Customizing Graph Neural Network for CAD Assembly Recommendation

Fengqi Liang*
Beijing University of Posts and Telecommunications
Beijing, China
lfq@bupt.edu.cn

Huan Zhao*
4Paradigm Inc.
Beijing, China
zhaohuan@4paradigm.com

Yuhan Quan
4Paradigm Inc.
Beijing, China
quanyuhan@4paradigm.com

Wei Fang
Beijing University of Posts and Telecommunications
Beijing, China
fang_wei@bupt.edu.cn

Chuan Shi†
Beijing University of Posts and Telecommunications
Beijing, China
shichuan@bupt.edu.cn

ABSTRACT

CAD assembly modeling, which refers to using CAD software to design new products from a catalog of existing machine components, is important in the industrial field. The graph neural network (GNN) based recommender system for CAD assembly modeling can help designers make decisions and speed up the design process by recommending the next required component based on the existing components in CAD software. These components can be represented as a graph naturally. However, present recommender systems for CAD assembly modeling adopt fixed GNN architectures, which may be sub-optimal for different manufacturers with different data distribution. Therefore, to customize a well-suited recommender system for different manufacturers, we propose a novel neural architecture search (NAS) framework, dubbed CusGNN, which can design data-specific GNN automatically. Specifically, we design a search space from three dimensions (i.e., aggregation, fusion, and readout functions), which contains a wide variety of GNN architectures. Then, we develop an effective differentiable search algorithm to search high-performing GNN from the search space. Experimental results show that the customized GNNs achieve 1.5-5.1% higher top-10 accuracy compared to previous manual designed methods, demonstrating the superiority of the proposed approach. Code and data are available at https://github.com/BUPT-GAMMA/CusGNN.

CCS CONCEPTS

• **Information systems** → **Data mining**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Computer-Aided Design, Recommender System, Graph Neural Networks, Neural Architecture Search

ACM Reference Format:
Fengqi Liang, Huan Zhao, Yuhan Quan, Wei Fang, and Chuan Shi. 2024. Customizing Graph Neural Network for CAD Assembly Recommendation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3637528.3671788

1 INTRODUCTION

Computer-aided design (CAD), which refers to adopting computers to support the creation, modification, analysis, or optimization of designs, has been widely applied to three-dimensional mechanical design [31] and these designed physical products surround us all the time [14]. As one of the basic application scenarios, assembly modeling in CAD refers to the engineering process of designing new products from a common catalog of existing machine components. For example, as shown in Figure 2(f), a truck consists of several components (base, tires, axle, etc). To design such a product, two processes are required: importing the required components into the CAD software, e.g., FreeCAD, and assembling the imported components together.

Several works [14, 41] focus on how to assemble the imported components together by using neural networks to predict whether different components are geometrically mated. However, few works consider how to import the required components into the CAD software. To do so, the engineers have to pick and insert each component from the catalog (a component library), which contains thousands of components, into CAD software one by one manually, which is a tedious task and leads to the waste of human resources, especially for more complex product designing. As mentioned in [14], designers spend **one third** of their time within CAD sofeware for assembly task. If we could build a recommender system to recommend the next selected component from the catalog for a CAD system, it could accelerate the design process by reducing the frequency with which designers select the next component directly from the catalog. Nevertheless, this type of recommendater system for the **n**ext **c**omponent **r**ecommendation (NCR) task is distinct

*Equal contribution.
†Corresponding author.

KDD '24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-0490-1/24/08…$15.00
https://doi.org/10.1145/3637528.3671788

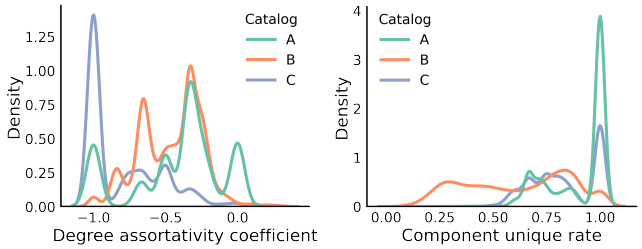


Figure 1: This figure shows that data distribution for different catalogs from different manufacturers varies greatly, which inspires us to consider how to customize data-specific GNN architectures for different manufacturers. We show the data distribution using kernel density estimation [32]. Degree assortativity coefficient [28] represents the graph structure information for an assembly. Component unique rate denotes the number of component types divided by the total number of components in an assembly.

from the conventional recommender systems. As shown in Figure 2, it has the following characteristics: 1) The recommender system in CAD system is shared for all designers, and which component should be recommended is determined by the current state, i.e., the components that have already been assembled. Therefore, the personalized characteristic about the designers may not be modeled; 2) The current state (partially assembly in Figure 2(b)) could be modeled as a graph naturally, i.e., treating the components already in CAD system as nodes and the mating conditions [14, 41], which define the relative positions of components to each other and can be read off by geometric proximity in CAD systems, as edges. Then capturing graph structure is just capturing the relative positions of components to further extract design intent which facilitates to recommend next required component.

Very recently, Gajek et al. [17] proposed to model the NCR task in CAD system as a graph classification problem (i.e, classifying current partial graph to the class of the next component) and employ graph neural networks (GNNs), such as GCN [16] and GAT [33], to learn partial graph representation. Nevertheless, the proposed framework [17] relies on fixed GNN architectures designed by human experts. In reality, different manufacturers have their unique catalogs (e.g., a 3D product data platform like Partsolutions[1] has over 3,000 certified manufacturer catalogs), and as shown in Figure 1, we could observe that both graph structure distribution and component type distribution vary widely on three benchmark catalogs from different manufacturers. More visually, in Figure 9, we could easily observe that different assemblies have different shapes. Thus, using fixed models directly may be suboptimal for different data distribution due to the *no free lunch theorem* [43], which state that no one-size-fits-all model that works well for every task. In other words, different manufacturers may benefit from customizing data-specific GNN based recommender systems which are suitable for their individual data distribution. Then a natural and practical question can be asked: *How to customize well-performing GNN based recommender systems for different manufacturers automatically?*

[1]https://partsolutions.com/ecatalogsolutions/

In this paper, we propose a novel framework, dubbed CusGNN (**Cus**tomized **G**raph **N**eural **N**etwork), to address this problem based on the neural architecture search (NAS) techniques which have made great success in designing graph neural networks automatically [8, 11, 38, 40, 47]. Specifically, we design a search space that contains three dimensions (i.e., aggregation functions, fusion functions, and readout functions) for GNN based NCR problem. Each dimension contains several typical candidate operations. Besides, we add two candidate operations "Identity" and "Zero" to decide whether the outputs of aggregation functions in the previous GNN layers should be used as input to the next layer, thereby capturing representations from different layers adaptively. All the operations in these dimensions constitute a supernet, and we adopt an efficient differentiable search algorithm to search a suitable operation for each dimension automatically. To demonstrate the effectiveness of our proposed approach, we conduct experiments on three datasets from three different manufacturers. Experimental results show that our approach achieved state-of-the-art (SOTA) performance on all catalogs with a 1.5-5.1% performance improvement in top-10 accuracy compared to previous manually designed methods.

To summarize, this work makes the following contributions:

- To the best of our knowledge, we are the first to consider how to automatically customize data-specific GNN based recommender systems for CAD assembly modeling problems in different manufacturers, which is significant for the industrial application of recommender systems and GNNs.
- We propose an effective NAS framework, CusGNN, for customizing data-specific GNN based recommender systems. In this framework, we design a search space from three basic design dimensions of GNNs and adopt an efficient differentiable search algorithm to automatically search data-specific GNN architectures.
- The experiments show that CusGNN improve the performance in a large margin compared to existing baselines on benchmark datasets from three different manufacturers, demonstrating the superiority of CusGNN.

Relevance to KDD. Assembly modeling in CAD is a prevalent challenge within the industry. We address this problem as a next component recommendation task, aiming to recommend the next component required based on the partial assembly. The partial assembly naturally forms a graph structure, therefore it is very suitable to deal with CAD assembly tasks by graph learning approach. Graph learning and graph neural networks are well-established in the field of data mining and considerable papers which adopt graph learning approach for CAD assembly task have been published in data mining conferences [17, 26, 50], emphasizing the significance of our work in data mining domain. Thus, our research aligns closely with the overarching theme of the data mining conference.

2 RELATED WORKS

2.1 Recommender Systems for CAD software and assembly modeling

Recommender systems for CAD software have been widely used to assist human designers in making design decisions, and various approaches have been proposed. Li et al. [20] adopt collaborative filtering to recommend useful software commands to improve the

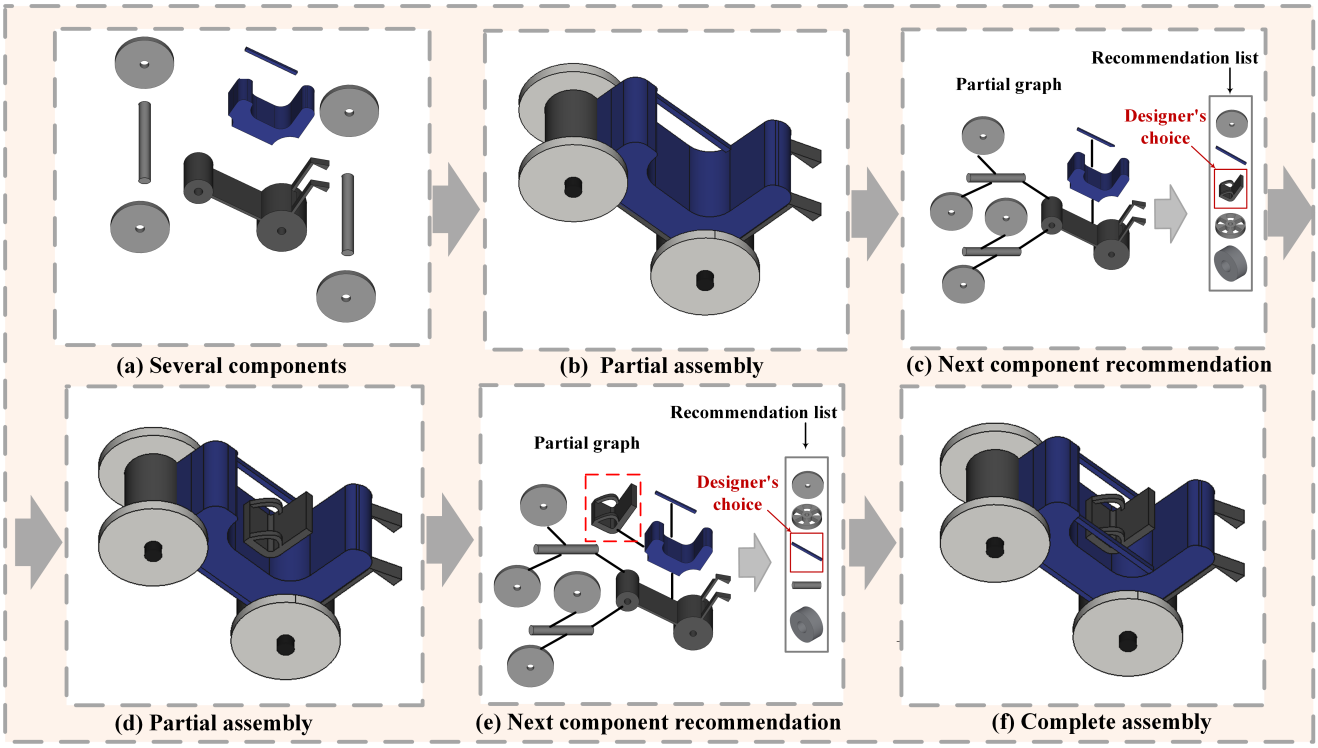


Figure 2: This figure presents an overview of recommender systems for CAD assembly modeling. In (a)-(b), multiple components can be assembled using CAD software. In (b)-(e), the recommender systems provide the designer with a list of recommendations based on the partial assembly already present in the CAD software, which can be treated as a partial graph. The designer then selects a component to use. Finally, in (f), the complete assembly is obtained after the design process.

workflow. Although the same motivation to aid design, our approach focuses on recommending the next component to be used for assembly modeling rather than commands for CAD software. Assembly retrieval [25] adopts information retrieval based technology to give a query assembly for similar assembly designs. There has also been recent work [30] using GNN to model CAD assemblies so that given a CAD assembly, other similar assemblies can be retrieved. However, these work focus on similar assembly retrieval rather than aiding in the design process of an assembly like our approach. Some generative approaches [7, 50] either generate the sequence of CAD-typical geometrical operations to design a component or generate the complete 3D shape of an assembly from randomly placed components in 3D meshes, which are orthogonal to our approach. MultiCAD [26] utilizes multimodal contrastive learning to enhance the 3D assembly representation which is also orthogonal to our approach. Several works [14, 41] focus on predicting whether different components are geometrically mated to assemble the imported components rather than exploring how to import required components and these two tasks are complementary. Gajek et al. [17] proposed a GNN-based approach to recommend the next required component for assembly modeling. In their settings, if rolled out step by step, the NCR task can be seen as a graph generative task [9] and other graph generative models [23] not applicable to this task because these methods are unstable to train and may lead to disconnected graphs [17]. However, their approach cannot customize the most suitable models for different manufacturers automatically, and their proposed framework requires a pre-training phase, which is not convenient. In contrast, our proposed approach, CusGNN, uses neural architecture search to customize data-specific GNN models for different manufacturers automatically without the need for pre-training, achieving state-of-the-art performance on benchmark datasets.

2.2 Graph Neural Networks

Graph neural networks have been the state-of-the-art approaches to address graph data [16, 33]. GNN papers tend to focus on designing more powerful and expressive aggregation functions [2, 16, 33, 45] that define how to aggregate the information from the neighborhood and update the node representation more efficiently. Some papers [12, 18, 46] focus on making GNN deeper by stacking multi-layer. This approach allows GNNs to aggregate messages from multi-hop neighbors to expand the receptive field of each node. However, the performance of GNNs may decrease as the layer goes deeper due to the problem of over-smoothing [19]. To address this problem, some papers propose to fuse multi-layer node representations by adding skip-connection [12, 18, 46]. In addition, in graph classification task, researchers tend to study the readout functions which are adopted to pool all node representations and get the graph-level representation for graph classification tasks [3, 45].

Researchers have tended to design GNN architectures [8, 11, 21, 36–39, 47, 51] using NAS approaches [6, 24, 52] automatically.

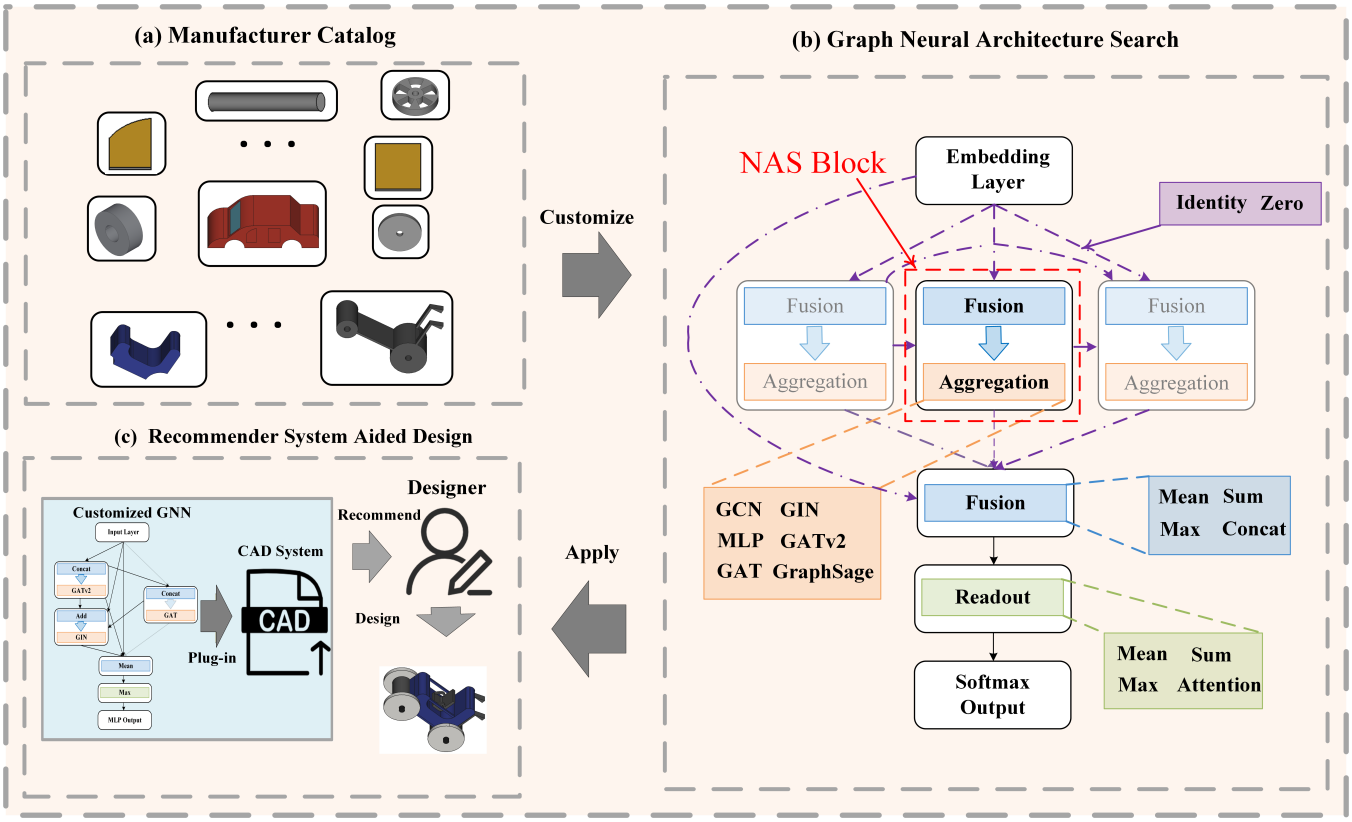


Figure 3: (a) The catalog for one manufacturer. Different manufacturers have their unique catalog. (b) The CusGNN framework to customize GNN based recommender systems for different manufacturers. CusGNN could design data-specific GNN architecture by choosing the data-specific aggregation functions, fusion functions, readout functions, and selection operations automatically. The number of NAS blocks could be any number and we use three blocks as an illustrative example. (c) The customized recommender system could be a plug-in for CAD systems and be used to assist designers in designing products.

These approaches focus on designing the aggregation layers, e.g., GraphNAS [8] and GraphGym [47] provide diverse dimensions and candidate operations to design the GNN layers. Additionally, approaches such as [11, 21, 38] add the skip-connections operation into their search space. PAS [40] is the sota NAS approach for graph classification task by considering hierarchical pooling design dimension. However, there is still a need to explore how to customize GNN recommender systems for next component recommendation and which design dimensions are more appropriate for this question.

3 METHODOLOGY

In this section, we present the details of CusGNN. First, we define the NCR problem and review the GNN approaches for this problem. Then we show CusGNN search space for NCR problem. Finally, we present the differentiable search algorithm for CusGNN. An overview of the proposed framework is shown in Figure 3.

3.1 GNN for NCR problem

We begin by defining the task of the next component recommendation (NCR) problem. Given a partial graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that represents the current state in a CAD system, where the node set $\mathcal{V}$ represents the components already present in the CAD system and an edge $e = (i, j) \in \mathcal{E}$ denotes that component i and component j are connected in the CAD system. It is also associated with a component type mapping function $\phi : \mathcal{V} \rightarrow \mathcal{T}$ as the node feature. Our goal is learning the conditional probability distribution $P(\mathcal{T}_{next}|\mathcal{G})$ from the dataset $\mathcal{D} = \{(\mathcal{G}_1, t_1), \cdots, (\mathcal{G}_n, t_n)\}$, where $\mathcal{G}_i$ is the partial graph and t_i is the corresponding label which represents the next component to be used. We intend to train a classifier $f \circ g$ that maps the partial graph $\mathcal{G}$ to its most probable next component $\hat{t}_{max} \in \mathcal{T}$, i.e.,

$$\hat{t}_{max} = \arg\max_{\hat{t} \in \mathcal{T}} f(\mathbf{z}_G), \text{where } \mathbf{z}_G = g(\theta, \mathcal{G}), \tag{1}$$

where f denotes the softmax function, g denotes the encoder to extract the partial graph representation, and θ denotes the learnable parameters of g.

As GNNs are the most powerful encoders to learn the representation of graph structure data, Gajek et al. [17] naturally propose to employ GNNs as the encoder to learn partial graph representation for NCR problem. Their approach consists of two phases. In the first phase, they utilize a variant of a well-known pre-training technique word2vec [27] to train component representations that serve as

inputs to the GNNs in the second phase. To do so, they first identify the neighbor components within n-hop for each component in every graph. They then use the skip-gram variant of word2vec [27] to train the component representations. This training loss of their proposed word2vec variant can be formulated as follows:

$$\begin{aligned} L &= -\frac{1}{|\mathcal{D}|}\sum_{\mathcal{G}_i\in\mathcal{D}}^{|\mathcal{D}|}\sum_{v_j\in\mathcal{G}_i}^{|\mathcal{N}_i|}\sum_{v_k\in\mathcal{N}_n(j)}^{|\mathcal{N}_n(j)|}\log P\left(w_k \mid w_j\right), \\ \text{where } P\left(w_k \mid w_j\right) &= \frac{\exp\left(\mathbf{o}_{\phi(v_k)}^T\mathbf{h}_{\phi(v_j)}\right)}{\sum_{t\in\mathcal{T}}\exp\left(\mathbf{o}_t^T\mathbf{h}_{\phi(v_j)}\right)}, \end{aligned} \tag{2}$$

where $\mathcal{N}_n(j)$ is the neighbor components within n-hop of node j, $\mathbf{o}_t$ and $\mathbf{h}_t$ are the corresponding input and output representation vector of component type t, respectively. In the second phase, they adopt GCN [16] and GAT [33] to further extract partial graph representation to recommend the next required component.

3.2 GNN Search Space for NCR problem

As mentioned in the introduction, different manufacturers have their unique catalogs with different data distribution and therefore require different GNN architectures. Inspired by this demand, in this paper, we design a search space that supports the customization of GNN architectures for different catalogs from different manufacturers. Our search space does not consider hierarchical pooling like PAS [40] and we consider the following dimensions.

3.2.1 Aggregation functions. Aggregation functions are crucial in determining how to aggregate information from the neighboring components and update the representations of the current component. As shown in [17], different aggregation functions (e.g., GAT and GCN) perform differently on different catalogs which motivated us to consider aggregation functions as a design dimension in our search space. The aggregation functions can be formulated as follows:

$$\begin{aligned} \mathbf{h}_v^{(l+1)} &= \text{UPDATE}^{(l)}\left(\mathbf{k}_v^{(l)}, \mathbf{m}_{\mathcal{N}(v)}^{(l)}\right), \\ \text{where } \mathbf{m}_{\mathcal{N}(v)}^{(l)} &= \text{AGGREGATE}^{(l)}\left(\left\{\mathbf{k}_u^{(l)} \mid \forall u\in\mathcal{N}(v)\right\}\right), \end{aligned} \tag{3}$$

where $\mathbf{h}_v^{(l+1)}$ is the hidden embedding of node $v\in\mathcal{V}$ updated based on aggregated information from v's neighborhood $\mathcal{N}(v)$ in graph at the l-th layer. Note that the information $\mathbf{k}^{(l)}$ used in the update is the component representation after applying fusion functions. We refer to the combination of AGGREGATE and UPDATE operations as an aggregation function.

In this work, we consider six aggregation functions, including GCN [16], GAT [33], GraphSage [10], GIN [45], GATv2 [2], and MLP functions which focus on component own information rather neighbor information.

3.2.2 Fusion functions. The feature fusion functions combine the output representations of components from different GNN layers, enabling components to adaptively fuse information within multi-hop neighborhoods. This approach can be beneficial for recommending suitable components for the same component within different multi-hop neighborhood relationships. As suggested in [38], the fusion function involves two steps: the selection step and the fusion step. This can be formulated as follows:

$$\mathbf{k}_v^{(l)} = \text{FUSE}^{(l)}\left(\left\{\text{SEL}_i^l(\mathbf{h}_v^{(i)}) \mid 0\le i\le l\right\}\right), \tag{4}$$

where SEL_i^l represents the select functions that determine whether the i-th layer hidden embedding is adopted by the fusion function at layer l.

In the selection step, the selection operation determines whether features from different previous layers are chosen for the fusion step. Specifically, two operations Identity and Zero are considered in our framework, which can be formulated as $\text{SEL}(\mathbf{h}) = \mathbf{h}$ and $\text{SEL}(\mathbf{h}) = \mathbf{0}\cdot\mathbf{h}$ respectively.

As for the fusion step, we provide four fusion operations to fuse the selected different layer features: Sum, Mean, Max, and Concat. These can be formulated as:

$$\begin{aligned} &\text{Sum:} && \mathbf{k}_v^{(l)} = \text{SEL}(\mathbf{h}_v^{(1)}) + \cdots + \text{SEL}(\mathbf{h}_v^{(l)}), \\ &\text{Max:} && \mathbf{k}_v^{(l)} = \max\left(\text{SEL}(\mathbf{h}_v^{(1)}),\ \cdots,\ \text{SEL}(\mathbf{h}_v^{(l)})\right), \\ &\text{Mean:} && \mathbf{k}_v^{(l)} = \frac{1}{l}\left(\text{SEL}(\mathbf{h}_v^{(1)}) + \cdots + \text{SEL}(\mathbf{h}_v^{(l)})\right), \\ &\text{Concat:} && \mathbf{k}_v^{(l)} = \left[\text{SEL}(\mathbf{h}_v^{(1)}) \| \cdots \| \text{SEL}(\mathbf{h}_v^{(l)})\right], \end{aligned} \tag{5}$$

where $\mathbf{k}_v^{(l)}$ replaces the output component feature $\mathbf{h}_v^{(l)}$ at l layer as the input to the next aggregation layer. In our experiments, we found that the flexibility of choosing different fusion functions for different layers plays a crucial role in the final architecture performance for different catalogs.

As shown in Figure 3(b), we act on a fusion function after each aggregation function and refer the combination of two functions to **NAS block**.

3.2.3 Readout functions. The readout functions define how to obtain the partial assembly representation for the NCR problem by pooling all components' representations. It can be formulated as follows:

$$\mathbf{z}_{\mathcal{G}} = \text{READOUT}\left(\left\{\mathbf{k}_v^L \mid \forall v\in\mathcal{V}\right\}\right). \tag{6}$$

In addition to the three basic readout operations (Mean, Max, and Sum), we add an adaptive approach Attention [22] to the search space in this work. The attention-based approach can be formulated as follows:

$$\mathbf{z}_{\mathcal{G}} = \sum_{v\in\mathcal{V}}\text{softmax}\left(\mathbf{W}\mathbf{k}_v^L\right)\mathbf{k}_v^L, \tag{7}$$

where $\mathbf{W}$ is the learnable weight vector for attention. The attention based readout could adaptively adjust the contribution of each component representation to the whole partial assembly representation.

In summary, our **main technological innovation** lies in exploring how to design a suitable GNN search space to encode components and partial assemblies with different distribution in NCR tasks from various manufacturers. Our search space includes three design dimensions and excludes one design dimension. The selected operations within each design dimension embody versatile and well-established choices, subjected to thorough scrutiny in practical applications, thus affirming their effectiveness. Consequently, our defined search space ensures the automatic design of high-performance GNN architectures capable of handling data from diverse manufacturers in real-world scenarios.

3.3 Differentiable Architecture Search

The search process for CusGNN to search data-specific GNN achitecture is formulated as follows:

$$\begin{aligned} g^* &= \arg\max_{g\in\mathcal{A}} \mathbb{E}_{(\mathcal{G}_i,t_i)\in\mathcal{D}_{\text{val}}} \left[M\left(\boldsymbol{t}_i, P\left(\hat{\boldsymbol{t}}_i \mid g\left(\theta^*\right)\right)\right)\right], \\ \text{s.t. : } \theta^* &= \arg\min_{\theta} \ell\left(g\left(\theta\right), \mathcal{D}_{\text{train}}\right), \end{aligned} \tag{8}$$

where ℓ is loss function, M is the evaluation metric for NCR tasks, $\mathcal{D}_{\text{train}}$ denotes the training set, and $\mathcal{D}_{\text{val}}$ the valid set correspondingly. As shown in Figure 2 (c) and (e), our NCR task provides a list of recommendations that could be regarded as classifying the partial assembly to the next component to be used. Thus, we use the cross-entropy function, which is commonly used for classification tasks, as the loss function ℓ and the evaluation metric M.

In this paper, we employ a differentiable search algorithm for its demonstrated efficiency and effectiveness in previous works such as [24, 38, 44]. The key idea behind the differentiable algorithm is to relax the discrete selection of operations into a continuous weighted summation of all possible operations which can be regarded as a supernet. This process can be formulated as:

$$\begin{aligned} \mathbf{z}_{\mathcal{G}} &= \sum_{i=1}^{|O_r|} c_i^r o_i\left(\mathcal{G}, \mathbf{k}^L\right), \ \mathbf{h}_v^l = \sum_{i=1}^{|O_a|} c_i^{l,a} o_i\left(\mathcal{G}, \mathbf{k}_v^{l-1}\right), \\ \mathbf{k}_v^l &= \sum_{i=1}^{|O_f|} c_i^{l,f} o_i\left(\bar{s}^{l,0}(\mathbf{h}_v^0), \cdots, \bar{s}^{l,l}(\mathbf{h}_v^l)\right), \\ \bar{s}^{l,j}\left(\mathbf{h}_v^j\right) &= \sum_{i=1}^{|O_s|} c_i^{l,j} o_k^{l,j}\left(\mathbf{h}_v^j\right) = c_1^{l,j}\mathbf{0} + c_2^{l,j}\mathbf{h}_v^j, \end{aligned} \tag{9}$$

where L is the final layer, and l ranges from 0 to L. O_r, O_a, O_f, and O_s are sets of the readout, aggregation, fusion, and selection operation set, respectively. For each operation selection weight c, we use the Gumbel-Softmax [13] to approximate the discrete distribution of selecting operations in each dimension. This approach allows us to effectively train the supernet [5, 44]. This process can be formulated as:

$$c_i = g(\boldsymbol{\alpha}) = \frac{\exp\left(\left(\log \boldsymbol{\alpha_i} - \log\left(-\log\left(\mathbf{U}_i\right)\right)\right)/\lambda\right)}{\sum_{j=1}^{|O|}\exp\left(\left(\log \boldsymbol{\alpha}_j - \log\left(-\log\left(\mathbf{U}_i\right)\right)\right)/\lambda\right)}, \tag{10}$$

where $\boldsymbol{\alpha}_i$ is the corresponding supernet parameter for c_i, $\mathbf{U}_i$ is a uniform random variable, and λ is the temperature of Softmax. We update the GNN parameters and the parameters $\boldsymbol{\alpha}$ using the training loss and valid loss respectively. After the search process, we select the subnetwork consisting of the operations with the **highest** parameters $\boldsymbol{\alpha}$ as the **searched** architecture, and **retrain** it to obtain the final model. As shown in Figure 3 (c), the retrained subnetwork can be integrated into a CAD system as a plug-in to assist designers in making component choices by providing a recommendation list based on the top-k predicted scores.

4 EXPERIMENTS

4.1 Experimental Settings

4.1.1 Datasets and Metrics. We utilize three different real-world catalogs (datasets) from different manufacturers in previous study [17] and use the original division with 6:2:2 train/validation/test assemblies. We provide more statistical information of these datasets is presented in Table 5. From the table, we can further observe that these catalogs differ significantly in many aspects, such as the number of components, nodes, edges, and so on. This further underscores the significance of customizing GNNs. For the metric, we follow the top-k rate in [17], which refers to the percentage of the true component ID (i.e., the label) being present in the top-k predictions of a model. As mentioned in [17], k=10 is much more important as this number of recommendations can be well integrated into a CAD system and provides a diverse range of component options for designers. For more datasets and metric details, we refer readers to Appendix A.

4.1.2 Baselines. We compare our proposed methods with three types of baselines: **1) Rule-based Methods**, which includes **F**requency **Base**line (FBase) [17] that predicts the k most frequent labels for a given partial graph in the training set. If the graph has not been seen in the training set, it looks for the largest previously seen subgraphs that together form the partial graph. **Evergreen** [17] predicts the k most common labels in the training set. **2) GNN based methods**, which includes **GCN** [16] and **GAT** [33]. We follow the implementation in [17], where the node features are the one-hot encoding for corresponding component types. We also compared against the pre-training based methods: $\mathbf{GCN}_{20}$, $\mathbf{GCN}_{100}$, $\mathbf{GAT}_{20}$, and $\mathbf{GAT}_{100}$ in [17]. The numbers 20 and 100 denote the dimensions of the pre-trained component type embeddings. However, we find information leakage and unfair comparisons in their source code. We fix the code and the fixed models are denoted with a star *. A detailed description of the unreasonable source code can be found in Appendix B. **3) PAS** [40] is the SOTA GNN and NAS method for graph classification task which consider hierarchical pooling dimension in their search space. We apply PAS to the NCR task as a strong baseline.

4.1.3 Implementation Details. For the implementation details and hyperparameters of CusGNN and other baselines, we refer readers to Appendix C.

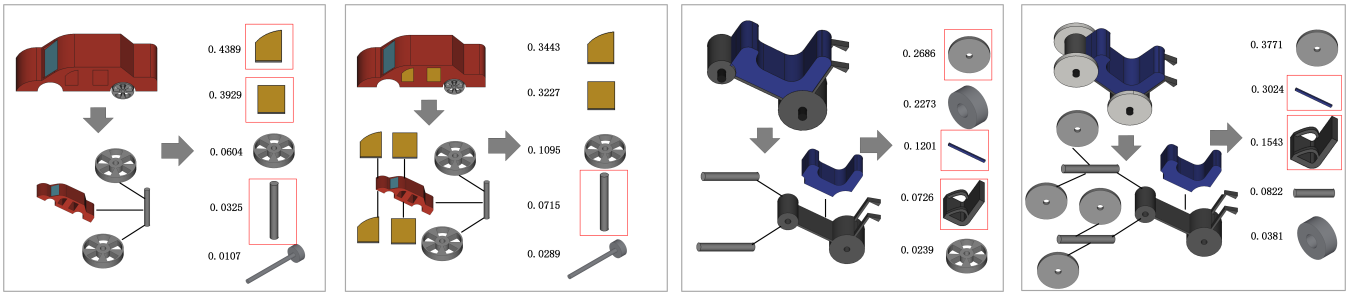
4.2 Performance Comparison

The results are shown in Table 1. From the table, we emphasize the following observations:

• CusGNN outperforms all baselines on almost all metrics, except for the top-1 rate of the frequency baseline. This demonstrates the effectiveness of our approach in customizing GNN models for different manufacturers. As mentioned in [17], since our recommender system focuses on presenting the designers with more than one component, the case k = 1 could be negligible. Since the top-10 rate is a more important metric, we provide a detailed analysis based on it. Our approach achieves a 1.5-5.1% performance improvement on the top-10 rate compared to previous baselines for NCR task on the three catalogs, respectively. We also visualize the searched SOTA architectures for the three different catalogs from different manufacturers in Figure 5. The searched SOAT architectures vary greatly, which further illustrates the importance of customizing data-specific GNNs for different manufacturers. Furthermore, from Table 5, we observe that our approach achieves more performance

Table 1: Top-k accuracy on the test set for $k = 1; \ldots; 20$ recommendations. The results of baselines come from [17]. The baselines with star $*$ denote the pre-training baselines after fixing the code and the original results are in [17].

Catalog	k \ model	Upper Bound	GAT^*_{100}	GAT^*_{20}	$GAT_{one-hot}$	GCN^*_{100}	GCN^*_{100}	$GCN_{one-hot}$	FBase	Evergreen	PAS	CusGNN
A	1	94.1%	37.5%	43.8%	46.0%	44.5%	41.0%	45.4%	**57.5 %**	4.5%	47.1%	49.2%
	2	99.8%	59.2%	67.1%	70.3%	68.0%	63.1%	69.3%	64.3%	6.3%	71.5%	**73.6 %**
	3	99.9%	69.6%	75.9%	79.0%	76.9%	73.1%	78.1%	66.9%	8.1%	79.9%	**82.2 %**
	5	99.9%	78.0%	81.8%	84.6%	82.9%	80.4%	83.8%	69.0%	11.3%	85.3%	**87.3 %**
	10	100%	85.3%	86.7%	89.6%	87.8%	86.2%	88.6%	70.2%	15.9%	89.9%	**91.7%**
	15	100%	88.1%	88.9%	91.5%	89.8%	88.4%	90.7%	70.7%	19.6%	91.9%	**93.6 %**
	20	100%	89.6%	90.1%	92.6%	91.0%	89.6%	92.0%	70.8%	22.2%	92.9%	**94.6%**
B	1	63.5%	22.9%	25.0%	30.2%	26.1%	23.8%	27.8 %	24.5 %	13.3 %	29.3%	**32.6%**
	2	81.5 %	38.5%	41.8%	49.5%	42.9%	40.0%	45.1 %	32.9 %	15.5 %	49.0%	**53.3%**
	3	92.5 %	49.8%	53.6%	61.8 %	54.7%	51.3%	56.8 %	39.1 %	18.4 %	61.9%	**66.4%**
	5	99.4 %	64.0%	67.5%	72.5 %	68.3%	65.3%	69.2 %	46.6 %	22.2 %	74.1%	**78.0%**
	10	99.99 %	77.1	79.5%	80.7 %	80.9%	78.8%	79.8 %	52.8 %	30.0 %	83.5%	**86.0%**
	15	100 %	82.5%	84.2%	84.6 %	85.7%	84.1	84.3 %	53.7 %	36.2 %	87.4%	**89.3%**
	20	100 %	85.8%	86.9%	86.6 %	88.4%	87.1%	86.7 %	53.8 %	41.7 %	89.8%	**91.2%**
C	1	74.0%	22.2%	24.1%	27.5%	23.0%	25.0%	27.0%	**30.1 %**	14.4%	27.4%	28.7%
	2	91.1%	39.7%	43.6%	48.4%	41.3%	44.7%	47.1%	42.1%	19.9%	48.3%	**50.0%**
	3	98.0%	53.8%	58.8%	64.0%	55.8%	60.2%	62.4%	50.9%	23.3%	64.1%	**65.9%**
	5	99.9%	72.2%	77.1%	80.6%	74.2%	78.5%	79.2%	60.2%	27.8%	82.0%	**83.0%**
	10	100%	87.7%	89.5%	91.2%	88.3%	90.5%	90.8%	63.9%	37.0%	92.4%	**92.7%**
	15	100%	91.9%	92.6%	94.0%	92.0%	94.0%	94.2%	64.2%	44.6%	95.3%	**95.4%**
	20	100%	93.6%	94.1%	95.2%	93.7%	95.0%	95.7%	64.2%	48.1%	**96.5%**	**96.5%**



(a) (b) (c) (d)

Figure 4: The visualization of the top-5 recommendation list of four partial assemblies in our internal scenario. These numbers represent the predicted probabilities of the next required components and the components in the red box are the ground truth of the next required components.

improvement on more complex catalog B, which further demonstrates the superiority of our approach in aiding more complex product designs.

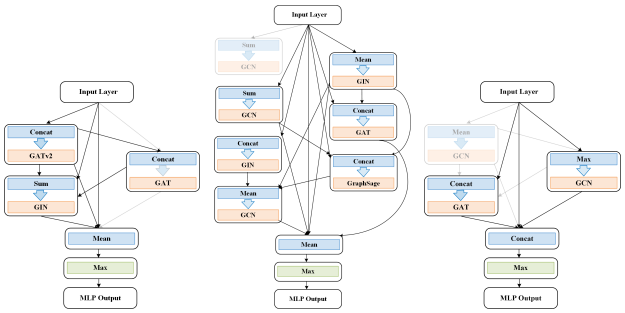
• CusGNN outperforms PAS [40], the SOTA method in graph classification task in all cases for NCR task, indicating that our search space is better suited for NCR tasks and the design dimension of hierarchical pooling may not be well-suited for this task. One possible reason is that the partial graphs are small (As shown in Table 5, the average number of nodes is less than 20) and the hierarchical pooling layer may be redundant for smaller graphs.

• The pre-training based approaches [17], e.g., GAT^*_{20}, suffer varying degrees of performance degradation (0.9-5.1% on top-10 rate) to the results reported in the original paper after we fix the settings to avoid information leakage and unfair comparisons with one-hot based approaches. Unlike the observations in [17], our results show that the pre-training approach may not improve the model performance in most cases. A possible reason is that the two stage pre-training approach is unstable.

• As mentioned in [17], since there may be different target components for the same input, the upper bound of a model cannot reach 100% at the top-k rate. However, our customized approach is closer to the upper bound compared to other baselines.

4.3 Case Study

To further illustrate the efficacy of CusGNN in real-world scenarios, we conduct a visualization study on four distinct partial assemblies in our internal scenarios with respect to the searched GNN architectures. The initial two assemblies revolve around a car assembly. In Figure 4(a), during the early stages of the design process, the ground truth for the next required components car windows which



(a) Catalog A (b) Catalog B (c) Catalog C

Figure 5: Searched GNN architectures for different catalogs from different manufacturers. From this figure, we can see that the searched architectures are different for different catalogs.

are recommended in the top-2 list. This dependency on the local information of the car body. However, as the design progresses (Figure 4(b)), the axle emerges as the subsequent component required. It is noteworthy that, at this point, the window components are already assembled, necessitating multiple layers of GNNs to capture multi-hop information. Similar phenomena are observed in the other two subfigures, which pertain to partial assemblies of a truck. In Figure 4(c), the next required component is identified as truck tires, with both wide and narrow truck tires exhibiting high prediction probabilities. However, in Figure 4(d), after the importation of all necessary narrow tires, the prediction probability for wide truck tires decreases. In summary, GNN models for the NCR task need a better ability to aggregate neighbor information and fusion multi-hop information to capture changes in the graph structure and our CusGNN has these abilities through searching for optimal structures in our well-designed search space, thereby enhancing prediction probabilities.

4.4 Model Analysis

In this section, we investigate the impact of different experimental settings on the on the model performance, including the impact of NAS block number, design dimensions in search space, and search algorithm.

4.4.1 Impact of NAS Block Number. We investigate the impact of different NAS Blocks on performance. The number of NAS blocks determines the maximum GNN layer, which, in turn, affects the receptive field of nodes in a graph. In other words, the number of GNN layer determine the receptive field of nodes in a graph. Based on the average graph diameter presented in Table 5, we heuristically set the number of NAS blocks from 1 to 8 for catalog B with an average graph diameter of 10.06, and from 1 to 4 for catalog A and C with average graph diameter 4.45 and 2.94 separately. As shown in Figure 6, the top-10 rate for catalogs A and C reaches its maximum when the number of blocks is 3 and slightly degrades when the block number is 4, which is close to the average graph diameter. On the other hand, the model performance tends to increase as the block number increases for catalog B, reaching its highest top-10 rate when the block number is 7 and stabilizing when the block number is 8.

4.4.2 Impact of Design Dimension. We conduct ablation experiments on the search space to evaluate the necessity of different component dimensions. We evaluate three design dimensions in our search space individually.

First, we fix the aggregation function and provide two variants CusGNN (GCN) and CusGNN (GAT). As shown in Table 2, CusGNN outperforms both variants, indicating the necessity of adding more expressive aggregation functions. Additionally, we observed that GCN is more suitable for simpler datasets (catalogs A and C), while GAT is more suitable for the more complex dataset (catalog B). This phenomenon may be explained by the fact that more complex datasets require more complex models. Therefore, different datasets from different manufacturers require different aggregation functions, further emphasizing the importance of customizing aggregation functions.

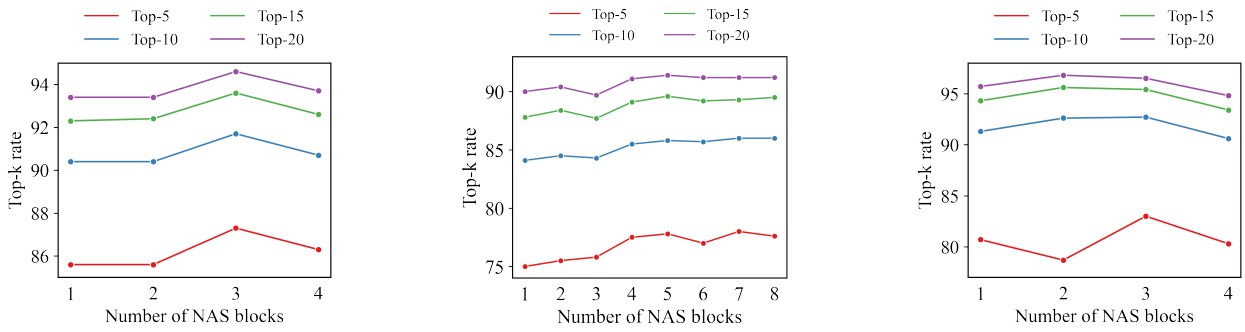
Second, we fix the fusion function and provide four variants: CusGNN (Mean), CusGNN (Max), CusGNN (Sum), and CusGNN (Concat). As shown in Table 3, among the four variants, CusGNN (Concat) performs well on catalog A while CusGNN (Max) performs poorly. However, when it comes to catalog C, the two variants have the opposite performance. In catalog B, most variants perform well except CusGNN (Max). These observations indicate that different datasets require different fusion functions. However, in most cases, CusGNN outperforms most variants, demonstrating the necessity to customize fusion function functions for different datasets from different manufacturers.

Third, from the searched architectures in Figure 5, we observe that the Max readout function performs well on all three catalogs. Thus, to explore whether the other readout operations could perform equally well, we fix the readout function and provide three variants: CusGNN (Mean), CusGNN (Sum), and CusGNN (Attention). As shown in Table 4, all three variants obtained varying degrees of performance degradation compared to CusGNN, which adopts the searched Max readout function. Therefore, the Max readout function may be more suitable for CAD assembly modeling tasks. Anyway, our customized approach could find the most appropriate readout function automatically.

Table 2: Ablation study with fixed aggregation functions.

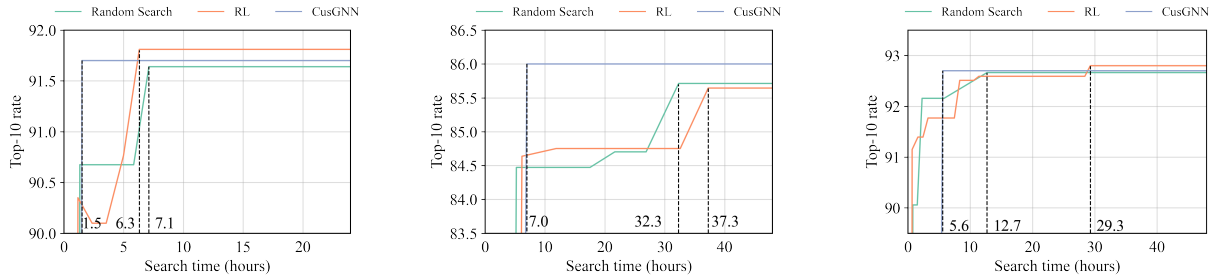
catalog	model \ k	1	2	3	5	10	15	20
A	CusGNN (GAT)	47.8%	71.9%	80.6%	85.8%	90.6%	92.5%	93.5%
	CusGNN (GCN)	47.6%	71.5%	80.4%	86.1%	90.9%	92.9%	93.9%
	CusGNN	**49.2 %**	**73.6 %**	**82.2 %**	**87.3 %**	**91.7%**	**93.6 %**	**94.6%**
B	CusGNN (GAT)	30.7%	51.8%	65.5%	77.9%	85.8%	89.0%	90.9%
	CusGNN (GCN)	30.9%	50.3%	63.0%	75.4%	84.4%	88.3%	90.2%
	CusGNN	**32.6%**	**53.3%**	**66.4%**	**78.0%**	**86.0%**	**89.3%**	**91.2%**
C	CusGNN (GAT)	28.0%	48.8%	64.1%	81.2%	91.4%	94.4%	95.7%
	CusGNN (GCN)	27.3%	48.0%	64.0%	81.5%	92.2%	95.2%	96.3%
	CusGNN	**28.7%**	**50.0%**	**65.9%**	**83.0%**	**92.7%**	**95.4%**	**96.5%**

4.4.3 Impact of Search Algorithm. To evaluate the effectiveness and efficiency of our differentiable search algorithms, we conduct experiments using two other search algorithms on our search space: Random and **R**einforcement **L**earning (RL), as discussed in C. For smaller dataset catalog A, We evaluate the performance of the models obtained through these algorithms on the three catalogs and report the top-10 rates in Figure 7. The results show that CusGNN only takes 1.5 hours (including one supernetwork training time plus subnetwork retraining time) to achieve comparable performance on



(a) Catalog A (b) Catalog B (c) Catalog C

Figure 6: Performance comparison for different numbers of NAS blocks.



(a) Catalog A (b) Catalog B (c) Catalog C

Figure 7: Search time for different search algorithms.

Table 3: Ablation study with fixed fusion functions.

catalog	model \ k	1	2	3	5	10	15	20
A	CusGNN (Mean)	47.8%	71.9%	80.6%	85.8%	90.6%	92.5%	93.5%
	CusGNN (Max)	47.4%	71.8%	80.5%	85.7%	90.3%	92.4%	93.6%
	CusGNN (Sum)	48.5%	72.3%	80.8%	86.2%	90.7%	92.7%	93.7%
	CusGNN (Concat)	49.0%	73.3%	82.1%	**87.4 %**	**91.8 %**	**93.6 %**	**94.6%**
	CusGNN	**49.2 %**	**73.6 %**	**82.2 %**	87.3%	91.7%	**93.6 %**	**94.6%**
B	CusGNN (Mean)	31.4%	51.5%	64.8%	76.9%	85.6%	**89.4%**	91.2%
	CusGNN (Max)	31.0%	50.9%	63.8%	75.9%	84.6%	88.3%	90.3%
	CusGNN (Sum)	31.5%	52.0%	65.4%	77.2%	85.7%	**89.4 %**	91.1%
	CusGNN (Concat)	31.4%	52.2%	65.5%	77.3%	85.9%	**89.4 %**	**91.4%**
	CusGNN	**32.6%**	**53.3%**	**66.4%**	**78.0%**	**86.0%**	89.3%	91.2%
C	CusGNN (Mean)	26.6%	46.9%	62.7%	81.0%	91.9%	94.7%	96.0%
	CusGNN (Max)	27.8%	49.0%	65.0%	82.5%	**92.7 %**	95.4%	96.4%
	CusGNN (Sum)	27.3%	48.2%	64.4%	82.4%	92.6%	**95.6 %**	**96.8%**
	CusGNN (Concat)	27.1%	47.4%	63.0%	80.2%	90.9%	94.1%	96.0%
	CusGNN	**28.7%**	**50.0%**	**65.9%**	**83.0%**	**92.7%**	95.4%	96.5%

Table 4: Ablation study with fixed readout functions.

catalog	model \ k	1	2	3	5	10	15	20
A	CusGNN (Mean)	47.5%	71.3%	79.7%	85.2%	90.1%	92.1%	93.3%
	CusGNN (Attention)	47.2%	71.0%	79.5%	85.0%	90.0%	92.0%	93.1%
	CusGNN (Sum)	47.9%	72.1%	80.6%	86.1%	90.7%	92.5%	93.6%
	CusGNN	**49.2 %**	**73.6 %**	**82.2 %**	**87.3 %**	**91.7%**	**93.6 %**	**94.6%**
B	CusGNN (Mean)	30.4%	49.7%	62.6%	75.2%	84.5%	88.3%	90.4%
	CusGNN (Sum)	30.6%	50.5%	63.0%	74.9%	84.0%	88.0%	90.1%
	CusGNN (Attention)	30.1%	49.1%	61.9%	74.9%	84.3%	88.2%	90.3%
	CusGNN	**32.6%**	**53.3%**	**66.4%**	**78.0%**	**86.0%**	**89.3%**	**91.2%**
C	CusGNN (Mean)	27.7%	48.3%	63.7%	80.8%	91.4%	94.3%	95.6%
	CusGNN (Sum)	26.9%	47.5%	63.3%	81.1%	91.5%	94.3%	95.6%
	CusGNN (Attention)	27.5%	48.0%	63.4%	80.4%	91.2%	94.4%	95.8%
	CusGNN	**28.7%**	**50.0%**	**65.9%**	**83.0%**	**92.7%**	**95.4%**	**96.5%**

catalog A, while RL and Random take 6.3 and 7.1 hours, respectively. Moreover, when the search overhead reaches 24 hours, there are no further improvements in performance by these two algorithms, indicating that the performance of 91.8% top-10 rate may be close to the global optimum on catalog A, and CusGNN is able to approach it in just 1.5 hours. However, the final performance of CusGNN was slightly lower than the RL baseline on catalog A, which could be attributed to the fact that differentiable algorithms are sometimes not robust enough [4, 35, 49]. Meanwhile, on the larger datasets like catalog B, which requires more time to train a single model, RL and Random were not able to comparable performance to CusGNN (7.0 hours) within 48 hours. On catalog C, CusGNN takes 5.6 hours to reach the comparable performance of RL and Random which take 12.7 and 29.3 hours respectively. The final performance of CusGNN was slightly lower than the RL baseline on catalog C for the same reason on catalog A.

5 CONCLUSION

In this work, we propose a novel NAS framework to customize GNN-based recommender systems for different datasets from different manufacturers. To achieve this, we design a search space with three dimensions and adopt an effective differentiable search algorithm to automatically search for a data-specific GNN architecture. Experiments demonstrate that our approach achieves state-of-the-art performance on three datasets from three manufacturers. In the future, we plan to collect more complex industrial datasets and explore how our approach can benefit in more complex scenarios.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (No. U20B2045, 62192784, U22B2038, 62002029, 62172052). We thank all anonymous reviewers for their constructive comments.

REFERENCES

[1] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* (2012).

[2] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *ICLR*.

[3] David Buterez, Jon Paul Janet, Steven J Kiddle, Dino Oglic, and Pietro Liò. 2022. Graph Neural Networks with Adaptive Readouts. In *NeurIPS*. 19746–19758.

[4] Xiangning Chen and Cho-Jui Hsieh. 2020. Stabilizing differentiable architecture search via perturbation-based regularization. In *ICML*. 1554–1565.

[5] Xuanyi Dong and Yi Yang. 2019. Searching for a robust neural architecture in four gpu hours. In *CVPR*. 1761–1770.

[6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20 (2019), 1997–2017.

[7] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. 2021. Computer-aided design as language. In *NeurIPS*. 5885–5897.

[8] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. 2021. Graph neural architecture search. In *IJCAI*. 1403–1409.

[9] Xiaojie Guo and Liang Zhao. 2022. A systematic survey on deep generative models for graph generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 5 (2022), 5370–5390.

[10] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*. 1024–1034.

[11] ZHAO Huan, YAO Quanming, and TU Weiwei. 2021. Search to aggregate neighborhood for graph neural network. In *ICDE*. 552–563.

[12] AJAY KUMAR JAISWAL, Peihao Wang, Tianlong Chen, Justin F Rousseau, Ying Ding, and Zhangyang Wang. 2022. Old can be Gold: Better Gradient Flow can Make Vanilla-GCNs Great Again. In *NeurIPS*.

[13] Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical Reparameterization with Gumbel-Softmax. In *ICLR*.

[14] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G Kim, and Adriana Schulz. 2021. Automate: A dataset and learning approach for automatic mating of cad assemblies. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–18.

[15] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[16] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[17] Carola Lenzen, Alexander Schiendorfer, and Wolfgang Reif. 2022. A recommendation system for CAD assembly modeling based on graph neural networks. In *ECML-PKDD*.

[18] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. 2019. Deepgcns: Can gcns go as deep as cnns?. In *CVPR*. 9267–9276.

[19] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*.

[20] Wei Li, Justin Matejka, Tovi Grossman, Joseph A Konstan, and George Fitzmaurice. 2011. Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction (TOCHI)* (2011), 1–35.

[21] Yaoman Li and Irwin King. 2020. Autograph: Automated graph neural network. In *ICONIP*. 189–201.

[22] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *ICLR*.

[23] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324* (2018).

[24] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. Darts: Differentiable architecture search. In *ICLR*.

[25] Katia Lupinetti, Jean-Philippe Pernot, Marina Monti, and Franca Giannini. 2019. Content-based CAD assembly model retrieval: Survey and future challenges. *Computer-Aided Design* (2019), 62–81.

[26] Weijian Ma, Minyang Xu, Xueyang Li, and Xiangdong Zhou. 2023. MultiCAD: Contrastive Representation Learning for Multi-modal 3D Computer-Aided Design Models. In *CIKM*. 1766–1776.

[27] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.

[28] M. E. J. Newman. 2003. Mixing patterns in networks. *Phys. Rev. E* (2003), 026126.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*. 8024–8035.

[30] Yuhan Quan, Huan Zhao, Jinfeng Yi, and Yuqiang Chen. 2024. Self-supervised Graph Neural Network for Mechanical CAD Retrieval. arXiv:2406.08863

[31] MMM Sarcar, K Mallikarjuna Rao, and K Lalit Narayan. 2008. *Computer aided design and manufacturing*. PHI Learning Pvt. Ltd.

[32] David W. Scott. 1992. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley.

[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *ICLR*.

[34] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop*.

[35] Ruochen Wang, Minhao Cheng, Xiangning Chen, Xiaocheng Tang, and Cho-Jui Hsieh. 2021. Rethinking Architecture Selection in Differentiable NAS. In *ICLR*.

[36] Zhenyi Wang, Huan Zhao, and Chuan Shi. 2022. Profiling the Design Space for Graph Neural Networks based Collaborative Filtering. In *WSDM*. 1109–1119.

[37] Lanning Wei, Zhiqiang He, Huan Zhao, and Quanming Yao. 2023. Search to capture long-range dependency with stacking gnns for graph classification. In *TheWebConf*. 588–598.

[38] Lanning Wei, Huan Zhao, and Zhiqiang He. 2022. Designing the topology of graph neural networks: A novel feature fusion perspective. In *TheWebConf*. 1381–1391.

[39] Lanning Wei, Huan Zhao, Zhiqiang He, and Quanming Yao. 2023. Neural architecture search for GNN-based graph classification. *ACM Transactions on Information Systems* 42, 1 (2023), 1–29.

[40] Lanning Wei, Huan Zhao, Quanming Yao, and Zhiqiang He. 2021. Pooling architecture search for graph classification. In *CIKM*. 2091–2100.

[41] Karl DD Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G Lambourne, Armando Solar-Lezama, et al. 2022. Joinable: Learning bottom-up assembly of parametric cad joints. In *CVPR*. 15849–15860.

[42] Karl DD Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2021. JoinABLe: Learning Bottom-up Assembly of Parametric CAD Joints. *arXiv preprint arXiv:2111.12772* (2021).

[43] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation* (1997), 67–82.

[44] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2018. SNAS: stochastic neural architecture search. In *ICLR*.

[45] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *ICLR*.

[46] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *ICML*. 5453–5462.

[47] Jiaxuan You, Zhitao Ying, and Jure Leskovec. 2020. Design space for graph neural networks. In *NeurIPS*. 17009–17021.

[48] Kaicheng Yu, Christian Suito, Martin Jaggi, Claudiu-Cristian Musat, and Mathieu Salzmann. 2020. Evaluating the search phase of neural architecture search. In *ICLR*.

[49] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. 2020. Understanding and Robustifying Differentiable Architecture Search. In *ICLR*.

[50] Guanqi Zhan, Qingnan Fan, Kaichun Mo, Lin Shao, Baoquan Chen, Leonidas J Guibas, Hao Dong, et al. 2020. Generative 3d part assembly via dynamic graph learning. In *NeurIPS*. 6315–6326.

[51] Tianyu Zhao, Cheng Yang, Yibo Li, Quan Gan, Zhenyi Wang, Fengqi Liang, Huan Zhao, Yingxia Shao, Xiao Wang, and Chuan Shi. 2022. Space4hgnn: a novel, modularized and reproducible platform to evaluate heterogeneous graph neural network. In *SIGIR*. 2776–2789.

[52] Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. In *ICLR*.

[53] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*. 8697–8710.

Table 5: Statistical information about CAD datasets.

catalog	# graphs	# comp	node degrees	# nodes	# edges	graph diameter	# partial graphs (train/val/test)
A	11,826	1,930	$1-9;\ \emptyset\ 1.7$	$4-33;\ \emptyset\ 6.1$	$3-32;\ \emptyset\ 5.1$	$2-32;\ \emptyset\ 4.45$	133,271/42,531/43.420
B	11,895	3,099	$1-13;\ \emptyset\ 1.9$	$4-69;\ \emptyset\ 18.2$	$3-68;\ \emptyset\ 17.2$	$2-38;\ \emptyset\ 10.06$	1,557,498/514,107/484,009
C	11,943	1,924	$1-16;\ \emptyset\ 1.7$	$4-20;\ \emptyset\ 6.7$	$3-19;\ \emptyset\ 5.7$	$2-6;\ \emptyset\ 2.94$	1,081,453/337,200/379,344

A DATASET AND METRIC

For the dataset, we follow the original division with 6:2:2 train/validation/test assemblies in the previous study [17]. The assemblies were then transformed into partial graphs and corresponding labels using the raw cut-off method in [17]. The cut-off process begins with a complete assembly and cut off non-cohesive nodes. A node is non-cohesive if its removal does not cause a graph composed of multiple connected components. The partial graphs together with the corresponding cut-off components form the data instances stored in a multiset D. They keep repeating this process until the remaining graph contains a minimum number of nodes. Finally, they remove duplicate instances from D. The assemblies and processed partial graph datasets are available at their open-source link[2].

For the metric, we follow the top-k rate in [17], which refers to the percentage of the true component ID (i.e., the label) being present in the top-k predictions of a model. For the current partial assembly, a higher topk rate proves that the next component to be used is better able to be included in the recommended list of length k. Thus, if the topk rate is higher, it proves that the model is better. We evaluate all methods at metric when k=1, 2, 3, 5, 10, 15, 20. As mentioned in [17], k=10 is much more important as this number of recommendations can be well integrated into a CAD system and provide a diverse range of component options for designers.

B FIXING OF UNREASONABLE SETTINGS

For the pre-training based methods: $\mathbf{GCN}_{20}$, $\mathbf{GCN}_{100}$, $\mathbf{GAT}_{20}$, and $\mathbf{GAT}_{100}$. The numbers 20 and 100 denote the dimensions of the pre-trained component type embeddings. However, we find some unreasonable settings in their source code[3]. As shown in Fig. 8(a), they pre-trained their embeddings table on both the train set and the valid set and did hyperparameter tuning on the test set which is information leakage. As shown in Fig. 8(b), we fix the settings by removing the valid set from the training process and using it to verify the performance of embeddings rather than the test set. The fixed models are denoted with a star *.

C IMPLEMENTATION DETAILS

We show the specific experimental setup as follows:

- For $\mathrm{GCN}_{one-hot}$, $\mathrm{GCN}_{one-hot}$, FBase and Evergreen baselines[4], we report the result in original paper [17].
- For GAT^*_{100}, GAT^*_{20}, GCN^*_{20} and GCN^*_{100}, we follow the best searched hyperparameters [5] in original paper [17] and report the result after fix the unreasonable settings.

[2]https://figshare.com/articles/dataset/ECML22_GRAPE_Data/20239767

[3]https://github.com/isse-augsburg/ecml22-grape/blob/main/scripts/train_embedding.py

[4]https://github.com/isse-augsburg/ecml22-grape/tree/main

[5]https://github.com/isse-augsburg/ecml22-grape/blob/main/models/component_prediction/model_configuration.py

- For CusGNN, we set the search epochs to 100 for catalog A and 20 for catalogs B and C (the train samples for B and C are nearly 10 times larger than A). For all datasets, we set the softmax temperature λ to 0.001 as suggested in [38] and batch size 8192 in the searching phase. We utilize Adam [15] optimizer with 0.01 and 0.0001 learning rates to optimize operation selection weight c and operation weight respectively. After the search process, we obtained one candidate GNN. We retrained the searched GNN for 100 epochs for catalog A and 20 epochs for catalog B and C with a batch size of 1024, hidden size of 2048 for catalogs A and C, and 3072 for catalog B. We used a learning rate of 0.0001 and weight decay of 0.0001 for Adam optimizer in the same seed 0 setting in [17] for a fair comparison. We save the best valid loss epoch model and used it to report the final test performance.
- For PAS [40], we use their original implementation [6] and the input is consistent with CusGNN. Hyperparameters in the search and retrain process are consistent with CusGNN for a fair comparison.
- For experiments of different NAS blocks for CusGNN, we use the different number of NAS blocks and keep the other settings consistent.
- For ablation experiments, we fixing the type of aggregation functions, fusion functions, or readout functions separately and the rest of the settings are consistent with CusGNN.
- For baselines for different search algorithms, we adopt Reinforcement Learning (RL) based search algorithm [52, 53] and Random Search [1], which is considered to be a strong NAS baseline in [48]. For RL-based baseline, we follow the implementation in [8] and replace the search space with our approach. Due to time constraints, we limited the search time to 24 hours for both search algorithms on catalog A and 48 hours for catalog B and C.

We implement all the models with PyTorch [29] and the GNN operations by the popular GNN library: Deep Graph Library (DGL) [34]. We run all experiments utilizing Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and GeForce RTX 3090 as the experimental environment.

[6]https://github.com/LARS-research/PAS

```
41          logger.info('Creating new grams')
42          # Use training and validation data from challenge for training; test data for validation
43          train_graphs = file_handler.deserialize(f'{data_path}graphs_train.dat')
44                                                                  Adding the valid set to train embeddings
45          # add graphs from train and validation data from challenge for training
46          train_graphs = list(itertools.chain(train_graphs, file_handler.deserialize(f'{data_path}graphs_val.dat')))
47
48          # same for validation data: use test data from challenge for validation
49          val_graphs = file_handler.deserialize(f'{data_path}graphs_test.dat')  ⟶  Test data leakage
50          logger.info('Graph files loaded.')
```

(a) Raw Code

```
logger.info('Creating new grams')
# Use training data from challenge for training; val data for validation
train_graphs = file_handler.deserialize(f'{data_path}graphs_train.dat')

# same for validation data: use val data from challenge for validation
val_graphs = file_handler.deserialize(f'{data_path}graphs_val.dat')
logger.info('Graph files loaded.')
```

(b) Fixed Code

Figure 8: Code comparison between the raw code and the fixed code. The raw code adds the valid set to train embeddings and uses the test set to select hyperparameters which is an information leakage. We fix these unreasonable settings in the fixed code.

Figure 9: This image is from [42] and we could observe that different assemblies have different shapes.