 32(15):11195–11215
50. Tsitsulin A, Mottin D, Karras P, Bronstein A, Müller E (2018) Netlsd: hearing the shape of a graph. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2347–2356
51. Said A, Hassan S-U, Tuarob S, Nawaz R, Shabbir M (2021) Dgsd: Distributed graph representation via graph statistical properties. Futur Gener Comput Syst 119:166–175
52. Zhang M, Chen Y (2018) Link prediction based on graph neural networks. Adv Neural Inf Process Syst **31**
53. Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y (2017) Graph attention networks. arXiv preprint arXiv:1710.10903
54. Xu K, Hu W, Leskovec J, Jegelka S (2018) How powerful are graph neural networks?. arXiv preprint arXiv:1810.00826
55. Zhang M, Li P (2021) Nested graph neural networks. Adv Neural Inf Process Syst 34:15734–15747
56. Ahmed A, Hassan ZR, Shabbir M (2020) Interpretable multi-scale graph descriptors via structural compression. Inf Sci 533:169–180
57. Verma S, Zhang Z-L (2017) Hunt for the unique, stable, sparse and fast feature learning on graphs. Adv Neural Inf Process Syst **30**
58. Berlingerio M, Koutra D, Eliassi-Rad T, Faloutsos C (2013) Network similarity via multiple social theories. In: Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining, pp 1439–1440
59. Shervashidze N, Schweitzer P, Van Leeuwen EJ, Mehlhorn K, Borgwardt KM (2011) Weisfeiler-Lehman graph kernels. J. Mach. Learn. Res. 12(9):2539–2561
60. Weisfeiler B, Leman A (1968) The reduction of a graph to canonical form and the algebra which appears therein. NTI, Ser 2(9):12–16
61. Borgwardt KM, Kriegel H-P (2005) Shortest-path kernels on graphs. In: Fifth IEEE international conference on data mining (ICDM'05), p 8. IEEE
62. Cangea C, Veličković P, Jovanović N, Kipf T, Liò P (2018) Towards sparse hierarchical graph classifiers. arXiv preprint arXiv:1811.01287
63. Zhang M, Cui Z, Neumann M, Chen Y (2018) An end-to-end deep learning architecture for graph classification. In: Proceedings of the AAAI conference on artificial intelligence, vol 32