A Recommendation System for CAD Assembly Modeling based on Graph Neural Networks[⋆]

Carola Gajek (✉)[1][0000−0002−1678−6795], Alexander Schiendorfer[2][0000−0002−5283−5304], and Wolfgang Reif[1][0000−0002−4086−0043]

[1] Institute for Software & Systems Engineering, University of Augsburg, Augsburg, Germany `{gajek,reif}@isse.de`

[2] Institute AImotion Bavaria, Technische Hochschule Ingolstadt, Ingolstadt, Germany `Alexander.Schiendorfer@thi.de`

Abstract. In computer-aided design (CAD), software tools support design engineers during the modeling of assemblies, i.e., products that consist of multiple components. Selecting the right components is a cumbersome task for design engineers as they have to pick from a large number of possibilities. Therefore, we propose to analyze a data set of past assemblies composed of components from the same component catalog, represented as connected, undirected graphs of components, in order to suggest the next needed component. In terms of graph machine learning, we formulate this as graph classification problem where each class corresponds to a component ID from a catalog and the models are trained to predict the next required component. In addition to pretraining of component embeddings, we recursively decompose the graphs to obtain data instances in a self-supervised fashion without imposing any node insertion order. Our results indicate that models based on graph convolution networks and graph attention networks achieve high predictive performance, reducing the cognitive load of choosing among 2,000 and 3,000 components by recommending the ten most likely components with 82–92% accuracy, depending on the chosen catalog.

Keywords: Graph Machine Learning · Recommendation · Computer-Aided Design · AI-Aided Design.

1 Recommending Components in Assembly Modeling

Computer-aided design (CAD) most generally refers to using computers to support the creation, modification, analysis, or optimization of three-dimensional mechanical designs [15]. More specifically, we consider the problem of recommending existing CAD *components* to design engineers during assembly modeling, i.e., the design of new assemblies composed of those components. Consider,

⋆ Research leading to this paper was funded by the Bavarian State Ministry for Economic Affairs, Regional Development and Energy (StMWi), Project KOGNIA, in cooperation with Cadenas GmbH.

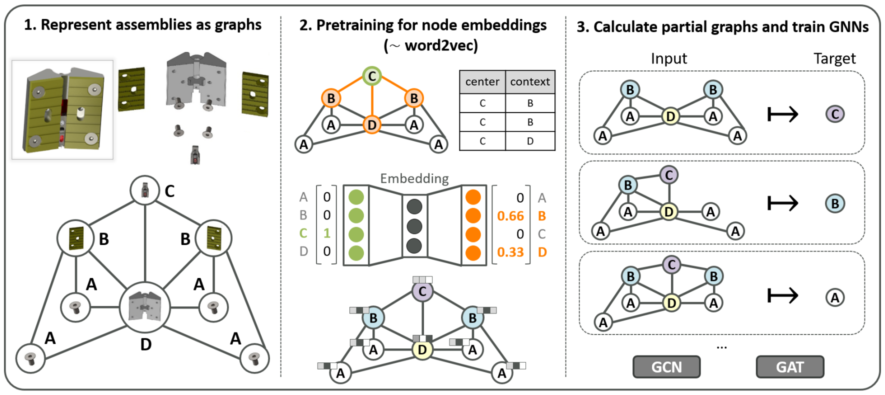

1. Represent assemblies as graphs
2. Pretraining for node embeddings (~ word2vec)
3. Calculate partial graphs and train GNNs
Input
Target
Embedding
center
context
GCN
GAT

Fig. 1: Our overall approach: CAD assemblies are converted to graphs which are enriched by component representations obtained from a variation of word2vec [14]. Data instances for component recommendation are derived from CAD assemblies to train GNNs to predict the next required components during construction. Best viewed in color on screen.

as a simplified example, a cabinet that consists of five plates, a hinge, a handle, screws, etc. A governing assumption of our approach is that components that are *used together frequently* have a causal relationship that is captured in the data. For example, a heavy hinge might often be combined with a heavy door and similarly for lighter components. Such information could lead to *recommendations* regarding the next component to be inserted. Rather than having design engineers (i.e., CAD system users) manually maintain such logical and domain-specific rules (a tedious task that is likely to be neglected in practice), we strive for a data-driven solution that adapts to the sets of assemblies on which it is trained – even if that entails imperfect recommendations in some cases. That way, experienced designers' knowledge could be extracted from existing assemblies and speed up the design process.

Unfortunately, assumptions made for standard recommendation systems (collaborative or content-based filtering) do not hold in this use case. First, personalized information about the design engineers (e.g., their typical area of work, their recent designs, or even their current intention) that would be necessary for collaborative filtering is typically not available. Second, what constitutes a good recommendation for the next component depends on the intended design as opposed to simply going for a rather static "liking" of elements.

Therefore, we treat mechanical designs as undirected graphs, where nodes correspond to CAD components and edges denote connections between them. Edges can result from so-called "mating" conditions[3] in CAD systems or can be read off by geometric proximity. Even though a graph can be thought of as being generated by a sequence of node and edge insertions (and therefore be

[3] Mating conditions define relative positions of components to each other.

amenable to recurrent models), such sequential information is not stored in CAD designs. Moreover, an assembly can emerge in any order – also depending on the designer's preferences – which is why we prefer a model that is invariant to permutations. Therefore, we propose the following approach, as shown in Figure 1:

1. Extract only components and connections as connected, undirected graphs from CAD assembly models – ignoring other metadata
2. Pretrain to get low-dimensional component embeddings using a technique based on word2vec [14], choosing an appropriate context size instead of random walks like in node2vec [5]
3. Generate data instances for graph neural networks (GNN) by means of "cutting off" nodes in a self-supervised fashion, resulting in pairs consisting of partial graph and the cut-off node
4. Train GNNs to predict the next component given a partial graph, i.e., learn a discriminative model $p(\mathit{next\_node} \mid \mathit{partial\_graph})$ that is part of an autoregessive model which, when unrolled, leads to a generative process [9]
5. Evaluate the performance of GNNs by the top-10 rate on an unseen test set

Our approach contributes to the ongoing trend of extending CAx (computer-aided processes like design, engineering and manufacturing) to AIAx (artificial intelligence-aided processes) [3,7,20,24]. In particular, it falls into the realm of AI-aided design that supports design engineers in a data-driven fashion. Whereas most existing AIAD approaches are concerned with adequately modeling the 3D geometry [3,24], our model is only concerned with components and connections found in assemblies and deriving useful recommendations only from their usage patterns in shared component catalogs.

Related Work The cognitive load imposed onto design engineers working with a complex CAD system has already been recognized [11]. The authors apply collaborative filtering to suggest useful software commands to improve the workflow. Our approach is based on the same motivation but focuses on components that are likely to be inserted into the current assembly instead of commands. Applying principles of search engines and information retrieval to support CAD design engineers is called assembly retrieval [13]. There, different notions of similarity (usage similarity, component overlaps, etc.) are considered to enable queries for similar assembly designs, instead of suggesting how to extend an assembly like in our approach. Recently, transformer-based generative models have been applied to CAD models by generating sequences of CAD-typical geometrical operations such as "sketching", "extruding", "boolean subtracting" [17]. While this method addresses component design tasks, our method focuses on usage similarity of components reoccurring in assemblies, abstracts away from geometrical features, and – most notably – does not require a linear order of operations.

Deep learning on non-Euclidean domains such as graphs or manifolds is often referred to as *geometric deep learning* [2]. This is not limited to the geometry in the sense of a 3D (mesh) model but stems from the fact that certain equivariances or invariances are derived from the symmetry properties prevalent in the data –

such as permutation equivariance for the neighboring nodes in a graph. In that sense, our approach belongs to geometric deep learning: a component graph that we extract from a CAD model should lead to the same hidden representation, regardless of how the components are (arbitrarily) ordered, a property that is found in graph neural networks as well as in deep sets [22].

The applicability of graph neural networks to recommendation systems has also been recognized [18], essentially since the domain of many recommendation tasks (e.g., movies to users, products to customers, etc.) can be formalized as a (hyper)graph consisting of all users and items, connecting users with their liked or purchased items. However, the goal here is to predict new connections, i.e., to recommend items to users, rather than adding new nodes to the graph.

The task of predicting the next component for an assembly can be seen as a generative model for graphs, if rolled out step by step. Existing approaches to generative deep graph models, however, tend to learn the probability distribution of the observed graphs at once, called "one-shot generating" in [6], using variational autoencoders, generative adversarial networks, or normalizing flows. The authors of [12] propose to learn a sequence of node and edge insertions (called structure building actions). Their approach, in particular the neural network architecture that they use to map an intermediate graph to the next insertion action, bears similarities in terms of the architecture we use in our approach. However, the edges can appear in any order which may lead to graphs that are not connected – which we want to explicitly exclude. Our focus on individual component insertion steps also removes the need for recurrent structures that [12] employs. Moreover, training general generative graph models is reported to be more difficult to balance which is why we focus on the discriminative task of predicting $p(\mathit{next\_node} \mid \mathit{partial\_graph})$ but manage the generation of appropriate data via self-supervision outside the training loop (cf. Figure 1).

2 Graph Neural Networks

In our approach, we use undirected graphs to represent assemblies. A graph $G = (\mathcal{N}, \mathcal{E})$ consists of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ where an edge $e = (i, j) \in \mathcal{E}$ denotes that nodes i and j are connected. For undirected graphs, all edges are bidirectional, i.e., $(i, j) \in \mathcal{E}$ implies $(j, i) \in \mathcal{E}$. The neighbors $N(i) \subseteq \mathcal{N}$ of a node i are given by the set of nodes adjacent to it, i.e., $N(i) = \{j \in \mathcal{N} \mid (i, j) \in \mathcal{E}\}$. A node i is called a *leaf node* if it has exactly one neighbor, i.e., if $|N(i)| = 1$. We define the removal $G \setminus \{i\}$ of a node i as the graph that results from removing i from $\mathcal{N}$ and, consequently, removing all edges involving i from $\mathcal{E}$. We call a node *cohesive* if its removal would result in a graph composed of multiple connected components.

Graph neural networks (GNN) operate on graphs to perform tasks such as graph classification, node classification, or link prediction [19]. Each node $i \in \mathcal{N}$ is described by a feature vector $\mathbf{x}_i$ of dimension d (e.g., one-hot encodings of numeric indices, pretrained embeddings, or other features describing i). Stacking the feature vectors of all nodes in a graph gives rise to an input matrix $\mathbf{X}$ with dimensions $|\mathcal{N}| \times d$. The edges $\mathcal{E}$ are represented by an $|\mathcal{N}| \times |\mathcal{N}|$-adjacency matrix $\mathbf{A}$ where $\mathbf{A}_{i,j} = 1 \Leftrightarrow (i,j) \in \mathcal{E}$. In addition, features can be assigned to edges, for example, to describe different relations between users of a social network.

The representation of $\mathcal{N}$ and $\mathcal{E}$ as matrices $\mathbf{X}$ and $\mathbf{A}$ is useful due to the efficient interoperability with existing deep learning frameworks but it enforces the set $\mathcal{N}$ to be ordered. Therefore, graph neural networks require functions f that are applied to any matrix representation of the same graph to be either *permutation equivariant* or *permutation invariant*. Equivariance is used for node classification and states that permuting the output of f is the same as applying f to the permuted input, i.e., $f(\mathbf{PX}) = \mathbf{P}f(\mathbf{X})$ for a permutation matrix $\mathbf{P}$. In contrast, invariance is used for graph classification and describes that permuting the input does not affect the output, i.e., $f(\mathbf{PX}) = f(\mathbf{X})$. In our use case, permutation invariance is crucial as the CAD components have no inherent order and their insertion sequence is not given. In a GNN layer, the node representations $\mathbf{h}_i$ are updated by a function over their neighbors' features. From the perspective of a node i, the update in layer $l+1$ (called message-passing) is given by

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in N(i)} \psi\left(\mathbf{h}_j^{(l)}\right)\right) \tag{1}$$

where ϕ and ψ are parameterized (i.e., trainable) mappings, e.g., fully connected layers followed by a nonlinearity [19]. Permutation equivariance is obtained by having ψ depend only on a single node, i.e., it is applied node-wise, and permutation invariance results from using a commutative and associative operation $\bigoplus$ such as the sum, the average, or the maximum. Since the features of each node are processed with the same transformation ψ (parameter sharing) and the aggregation can be performed with any number of elements, GNNs can handle graphs with different structures and even different sizes.

Stacking multiple GNN layers enables propagating node features to more distant nodes. Thus, a single node can contain information of all adjacent nodes within a range of l hops after l layers. In addition to message-passing layers, so-called *readout layers* combine node representations resulting from a message-passing layer into a representation on graph-level. Hence, with a subsequent fully connected layer and Softmax activation, a graph can be classified. The literature provides different forms of GNN layers, which can be divided into three flavors: convolutional, attentional, and generic message-passing [2]. The latter is a generalization of the first two, where edge features are incorporated in addition to node features. Since assemblies are represented by graphs with only one edge type and no additional edge features, we consider only the first two forms:

Graph Convolutional Layer (GCN) The node-wise update rule of a graph convolutional layer [10] is given by

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{j \in N(i)} \frac{1}{c_{ij}} \mathbf{W}^{(l)} \mathbf{h}_j^{(l)}\right) \tag{2}$$

where $\sigma(\cdot)$ denotes a nonlinear activation function such as ReLU and $\mathbf{W}^{(l)}$ is the weight matrix of the fully connected layer transforming the node representations. The normalization constant $c_{ij} = \sqrt{|N(i)|}\sqrt{|N(j)|}$ is derived from the nodes' degrees only and thus heavily depends on the structure of the graph. Consequently, all neighbor nodes are weighted equally for aggregation. It can be interpreted as the importance of node j to the representation of i.

Graph Attention Layer (GAT) In our use case, some components of the current assembly may be more important for recommending next components than others. For this reason, we also investigate GATs [16] that individually weight neighbor nodes according to an attention mechanism:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{j\in N(i)} \alpha_{ij}^{(l)} \mathbf{W}^{(l)} \mathbf{h}_j^{(l)}\right) \tag{3}$$

The attention weights $\alpha_{ij}^{(l)}$ result from a learned function over the node features of i and $N(i)$ and serve as coefficients of an convex combination of transformed node features. Thus, graphs of the same structure but with different node features will typically lead to different attention weights. As with other attention-based architectures, multiple independent attention mechanisms can be used to extend the learning process (multi-head attention).

3 Graph-Based Recommendations for Assemblies using Pretrained Embeddings

In our proposed approach (depicted in Figure 1), we represent CAD assemblies as undirected graphs with component embeddings as node features and train GNNs for recommending the next required components during construction. We model this learning problem as a *graph classification problem* where the classes correspond to component IDs – one graph is mapped to the ID of the next needed component. By using Softmax as activation function in a final fully connected layer, we get normalized scores over all components that can be interpreted as a ranking of the components' recommendations.

Since embeddings are used to represent in the input graph's components, one may intuitively be inclined to use the same representation in the output, i.e., to predict component embeddings. However, this modeling has some disadvantages:

a) For component recommendations, the predicted embedding must always be assigned to a component. It may happen that there is no corresponding component in the proximity of the prediction in the embedding space because no component satisfies the desired properties – which component should be taken in this case?
b) The number of desired recommendations determines the architecture of the model ($k \cdot$ *embedding_size* output neurons for k recommendations), so if more

recommendations are desired, the model must be re-built and re-trained. In our modeling, we can simply determine the desired number of most likely elements from the distribution.

The GNNs presented in this paper do not employ edge features due to the fact that the given assemblies only specify which components are connected but do not describe the nature of this connection in more detail. Furthermore, we assume all possible component IDs to be included in the training data - we do not yet consider new or updated components in this paper.

3.1 Pretraining of Component Embeddings (comp2vec)

In some machine learning domains such as natural language processing (NLP), representing discrete objects as continuous vectors, so-called embeddings, instead of one-hot vectors has proven to be advantageous [14]. Since assembly modeling has many parallels to NLP, we investigate whether using component embeddings as node features contributes to the performance of our recommendation models: Just as documents are composed of words or letters, assemblies consist of components. Moreover, neighboring words (predecessor and successor) correspond to the possibly larger set of adjacent nodes in a graph (see Section 1 for representing assemblies as graphs). Due to the heterogeneity of available attributes of components in catalogs, our approach uses the only information that is always available: a unique component identifier. However, the approach is sufficiently generic in that the resulting node features can be embedding vectors combined with additional, problem-specific features. This also applies to the inclusion of the 3D geometry of components, which we decided to exclude in our approach.

word2vec[14] is a popular method for word embeddings, since only plain text without annotations is needed for training. The basic idea is that the meaning of a word is defined by its context words. Transferring this to assemblies, connected components define the purpose of a component – this information stored in embeddings could be very helpful for component recommendations. node2vec[5] is a modification of word2vec for network graphs based on the homophily assumption, i.e., that connected components are similar, such as friends in a social network. However, this assumption is clearly violated in our case: Two components serving *different purposes* are most likely connected in mechanical designs. Therefore, we generalized word2vec from sequences to graphs with a small modification in creating instances, and refer to the resulting model as *comp2vec* in the following. Word2vec (in the Skip-gram variant) trains a mapping from words to their context words within a defined context window. Its architecture is similar to an autoencoder with two layers, encoding each word in a small internal vector representation. The last layer is only used in training and truncated for inference as the embeddings are calculated in the hidden layer – whose dimension (a hyperparameter) determines the compression of the representation. The equivalent to n-distant words of a center word are n-hop distant nodes of a center node in a graph. For a graph, this approach may result in more generated instances. However, the training process itself remains the same as in word2vec.

Because representation learning is a subdomain of unsupervised learning, the difficulty is that the model cannot be evaluated directly due to the lack of labels. In this case, a downstream model (extrinsic evaluation) or domain knowledge (intrinsic evaluation) is used instead. In NLP, for example, pairs of synonym terms [1] are used for that purpose. This is, in principle, transferable to CAD assembly modeling, but requires high manual effort from design experts, since such intrinsic evaluations have to be created individually for each component catalog. Fortunately, in word2vec the embeddings are learned by solving a supervised task (predicting context words from a word), which allows to directly evaluate by loss. We assume that the unsupervised task is well solved if the corresponding supervised task is well solved, which is why we optimize for loss. As a consequence, different embedding sizes can be compared, since the output dimension is defined by the task and thus does not depend on hyperparameters.

3.2 Generating Data Instances for Component Recommendation

For the component recommendation task, we need the intermediate states of the assemblies during construction, hereafter referred to as *partial assembly* or *partial graph*, as well as the components that were directly assembled to them. As we do not know the original sequence of node and edge insertions that led to the final design and, even if we did, the order can still depend on the designer's preferences, we create instances for *every* possible creation sequence in a self-supervised fashion by Algorithm 1: Starting from a complete assembly (graph G), we iteratively cut off non-cohesivenodes until the remaining graph contains a minimum number of nodes *min_nodes*. The partial graphs together with the corresponding cut-off components form the data instances stored in the multiset D. Finally, in order to process them with GNNs, component embeddings are used as node features of the graphs and the target component is replaced by its ID.

Algorithm 1 Decomposition of Graphs into Instances

```
1: procedure DecomposeGraph($G = (\mathcal{N}, \mathcal{E}), D$)
2:     if $|\mathcal{N}| > min\_nodes$ then            ▷ domain-specific hyperparameter
3:         for every non-cohesivenode $n \in \mathcal{N}$ do            ▷ see Section 2
4:             add $(G \setminus \{n\}, n)$ to $D$            ▷ see Section 2
5:             DecomposeGraph($G \setminus \{n\}, D$)
6:     remove duplicate instances from $D$
```

Regarding complexity, the decomposition produces $O(|\mathcal{N}|!)$ graph-node instances which is prohibitive for large assemblies. The designs we considered in our experiments (composed of up to 70 components, cf. Table 1) were highly sequential, i.e. each individual (partial) assembly contained sufficiently few leaf nodes, so that this was not an issue. To scale up the approach, we would expect a sampling-based approach that only performs a subset of the removals or a random walk-based approach like used for large network graphs [8] to work well. This remains to be tested in future work.

3.3 Frequency-Based Baseline Model

The task of component recommendation for assembly modeling bears some similarity to market basket analysis in that we want to recommend additional candidates for a given collection of elements in both cases. Most market basket analysis approaches search for frequent itemsets and corresponding associative rules via counting. However, elements in a market basket are not related per se, whereas components of an assembly are explicitly connected. To the best of our knowledge, no suitable established technique taking into account such relations (and thus, the graph structure) exists in literature.

Therefore, we developed an instance-based model inspired by market basket analysis as a baseline: It stores a relative frequency distribution over the assembled components for each partial assembly of the training set. During inference for a given query graph, it looks up which components were most frequently attached to it. If the graph has not been seen in the training data, it looks for previously seen subgraphs that together form the query graph. These subgraphs should be as large as possible to most widely cover the context of the assembly and consequently provide good component recommendations. To exclude redundant subgraphs, the model determines the *minimal* set of its largest seen subgraphs subsuming the query graph. Thereafter, the relative frequency distributions over components for the subgraphs are aggregated to an overall distribution by determining the component-wise maximum. Finally, the k most frequent components are identified from the distribution.

4 Experiments

To test the applicability of our approach, we performed separate experiments on each of three different catalogs corresponding to different manufacturers. Each catalog contains nearly 12,000 assemblies. The catalogs differ in the number of unique components as well as the designs in the number components per assembly and graph diameters, as shown in Table 1. The assemblies of each catalog were split 60:20:20 into training, validation, and test graphs and afterwards transformed into instances consisting of partial graphs and expected component ID (see Section 3.2). We set out to answer the following research questions:

1. Are GNNs better at predicting the next component needed than the frequency-based baseline model?
2. Does pretraining of embeddings using comp2vec increase accuracy, or are GNNs with one-hot encodings as node features just as accurate?
3. Are GATs or GCNs better suited for the task? By what margin?

Table 1: Key facts of the three used component catalogs. For each metric column, the front part indicates the range of values and the back part the corresponding average.

catalog	#graphs	#comp.	node degrees	#nodes	#edges	graph diameter
A	11,826	1,930	1 - 9; ∅ 1.7	4 - 33; ∅ 6.1	3 - 32; ∅ 5.1	2 - 32; ∅ 4.45
B	11,895	3,099	1 - 13; ∅ 1.9	4 - 69; ∅ 18.2	3 - 68; ∅ 17.2	2 - 38; ∅ 10.06
C	11,943	1,924	1 - 16; ∅ 1.7	4 - 20; ∅ 6.7	3 - 19; ∅ 5.7	2 - 6; ∅ 2.94

4.1 Experimental Setup

In a preliminary study, we investigated suitable embedding sizes for the three component catalogs. First, the assemblies were divided 80:20 into training and validation sets and afterwards transformed into instances (see Section 3.1). To rate the models, we computed the sum of training loss and gap between training and validation loss [4, p. 425]. In comparison, different embedding sizes behaved the same for each catalog: Sizes between 20 and 90 as well as from 100 upwards each led to the same error level, consequently, we chose the minimum per range (20 and 100, respectively) as fixed embedding sizes for each catalog. In order to investigate the influence of pretrained embeddings on the performance of the task, we additionally generated an one-hot encoding of the components.

Based on the preliminary study, we examined seven models per catalog: the frequency-based baseline model as well as the two flavors of GNN presented in Section 2, each based on the three types of component representation. Both models were implemented using the respective layers in DGL[4], the precise hyperparameters were determined using hyperparameter search and can be found in a publicly available repository[5].

4.2 How well can GNNs learn the task at hand?

The main evaluation metric is the top-k rate, referring to the percentage of the true component ID (i.e., the target) being in the top-k predictions of a model. We are especially interested in $k = 10$, as this number of recommendations can be well integrated into a CAD system and offers a wide choice of components for designers. Since we are dealing with a task and data that has been little researched, we would like to be able to better assess the results of the models by estimating their performance from above and below. If we take a closer look at Algorithm 1 for generating the recommendation instances (in particular, lines 3 and 4), it is apparent that there can be several target components for one partial graph (e.g., by removing two nodes in both orders). This implies that a model (without being an oracle) cannot reach 100% at the top-1 rate, and depending on the number of different targets for the same input, this may even

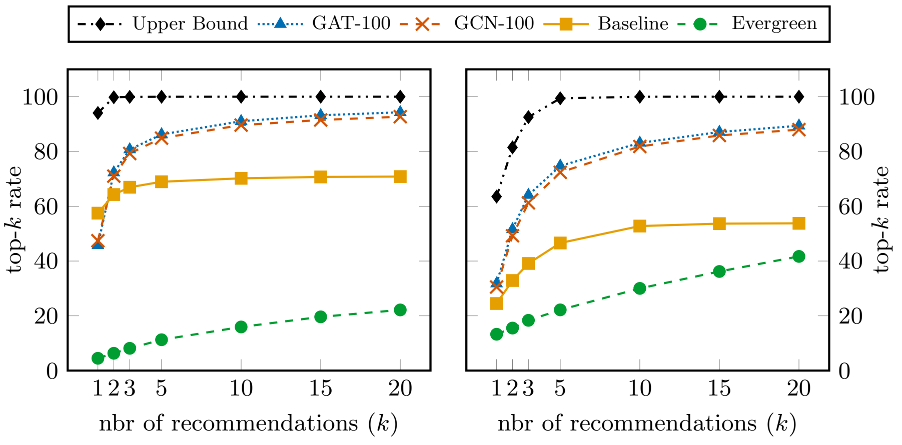
[4] Deep Graph Library https://www.dgl.ai/
[5] https://github.com/isse-augsburg/ecml22-grape

affect higher values of k. Therefore, for each catalog we determine the *upper bound* of the top-k rate a perfect, but non-oracle, model could reach for all k. Furthermore, a very simple model called *evergreen* serves as lower bound: This model predicts the k most common labels seen during training (based on a frequency distribution of the labels from the training set), independent of the specific input. The comparison with this model is to show whether the GNNs are capable of processing contextual information from the input graph and to prove that the prediction task is indeed non-trivial.

Table 2: Summary of results for component recommendation per catalog: Top-k rate on the test set for $k = 1, \dots, 20$ recommendations. The identifier following the GNN model architecture indicates the component representation.

catalog	model \ k	1	2	3	5	10	15	20
A	Upper Bound	94.1%	99.8%	99.9%	99.9%	100%	100%	100%
	GAT-100	46.7%	71.4%	79.7%	85.2%	**90.0%**	**92.2%**	**93.3%**
	GAT-20	47.9%	**72.2%**	**80.4%**	**85.5%**	89.8%	91.5%	92.5%
	GAT-one-hot	46.0%	70.3%	79.0%	84.6%	89.6%	91.5%	92.6%
	GCN-100	47.4%	71.0%	79.3%	84.9%	89.6%	91.5%	92.7%
	GCN-20	46.6%	70.9%	79.5%	84.8%	89.4%	91.3%	92.4%
	GCN-one-hot	45.4%	69.3%	78.1%	83.8%	88.6%	90.7%	92.0%
	Baseline	**57.5%**	64.3%	66.9%	69.0%	70.2%	70.7%	70.8%
	Evergreen	4.5%	6.3%	8.1%	11.3%	15.9%	19.6%	22.2%
B	Upper Bound	63.5%	81.5%	92.5%	99.4%	99.99%	100%	100%
	GAT-100	**30.8%**	**50.6%**	**63.1%**	**73.8%**	**82.1%**	**86.1%**	**88.4%**
	GAT-20	30.0%	50.1%	62.6%	73.4%	82.0%	85.9%	88.0%
	GAT-one-hot	30.2%	49.5%	61.8%	72.5%	80.7%	84.6%	86.6%
	GCN-100	30.5%	49.2%	61.2%	72.4%	81.8%	85.9%	88.0%
	GCN-20	28.0%	46.4%	58.8%	71.4%	81.3%	85.6%	88.1%
	GCN-one-hot	27.8%	45.1%	56.8%	69.2%	79.8%	84.3%	86.7%
	Baseline	24.5%	32.9%	39.1%	46.6%	52.8%	53.7%	53.8%
	Evergreen	13.3%	15.5%	18.4%	22.2%	30.0%	36.2%	41.7%
C	Upper Bound	74.0%	91.1%	98.0%	99.9%	100%	100%	100%
	GAT-100	28.5%	**49.3%**	**65.0%**	**81.8%**	**92.8%**	**95.8%**	96.9%
	GAT-20	27.9%	48.5%	64.0%	80.7%	91.9%	95.1%	96.3%
	GAT-one-hot	27.5%	48.4%	64.0%	80.6%	91.2%	94.0%	95.2%
	GCN-100	27.9%	48.3%	63.8%	80.5%	92.0%	95.5%	**97.1%**
	GCN-20	27.5%	48.4%	64.2%	81.2%	92.5%	**95.8%**	**97.1%**
	GCN-one-hot	27.0%	47.1%	62.4%	79.2%	90.8%	94.2%	95.7%
	Baseline	**30.1%**	42.1%	50.9%	60.2%	63.9%	64.2%	64.2%
	Evergreen	14.4%	19.9%	23.3%	27.8%	37.0%	44.6%	48.1%

Table 2 presents an overview of the performance of the models as well as the upper and lower bounds for the top-k rate. Additionally, Figure 2 visualizes



(a) Catalog A (b) Catalog B

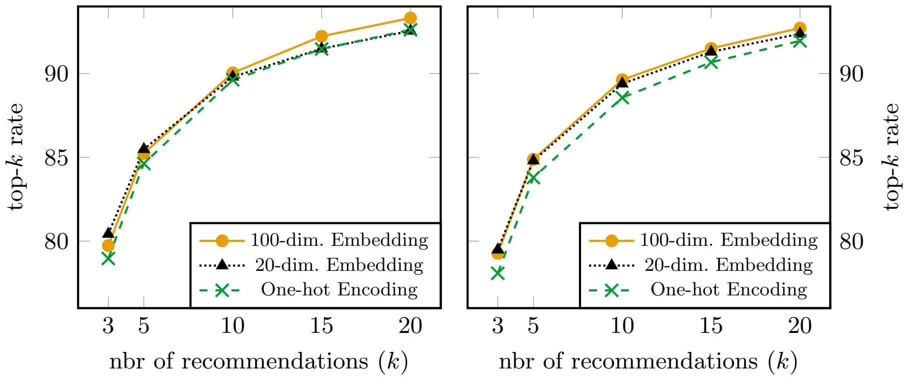
Fig. 2: Visual comparison of component prediction models: For GCN and GAT, the best models (embedding size 100) were used for each catalog.

the performance of the best models and bounds on catalog A and B. Both the baseline and the GNN models stand out clearly from the simple ever-green model for each catalog, with the latter performing considerably better. When evaluating the baseline model, a direct lookup of the input graph could only rarely be performed, on catalog B only for 1% of the test instances. Thus, in most cases subgraphs had to be found, which together form the input graph. In contrast to GNNs, the baseline model does not scale in terms of the size of the catalogs or the size of assemblies, making GNN models more suitable for larger data sets.

In all but the case of only one recommendation ($k = 1$), the GNN models outperform the baseline by a large margin, demonstrating that they are capable of generalizing beyond exact subgraph pattern-matching. Since our application scenario focuses on presenting the design engineers with more than one recommendation, the case $k = 1$ is negligible. Furthermore, the performance of the GNN models increases significantly more with growing number of recommendations k. Regarding the top-10 rate, the best performing model (GAT-100) achieved over 90% on catalogs A and C, and still 82.1% for catalog B despite the fact that this catalog is more ambiguous than the other two since it comprises more components – which is also reflected in the lower and upper bound rates. This demonstrates that the GNN models can reliably cut down the candidates for needed components from 1,930 (catalog A), 3,099 (catalog B), and 1,924 (catalog C) to 10, providing a useful preselection for design engineers.

4.3 Are component embeddings better than one-hot node features?

Regarding the used representation of the components, the results show that pretraining embeddings pays off compared to starting with one-hot node fea-



(a) GAT Model (b) GCN Model

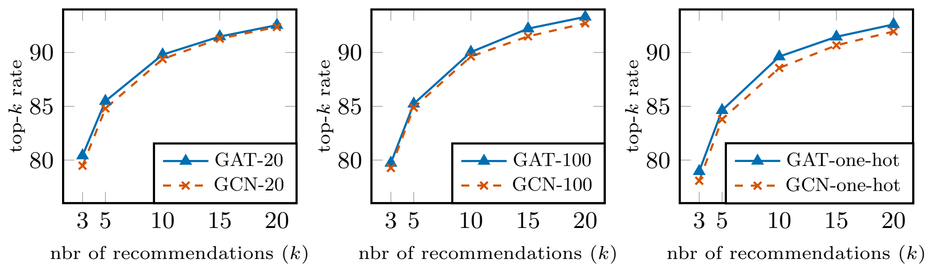
Fig. 3: Effects of the three component representations on the performance (top-k-rate) of both GNN models for catalog A. $k = 1, 2$ omitted due to scaling.

tures. In particular, the GNNs based on 100-dimensional embeddings perform slightly better than those using 20-dimensional ones, respectively. The improvements with increasing embedding dimensions is consistent for both GCNs and GATs, as visualized in Figure 3. This suggests that an even higher embedding dimension would lead to a further performance improvement, but the preliminary study did not show any improvement in the pretraining task with higher embedding dimensions. Therefore, we decided not to investigate them further in the component prediction task.

Thinking of the recommendation models as extrinsic evaluation for the component embeddings, i.e., using the downstream supervised model to evaluate the embeddings, the representation in 100 dimensions is preferable. Since in the preliminary study sizes of 100 and above resulted in lower loss values than only 20 dimensions, this confirms our assumption from Section 3.1 that we can measure the quality of an embedding by the associated supervised task of comp2vec. Concluding, pretraining turned out to be advantageous for the recommendation task, although one-hot-based GNNs also performed well which could be relevant for practical purposes (e.g., lack of time for setting up a pretraining pipeline).

4.4 Comparing GAT and GCN

In a direct comparison of the GNNs, the GAT models yield slightly better results based on the same embedding, as shown in Figure 4. This makes the GAT with embedding size 100 the overall winner. The difference between the two models lies in the weighting of the neighbors: while in GCN all neighbors are weighted with the same constant factor depending on the graph structure (cf. Equation (2)), in GAT an individual weight for each neighbor node is determined based on the node features, i.e., the attention scores (cf. Equation (3)). Whether



(a) 20-dim. Embedding (b) 100-dim. Embedding (c) One-hot Encoding

Fig. 4: Comparison of the performance of GCN and GAT trained on the same component representation for catalog A. $k = 1, 2$ omitted due to scaling.

a GAT indeed weights certain neighbor nodes higher (i.e., makes use of the attention mechanism) can be investigated by comparing the attention distribution to the uniform distribution, as suggested in [23]. For each graph instance and each of its nodes, the attention distribution over the neighbor nodes as well as the corresponding uniform distribution is determined. Subsequently, the Shannon entropy is calculated for both distributions. Figure 5 shows a histogram of the calculated entropy values (attention distribution and uniform distribution), excluding nodes with exactly one neighbor node (i.e., leaf nodes), since for those the learned attention distribution always corresponds to the uniform distribution. Since the characteristics were similar for all GAT models, the figure visualizes only one particular model (GAT-100 for catalog B). The high prevalence of entropy 0 scores indicates that for many neighborhoods in the assemblies, the attention scores tend to focus all attention on a single (or very few) neighbors – which result in a very different distribution from the uniform one, and explains why GATs are superior to GCNs on these data sets.

Finally, we want to stress that the GATs behaved more robustly in our experiments: Slight changes in the hyperparameters led to strong fluctuations in performance for GCNs, while this phenomenon was much less noticeable for GATs. In summary, after analyzing the resulting attention scores and evaluating the performance, we favor GATs over GCNs for our recommendation task.

5 Conclusion and Future Work

We proposed a recommendation system based on Graph Neural Networks for assembly modelling that recommends next required components during construction. For this purpose, we developed an approach to construct instances in a self-supervised fashion by recursively cutting off nodes from assembly graphs which then serve as target for the resulting partial graph instance. Our experiments on three different data sets proved that graph neural networks are well suited for this task, outperforming a self-developed frequency-based baseline model. Further, pretraining low-dimensional representations of components as

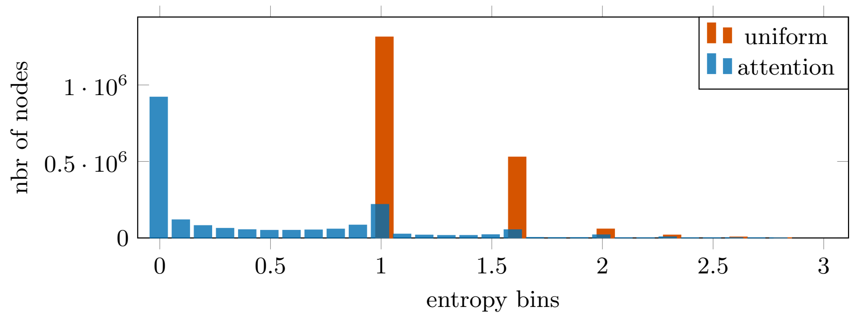


Fig. 5: Histogram of calculated Shannon-Entropy of the attention distribution and the corresponding uniform distribution evaluated on the test set. The attention values originate from an arbitrarily selected head of the last GAT layer from GAT-100 trained on catalog B. Nodes with only one neighbor were excluded, since their distribution equals the uniform distribution.

node features turned out to be beneficial for recommendation, although one-hot encoded node features also led to satisfying results.

In the presented approach, the components are recommended without locating which node of the current assembly they should be attached to. While this is acceptable for small assemblies, for designing larger ones conveniently, it is mandatory to highlight the node where a component should be attached. In future work, we want to integrate these *connection components* into the recommendation. Moreover, as the catalogs get updated over time, the models are likely to be confronted with new, unknown components during inference – here a suitable representation of the new components has to be found. Literature provides several other models, e.g., [21] and [22], to process graph-like data, each of them having a different inductive bias. We plan to further investigated them in terms of their applicability for our use case.

References

1. Bakarov, A.: A Survey of Word Embeddings Evaluation Methods (2018). https://doi.org/10.48550/ARXIV.1801.09536
2. Bronstein, M.M., Bruna, J., Cohen, T., Veličković, P.: Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges (2021). https://doi.org/10.48550/ARXIV.2104.13478
3. Cunningham, J.D., Simpson, T.W., Tucker, C.S.: An Investigation of Surrogate Models for Efficient Performance-Based Decoding of 3D Point Clouds. Journal of Mechanical Design **141**(12) (2019). https://doi.org/10.1115/1.4044597
4. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
5. Grover, A., Leskovec, J.: Node2vec: Scalable Feature Learning for Networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 855–864. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939754

6. Guo, X., Zhao, L.: A Systematic Survey on Deep Generative Models for Graph Generation (2020). https://doi.org/10.48550/ARXIV.2007.06686
7. Guo, X., Li, W., Iorio, F.: Convolutional Neural Networks for Steady Flow Approximation. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 481–490. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939738
8. Hamilton, W., Ying, Z., Leskovec, J.: Inductive Representation Learning on Large Graphs. In: Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)
9. Hamilton, W.L.: Graph representation learning. Synthesis Lectures on Artifical Intelligence and Machine Learning **14**(3), 1–159 (2020)
10. Kipf, T.N., Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks (2017). https://doi.org/10.48550/ARXIV.1609.02907
11. Li, W., Matejka, J., Grossman, T., Konstan, J.A., Fitzmaurice, G.: Design and Evaluation of a Command Recommendation System for Software Applications. ACM Transactions on Computer-Human Interaction (TOCHI) **18**(2), 1–35 (2011). https://doi.org/10.1145/1970378.1970380
12. Li, Y., Vinyals, O., Dyer, C., Pascanu, R., Battaglia, P.: Learning Deep Generative Models of Graphs (2018). https://doi.org/10.48550/ARXIV.1803.03324
13. Lupinetti, K., Pernot, J.P., Monti, M., Giannini, F.: Content-based CAD assembly model retrieval: Survey and future challenges. Computer-Aided Design **113**, 62–81 (2019). https://doi.org/10.1016/j.cad.2019.03.005
14. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient Estimation of Word Representations in Vector Space (2013). https://doi.org/10.48550/ARXIV.1301.3781
15. Sarcar, M., Rao, K.M., Narayan, K.L.: Computer aided design and manufacturing. PHI Learning Pvt. Ltd. (2008)
16. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph Attention Networks (2018). https://doi.org/10.48550/ARXIV.1710.10903
17. Wu, R., Xiao, C., Zheng, C.: DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). pp. 6772–6782 (2021)
18. Wu, S., Sun, F., Zhang, W., Xie, X., Cui, B.: Graph Neural Networks in Recommender Systems: A Survey (2020). https://doi.org/10.48550/ARXIV.2011.02260
19. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A Comprehensive Survey on Graph Neural Networks. IEEE Transactions on Neural Networks and Learning Systems **32**(1), 4–24 (2021). https://doi.org/10.1109/tnnls.2020.2978386
20. Yoo, S., Lee, S., Kim, S., Hwang, K.H., Park, J.H., Kang, N.: Integrating Deep Learning into CAD/CAE System: Generative Design and Evaluation of 3D Conceptual Wheel. Structural and Multidisciplinary Optimization **64**(4), 2725–2747 (2021)
21. Yun, S., Jeong, M., Kim, R., Kang, J., Kim, H.J.: Graph Transformer Networks. In: Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019)
22. Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R.R., Smola, A.J.: Deep sets. In: Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)
23. Zhang, H., Li, M., Wang, M., Zhang, Z.: Understand Graph Attention Network. https://www.dgl.ai/blog/2019/02/17/gat.html (02 2022), accessed: 06.04.2022

24. Zhang, Z., Jaiswal, P., Rai, R.: Featurenet: Machining feature recognition based on 3d convolution neural network. Computer-Aided Design **101**, 12–22 (2018). https://doi.org/10.1016/j.cad.2018.03.006