GRAPH PARSING NETWORKS

Yunchong Song¹, Siyuan Huang¹, Xinbing Wang¹, Chenghu Zhou², Zhouhan Lin¹*
¹Shanghai Jiao Tong University, ²Chinese Academy of Sciences
{ycsong, xwang8}@sjtu.edu.cn, lin.zhouhan@gmail.com

ABSTRACT

Graph pooling compresses graph information into a compact representation. State-of-the-art graph pooling methods follow a hierarchical approach, which reduces the graph size step-by-step. These methods must balance memory efficiency with preserving node information, depending on whether they use node dropping or node clustering. Additionally, fixed pooling ratios or numbers of pooling layers are predefined for all graphs, which prevents personalized pooling structures from being captured for each individual graph. In this work, inspired by bottom-up grammar induction, we propose an efficient graph parsing algorithm to infer the pooling structure, which then drives graph pooling. The resulting Graph Parsing Network (GPN) adaptively learns personalized pooling structure for each individual graph. GPN benefits from the discrete assignments generated by the graph parsing algorithm, allowing good memory efficiency while preserving node information intact. Experimental results on standard benchmarks demonstrate that GPN outperforms state-of-the-art graph pooling methods in graph classification tasks while being able to achieve competitive performance in node classification tasks. We also conduct a graph reconstruction task to show GPN’s ability to preserve node information and measure both memory and time efficiency through relevant tests.

1 INTRODUCTION

Graph neural networks (GNNs) (Scarselli et al., 2008; Bruna et al., 2013; Defferrard et al., 2016; Kipf & Welling, 2017; Veličković et al., 2018; Hamilton et al., 2017; Gilmer et al., 2017; Xu et al., 2019) have shown remarkable success in processing graph data from various fields (Li & Goldwasser, 2019; Yan et al., 2019; Suárez-Varela et al., 2021; Kipf et al., 2018). Graph pooling is built on top of GNNs. It aims to capture graph-level information by compressing a set of nodes and their underlying structure into a more concise representation. Early graph pooling methods, such as mean or add pool (Atwood & Towsley, 2016; Xu et al., 2019), perform permutation-invariant operations on all nodes in a graph. These flat pooling methods disregard the distinctions between nodes and fail to model the latent hierarchical structure within a graph. Thus, they can only capture information at the graph level in a rough manner. To overcome this limitation, researchers drew inspiration from pooling designs in computer vision (Ronneberger et al., 2015; Noh et al., 2015; Jégou et al., 2017), they turned to perform hierarchical pooling on graph data to capture structural information more accurately. The dominant approaches are node dropping (Gao & Ji, 2019; Lee et al., 2019) and node clustering (Ying et al., 2018; Bianchi et al., 2020). Each approach has its own advantages and disadvantages: node dropping methods reduce the size of a graph by dropping a certain proportion of nodes at each step. While they are memory-efficient, they irreversibly lose node information and may damage graph connectivity (Baek et al., 2021; Wu et al., 2022), leading to sub-optimal performance. On the other hand, node clustering methods perform clustering at each pooling layer, allowing for soft cluster assignments for each node, thus preserving its information. However, this comes at the cost of a dense cluster assignment matrix with quadratic memory complexity (Baek et al., 2021; Liu et al., 2023), sacrificing scalability. The comparison of graph pooling methods can refer to Figure 1. Hierarchical pooling methods need to strike a balance between preserving node information and being memory efficient, while our model excels in both aspects.

*Zhouhan Lin is the corresponding author.
1Code is available at https://github.com/LUMIA-Group/GraphParsingNetworks
Another fundamental problem of existing graph pooling methods is that they are unable to learn personalized pooling structure for each individual graph (Liu et al., 2023), i.e., they predefine a fixed pooling ratio or number of layers for all graphs. However, some graphs may be more compressible compared to others and imposing a constant pooling ratio may hurt performance on less compressible ones (Gao & Ji, 2019; Ranjan et al., 2020; Wu et al., 2022), leading to suboptimal performance. We represent the graph pooling structure as a rooted-tree to illustrate the problem, which we name the pooling tree (cf. Figure 1). The root node represents the graph-level representation, while the leaf nodes are taken from the input graph. The nodes produced by the $k$-th pooling layer correspond to the height $k$ of the tree. Ideally, the model should learn a personalized pooling tree for each graph with a flexible shape. This means that we should not constrain the total height $h_{\text{max}}$ or the number of nodes $n_k$ at height $k$. There are numerous methods that focus on optimizing graph structure (Gasteiger et al., 2019; Jiang et al., 2019; Chen et al., 2020; Jin et al., 2020; Suresh et al., 2021; Topping et al., 2022; Bo et al., 2023). However, only a few (Diehl, 2019; Wu et al., 2022) attempt to optimize pooling structure, and even those can only learn a partially flexible shape of the pooling tree and not in an end-to-end fashion. In Appendix B and Appendix C, we discuss these methods in detail and highlight the differences. Our model is the first to end-to-end learning of a fully flexible pooling tree. To overcome these two crucial problems, we propose a graph parsing algorithm inspired by recent advancements in bottom-up grammar induction (Shen et al., 2018b; a; 2019). In natural language processing, grammar induction techniques extract latent discrete structures from input text sequences without any guided supervision. We adapted this approach for the graph domain and used it to learn pooling trees. As shown in Figure 2, our model employs a graph parsing algorithm that assigns a discrete assignment to each node at layer $k$, allowing us to infer the number of nodes $n_k$ at height $k$. The parsing process ends when the graph has been fully parsed (i.e., the graph size remains unchanged), enabling us to infer the total height $h_{\text{max}}$. Therefore, our model can learn a pooling tree with a flexible shape for each individual graph in an end-to-end fashion. Our parsing algorithm is based on locality, thus ensures graph connectivity during pooling. We clustering nodes instead of dropping them, which preserves node information intact, and the discrete clustering process guarantees memory efficiency. Jo et al. (2021) also performs edge-centric graph pooling, however, they still need to pre-define the number of pooling layers or pooling ratio. A more detailed introduction for related works can be found in Appendix B.

2 PROPOSED METHOD

In this section, we first divide the graph pooling layer into three modules and introduce our design for each. Then, we present the details of our proposed graph parsing algorithm. Finally, we introduce the model architecture that is applied to both graph classification and node classification tasks.

2.1 GRAPH POOLING COMPONENTS

We represent a graph $G = (\mathcal{V}, \mathcal{E})$, $|\mathcal{V}| = n$, $|\mathcal{E}| = m$ with its feature matrix $X \in \mathbb{R}^{n \times d}$ and adjacent matrix $A \in \mathbb{R}^{n \times n}$, where $n, m, d$ represents the number of nodes, the number of edges and the
As such, when edge scores are updated through gradients, the pooling structure can also be updated. We infer clusters from nodes in a manner similar to how larger units (e.g., clauses) are inferred from smaller units (e.g., phrases) in grammar induction. In order to make the pooling structure learnable, we implement this module using DeepSets:

\[ H^{(k)} = \text{GNN} \left( X^{(k)}, A^{(k)} \right), C_{i,j}^{(k)} = \sigma \left( \text{MLP} \left( H_{i,:}^{(k)} \odot H_{j,:}^{(k)} \right) \right) \]  

\( \sigma(\cdot) \) here is a sigmoid function. \( C^{(k)} \) is the edge score matrix that satisfy \( C^{(k)} = C^{(k)} \odot A^{(k)} \). In practice, we simply implement the GNN block with multiple GCN layers.

**Graph information encoding** This module extracts node features and graph structure to generate useful metrics. For instance, it can generate node scores for node dropping methods or cluster assignment vectors for node clustering methods. Considerable effort has been put into improving the quality of these metrics, as evidenced by the advanced designs discussed in Section 3. To encode structural information, we apply a GNN block and then use an MLP on adjacent nodes to calculate the edge score:

\[ H^{(k)} = \text{GNN} \left( X^{(k)}, A^{(k)} \right), C_{i,j}^{(k)} = \sigma \left( \text{MLP} \left( H_{i,:}^{(k)} \odot H_{j,:}^{(k)} \right) \right) \]  

\( \sigma(\cdot) \) here is a sigmoid function. \( C^{(k)} \) is the edge score matrix that satisfy \( C^{(k)} = C^{(k)} \odot A^{(k)} \). In practice, we simply implement the GNN block with multiple GCN layers.

**Structural transformation** This module consists of two steps: node assignment and output graph construction. In the \( k \)-th pooling layer, the first step involves mapping nodes from the input graph to the output graph, resulting in an assignment matrix \( S^{(k)} \in \mathbb{R}^{n^{(k)} \times n^{(k+1)}} \). \( S^{(0)}, S^{(1)}, \ldots, S^{(K)} \) fully encode the pooling tree with height \( K \). The primary challenge here is to learn a flexible pooling tree for each graph, as we described in Section 1. We propose a graph parsing algorithm \( \mathcal{A} \) to construct node assignment and perform matrix multiplication for the second step:

\[ S^{(k)} = A \left( C^{(k)} \right), A^{(k+1)} = S^{(k)^T} A^{(k)} S^{(k)} \]  

**Multiset computation** This module computes the feature matrix \( X^{(k+1)} \) for \( G^{(k+1)} \). As the input node set can be treated as a multiset, this module can be seen as a multiset function (Baek et al., 2021). Simple implementations include operators such as ADD(\cdot), MEAN(\cdot), and MAX(\cdot). Recent studies have explored more expressive methods like DeepSets (Zaheer et al., 2017), or attention mechanisms (Baek et al., 2021). We implement this module using DeepSets:

\[ \hat{X}^{(k+1)} = \text{DeepSets} \left( H^{(k)}, S^{(k)} \right) = \text{MLP}_2 \left( S^{(k)^T} \text{MLP}_1 \left( H^{(k)} \right) \right) \]  

To update the MLP in Equation 1, we need to involve edge scores in backpropagation since the graph parsing algorithm \( \mathcal{A} \) is not differentiable. To achieve this, we compute a mask \( y^{(k)} \) through edge scores. Let us consider a subgraph \( \hat{G}^{(k)} \subseteq G^{(k+1)} \) induced by cluster \( v_i \) in \( G^{(k+1)} \). We denote edges inside \( \hat{G}^{(k)} \) as \( \hat{E}^{(k)} \), and \( 1 \) is a \( d \)-dimension one vector. Through adding the score and then multiplying to the features, the cluster can be aware of the number of edges assigned to it:

\[ X^{(k+1)} = \hat{X}^{(k+1)} \odot \left( y^{(k)} 1^T \right), y_i^{(k)} = \text{ADD} \left( \left\{ C^{(k)}_{j,k} | (j,k) \in \hat{E}^{(k)} \right\} \right) \]  

2.2 **Graph Parsing Algorithm**

Our graph parsing algorithm \( \mathcal{A} \) takes edge score matrix \( C^{(k)} \) as input and returns an assignment matrix \( S^{(k)} \). We do not impose any constraints on the pooling ratio or the number of clusters. Therefore, we infer clusters from nodes in a manner similar to how larger units (e.g., clauses) are inferred from smaller units (e.g., phrases) in grammar induction. In order to make the pooling structure learnable, we rely on both the graph topology and a set of continuous values defined on it, namely edge scores. As such, when edge scores are updated through gradients, the pooling structure can also be updated via decoding from it. To elaborate on this decoding process, we first introduce three operators that are utilized within it.

**Definition 1.** DOM(\cdot) (short for “dominant”): this operator selects the dominant edge for each node, which determines its assignment. First it performs a row-wise max indexing on the edge score matrix \( C^{(k)} \) to obtain seed edges idx, then construct matrix \( \hat{C}^{(k)} \) by filtering out other edges:

\[ idx = \arg\max_{\text{row-wise}}(C^{(k)}), \hat{C}^{(k)}_{i,j} = \begin{cases} C^{(k)}_{i,j}, & (i,j) \in \text{idx} \\ 0, & \text{otherwise} \end{cases} \]  

Published as a conference paper at ICLR 2024
Figure 2: Proposed graph parsing networks, driven by the graph parsing algorithm $\mathcal{A}$. The box with three stages indicates the graph pooling layer, which recurrently builds the entire model. The red box shows the details and an example of how $\mathcal{A}$ works.

**Definition 2.** $\exp(\cdot)$ (short for “expand”): this operator expands a set of seed edges $idx$ on the filtered graph $\hat{C}^{(k)}$ through a neighbor lookup. First, induce the node set $l$ using $idx$, then merge neighboring edges $idx'$ that are dominated by them. $idx_x, idx_y$ represent the sets of starting and ending node indices for the edge set $idx$:

$$l = \text{union}(\{idx_x\}, \{idx_y\}), idx' = \{(i, j)|(i \in l \lor j \in l) \land \hat{C}^{(k)}_{l,j} \neq 0\} \quad (6)$$

**Definition 3.** $\text{gen}(\cdot)$ (short for “generate”): given an assignment mapping $s$ that maps node $i$ to cluster $j$, this operator generates the assignment matrix $S^{(k)}$ using $s$:

$$S_{i,j}^{(k)} = \begin{cases} 1, & (i,j) \in s \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

Based on these definitions, we present the aforementioned decoding process in Algorithm 1. To facilitate understanding, we divided it into three stages. In Stage 1, we use the $\text{dom}(\cdot)$ operator to identify the dominant edge for each node, which determines its assignment. To ensure memory efficient, we opt for a sparse assignment by selecting a single dominant edge for each node. This also protects locality by avoiding instances where a single node is assigned to multiple clusters, which could result in unconnected nodes being assigned to the same cluster. During this stage, we initialize an empty mapping $s$ that associates nodes with clusters. As we create clusters and map nodes to them one by one, we also set up an indicator variable $p$, which keeps track of the latest cluster’s index.

In Stage 2, we use $\hat{C}^{(k)}$ to create $s$. This is done through a double-loop approach, where new cluster is generated in the outer loop and nodes are assigned to it in the inner loop. In the outer loop, we choose the edge with the highest score in $\hat{C}^{(k)}$ as the seed edge. This edge is considered as the first member of a new cluster. In the inner loop, we expand this seed edge iteratively using the $\exp(\cdot)$ operator which performs neighborhood lookup and merges. The inner loop terminates when no additional edges can be covered by expanding from the seed edges, i.e., $|idx| = q$. We then map all the nodes in final node set $l$ to the same $p$-th cluster. After that, we clear all the edges covered in this loop ($idx$) from $\hat{C}^{(k)}$.

**Algorithm 1** Graph parsing algorithm $\mathcal{A}$

**Input:** edge score matrix $C^{(k)}$

**Output:** assignment matrix $S^{(k)}$

1: $\hat{C}^{(k)} \leftarrow \text{dom}(C^{(k)})$ \hspace{1cm} $\triangleright$ **Stage 1**
2: $s \leftarrow \emptyset$
3: $p \leftarrow 0$
4: while $\text{sum}(\hat{C}^{(k)}) \neq 0$ do \hspace{1cm} $\triangleright$ **Stage 2**
5: $idx \leftarrow \text{argmax}(\hat{C}^{(k)})$
6: do
7: $q \leftarrow |idx|$
8: $l, idx' \leftarrow \exp(idx, \hat{C}^{(k)})$
9: $idx \leftarrow \text{union}(idx, idx')$
10: while $|idx| = q$ do
11: $s \leftarrow \text{union}(s, \{(i, p)|i \in l\})$
12: $\hat{C}^{(k)}_{i,j} \leftarrow 0, \forall(i, j) \in idx$
13: $p \leftarrow p + 1$
14: end while
15: $S^{(k)} = \text{gen}(s)$ \hspace{1cm} $\triangleright$ **Stage 3**
16: return $S^{(k)}$
and update the indicator \( p \). In Stage 3, we use the GEN(\( \cdot \)) operator to create an assignment matrix \( S^{(k)} \) with \( s \). If the graph has isolated nodes, we generate a separate cluster for each isolated node and create a corresponding mapping in \( s \).

**Complexity** The time complexity for Algorithm 1 is \( \mathcal{O}(\sum_{i=1}^{n^{(k+1)}} d_i) \), \( d_i \) indicates the diameter of the subgraph \( G_i^{(k+1)} \subseteq G^{(k)} \) induced by node \( v_i \) in \( G^{(k+1)} \). Compared to EdgePool, which has a time complexity of \( \mathcal{O}(m^{(k)}) \), our worst case time complexity of \( \mathcal{O}(n^{(k)}) \) is much smaller. In Appendix A.3, we provide further comparison to SEP and demonstrate that our model is more efficient than it. These results indicate that our model is also **time efficient** and can handle dense or large graphs.

**Proposition 4.** (Proof in Appendix A.1) In \( k \)-th graph pooling layer, the time complexity for Algorithm 1 is \( \mathcal{O}(\sum_{i=1}^{n^{(k+1)}} d_i) \), which is upper-bounded by the worst case \( \mathcal{O}(n^{(k)}) \).

**Permutation invariance** Permutation invariance is a crucial property in graph representation learning as it ensures the model’s ability to generalize on graph data. All the components (GNN, DeepSets) of our model are already permutation invariant, we only focus on analyzing the permutation invariance of the graph parsing algorithm \( \mathcal{A} \).

**Proposition 5.** (Proof in Appendix A.2) Given a permutation matrix \( \mathcal{P} \), suppose the row-wise non-zero entries in \( \mathcal{C} \) are distinct, \( \mathcal{A}(\mathcal{C}) = \mathcal{A}(\mathcal{P}\mathcal{C}\mathcal{P}^T) \), thus \( \mathcal{A} \) is permutation invariant.

**Graph connectivity** Keep the graph connectivity during pooling is important. Our parsing algorithm keep the connectivity of the input graph unchanged during graph pooling, which guarantees a fluent flow of information on the pooled graph.

**Proposition 6.** (Proof in Appendix A.4) For nodes \( i, j \in G^{(k)} \), if there is a path \( s^{(k)}_{ij} \) connect them, then a path \( s^{(k+1)}_{p_i,p_j} \) would exist between the corresponding clusters \( p_i, p_j \in G^{(k+1)} \) after parsing.

### 2.3 Model Architecture

**Graph-level architecture** By utilizing the pooling layer that was introduced in previous sections, we can directly employ this model for graph classification tasks. The architecture of our model (depicted in Figure 3) comprises solely of one pooling layer which progressively decreases the size of the input graph until it matches the output graph produced by our graph parsing algorithm \( \mathcal{A} \). Subsequently, a multiset computation module is utilized to convert the final graph into a single node and generate a compressed vector for graph classification.

**Node-level architecture** We developed an encoder-decoder architecture (refer to Figure 3) for the node classification task, inspired by previous graph pooling methods used in node-level tasks like TopKPool (Gao & Ji 2019) and SEP (Wu et al. 2022). Our approach involves compressing the graph into a smaller size using bottom-up pooling in the encoder block, followed by expanding it in the decoder block. We store the assignment at every pooling layer during the encoding process and use this information for un-pooling in the decoding process. We implement \( k \)-th un-pooling layer as follows:

\[
A^{(k+1)} = S^{(h-k)} A^{(k)} S^{(h-k)} \quad (8)
\]

\[
X^{(k+1)} = S^{(h-k)} X^{(k)} \quad (9)
\]

the variable \( h \) represents the height of the pooling tree in the encoder block. The assignment matrix \( S^{(h-k)} \) is stored in the \( (h-k) \)-th pooling layer and corresponds to the \( k \)-th un-pooling layer. Additionally, skip connections are applied between the encoder and decoder, following methods used in Gao & Ji (2019), Wu et al. (2022).
10-fold cross-validation with 20 random seeds for both model initialization and fold generation. Additionally, we evaluate our model’s performance at scale on six graph classification and four node clustering datasets. For hierarchical pooling methods, we include TopKPool (Gao & Ji, 2019), SAGPool (Lee et al., 2019), HGP-SL (Zhang et al., 2019), and SEP (Wu et al., 2022) for node clustering. Table 1 presents the statistics of them.

An informative node representation is crucial for node-level tasks. To achieve this, we utilize a separate GNN encoder for the multiset computation module. In the node-level architecture, Equation 9 in the pooling layer needs to be rewritten as:

\[
\hat{X}^{(k+1)} = \text{DeepSets} \left( \text{GNN'} \left( \hat{X}^{(k)}, A^{(k)} \right), S^{(k)} \right)
\]

3 EXPERIMENTS

In this section, we evaluate our model’s performance on six graph classification and four node classification datasets. We also demonstrate how the node information is well-preserved through a graph reconstruction task. To assess the efficiency of our parsing algorithm, we conduct time and memory tests. Additionally, we showcase our model’s ability to learn a personalized pooling structure for each graph by visualizing the number of pooling layers during training across different datasets. Finally, an ablation study is performed on all three graph pooling modules to determine their contributions.

3.1 GRAPH CLASSIFICATION

Graph classification requires a model to classify a graph into a label, which calls for graph-level representation learning.

**Experimental setup** We evaluate our model on five widely-used graph classification benchmarks from TUDatasets (Morris et al., 2020): DD, PROTEINS, NCI1, NCI109, and FRANKENSTEIN. Table 1 presents the statistics of them. Additionally, we evaluate our model’s performance at scale on ogbg-molpcba, one of the largest graph classification datasets in OGB (Hu et al., 2020), comprising over 400,000 graphs. For baselines, we select GCN (Kipf & Welling, 2017) (with simple mean pooling), Set2Set (Vinyals et al., 2016), Global-Attention (Li et al., 2016), SortPool (Zhang et al., 2018), HoscPool (Duval & Malliaros, 2022), and GMT (Baek et al., 2021) from the flat-pooling family. For hierarchical pooling methods, we include TopKPool (Gao & Ji, 2019), SAGPool (Lee et al., 2019), HGP-SL (Zhang et al., 2019), GSA- Pool (Zhang et al., 2020), ASAP (Ranjan et al., 2020), and SPGP (Lee et al., 2022) for node dropping and DiffPool (Yang et al., 2018), EdgePool (Dichi, 2019), MinCutPool (Bianchi et al., 2020), HoscPool (Duval & Malliaros, 2022), and SEP (Wu et al., 2022) for node clustering. We adopt 10-fold cross-validation with 20 random seeds for both model initialization and fold generation. This is a special case in node clustering based pooling, since it computes a sparse pooling tree before training, while others generate dense assignments for each forward pass.
Table 3: Node classification accuracy with mean and standard deviation based on 10 splits, we bold the model with best performance.

<table>
<thead>
<tr>
<th></th>
<th>Cora</th>
<th>CiteSeer</th>
<th>PubMed</th>
<th>Film</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>2.708</td>
<td>3.327</td>
<td>18.717</td>
<td>7.600</td>
</tr>
<tr>
<td># Edges</td>
<td>5.278</td>
<td>4.676</td>
<td>44.327</td>
<td>26.752</td>
</tr>
<tr>
<td># Classes</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>MLP</td>
<td>75.69±2.00</td>
<td>74.02±1.90</td>
<td>87.16±0.37</td>
<td>36.53±2.00</td>
</tr>
<tr>
<td>GCN [Kipf &amp; Welling 2017]</td>
<td>86.98±1.27</td>
<td>76.50±1.36</td>
<td>88.42±0.56</td>
<td>37.32±1.10</td>
</tr>
<tr>
<td>GraphSAGE [Hamilton et al. 2017]</td>
<td>86.90±1.04</td>
<td>76.64±1.30</td>
<td>88.45±0.50</td>
<td>34.23±0.99</td>
</tr>
<tr>
<td>GAT [Velickovic et al. 2018]</td>
<td>86.33±0.48</td>
<td>76.55±1.23</td>
<td>87.30±1.10</td>
<td>27.44±0.89</td>
</tr>
<tr>
<td>MixHop [Abu-El-Haija et al. 2019]</td>
<td>87.61±0.85</td>
<td>76.26±1.33</td>
<td>85.31±1.61</td>
<td>32.22±2.34</td>
</tr>
<tr>
<td>PairNorm [Zhao &amp; Akoglu 2020]</td>
<td>85.79±1.01</td>
<td>73.59±1.47</td>
<td>87.55±0.44</td>
<td>27.40±1.24</td>
</tr>
<tr>
<td>Geom-GCN [Pei et al. 2020]</td>
<td>85.35±1.57</td>
<td>78.02±1.15</td>
<td>89.95±0.47</td>
<td>31.59±1.15</td>
</tr>
<tr>
<td>GCNII [Chen et al. 2020a]</td>
<td>88.37±2.26</td>
<td>77.33±1.48</td>
<td>90.15±0.48</td>
<td>37.44±1.30</td>
</tr>
<tr>
<td>H2GCN [Zhu et al. 2020]</td>
<td>87.87±1.20</td>
<td>77.11±1.57</td>
<td>89.49±0.38</td>
<td>35.70±1.00</td>
</tr>
<tr>
<td>GPRGNN [Chien et al. 2021]</td>
<td>87.95±1.18</td>
<td>77.13±1.67</td>
<td>87.34±0.38</td>
<td>34.63±1.22</td>
</tr>
<tr>
<td>GCCN [Yan et al. 2022]</td>
<td>87.95±1.05</td>
<td>77.14±1.45</td>
<td>89.15±0.37</td>
<td>37.54±1.56</td>
</tr>
<tr>
<td>NSG [Bojinaru et al. 2022]</td>
<td>87.30±1.15</td>
<td>77.14±1.85</td>
<td>89.49±0.40</td>
<td>37.81±1.15</td>
</tr>
<tr>
<td>TopkPool [Gao &amp; Ji 2019]</td>
<td>79.63±2.82</td>
<td>71.97±3.54</td>
<td>85.47±0.71</td>
<td>–</td>
</tr>
<tr>
<td>NEF [Wu et al. 2020]</td>
<td>87.20±1.12</td>
<td>75.34±1.20</td>
<td>87.73±0.35</td>
<td>–</td>
</tr>
<tr>
<td>GPN</td>
<td>88.07±0.87</td>
<td>77.02±1.85</td>
<td>89.01±0.34</td>
<td>38.11±2.29</td>
</tr>
</tbody>
</table>

means there are 200 runs in total behind each data point, ensuring a fair comparison. This approach is consistent with Lee et al. (2019), Ranjan et al. (2020), and Lee et al. (2022). More details about datasets, implementations and model tuning can be found in Appendix H.1.

Classification results Table 1 shows that our model outperforms SOTA graph pooling methods on four out of five datasets, demonstrating its ability to capture graph-level information precisely. The sub-optimal performance of our model on the DD dataset can be attributed to graph size, as DD has an average node count of 284.3, making it the dataset with large and sparse graphs that pose long-range issues (Alon & Yahav 2020). Resolving such issue requires global information, the complete attention graph in GMT (Baek et al. 2021) provide such information shortcut, thus they perform better. Notably, our method does not include these designs, yet it has achieved considerable results on DD. On the large-scale ogbg-molpcba dataset, our model outperforms all SOTA graph pooling models and significantly improves upon GCN’s performance, demonstrating the superior scalability of our model, and the potential to handle complex large scale data.

3.2 NODE CLASSIFICATION

The node classification task focuses on node-level information. Each graph node is assigned a label, and the goal is to predict its label.

Experimental setup We conduct node classification experiments under full supervision for four datasets: Cora, CiteSeer, and PubMed from Sen et al. (2008), and Film from Per et al. (2020). Table 3 shows the datasets’ statistics. To evaluate our model’s performance comprehensively, we compare it with a range of baselines, including classical methods: GCN [Kipf & Welling 2017], GAT [Velickovic et al. 2018], GraphSAGE [Hamilton et al. 2017], PairNorm [Zhao & Akoglu 2020], MixHop (Abu-El-Haija et al. 2019); and SOTA methods: Geom-GCN (Pei et al. 2020), GCNII (Chen et al. 2020a), GPRGNN (Chien et al. 2021), H2GCN (Zhu et al. 2020), GCCN (Yan et al. 2022), and NSG (Bojinaru et al. 2022). We further compare with two graph pooling methods, TopKPool (Gao & Ji 2019) and SEP (Wu et al. 2022), as other pooling models cannot scale to thousands of nodes or lack a node-level architecture. We follow the same dataset splitting and settings as Zhu et al. (2020), Yan et al. (2022) and Bodnar et al. (2022) and report the average results of 10 dataset splits. Dataset and implementation details can be found in Appendix H.2.

Table 2: Graph classification accuracy on large scale dataset ogbg-molpcba.

<table>
<thead>
<tr>
<th></th>
<th>437,929</th>
<th>26.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># Graphs</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td># Nodes (Avg.)</td>
<td>76.65±0.31</td>
<td>87.30±0.97</td>
</tr>
<tr>
<td># Classes</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>GCN [Kipf &amp; Welling 2017]</td>
<td>20.20±0.24</td>
<td>23.48±0.41</td>
</tr>
<tr>
<td>HGSC [Zhuang et al. 2019]</td>
<td>18.64±0.29</td>
<td>23.70±0.50</td>
</tr>
<tr>
<td>EdgePool [Dinh 2019]</td>
<td>18.64±0.29</td>
<td>23.48±0.41</td>
</tr>
<tr>
<td>ASAP [Ranjan et al. 2020]</td>
<td>OOM</td>
<td>23.95±0.97</td>
</tr>
<tr>
<td>GMT [Baek et al. 2021]</td>
<td>23.70±0.50</td>
<td>23.70±0.50</td>
</tr>
<tr>
<td>SPGP [Lee et al. 2022]</td>
<td>23.95±0.97</td>
<td>23.95±0.97</td>
</tr>
</tbody>
</table>
Classification results The results in Table 3 demonstrate the competitiveness of our model on the selected datasets. Our approach achieves SOTA performance on Film dataset and performs comparably to the SOTA model for node classification, GCNII, on Cora and PubMed datasets. Furthermore, we surpass most baseline models on CiteSeer dataset. Since the compared SOTA methods are explicitly designed for node-level tasks and did not extend to graph-level, our overall performance is satisfactory. Notably, when compared with the leading graph pooling model SEP, we outperform it on all datasets. This reveals the superior generality across graph-level and node-level tasks of our proposed method.

3.3 Graph Reconstruction

To demonstrate our model’s ability to maintain node information, we conduct a graph reconstruction experiment. This can visually illustrate the potential structural damage from graph pooling.

Experimental setup We follow [Bianchi et al. (2020)] to construct a graph autoencoder that includes pooling and un-pooling operations on the input graph $G = (X, A)$ to recover the original node features. The recovered graph $G' = (X', A')$ has the same structure as the input graph, but the node features are produced solely by the pooling and un-pooling process. The mean square error loss $|X - X'|^2$ is used to train the autoencoder. We use the coordinates in a 2-D plane as the node features, thus the recovered node features can be visually assessed to evaluate the pooling model’s ability to preserve node information. We test on the grid and ring graphs. We choose TopKPool and MinCutPool as baselines, which represent the node dropping and node clustering methods, respectively. More baseline results and higher resolution images can be found in Appendix H.3.

Reconstruction results Figure 4 shows the graph reconstruction results on a 2D plane. It is obvious that TopKPool, which drops nodes, damages node information and performs poorly on two graphs. In contrast, MinCutPool uses clustering and preserves the original coordinates better. Compared to these methods, our model recovers a clearer ring graph and effectively preserves the central part of the grid graph, demonstrating its ability to preserve node information intact.

3.4 Efficiency Study

In this section, we assess the memory and time efficiency of our proposed model. Since models from the same class will have similar memory and time performance, we choose one model from each class of pooling methods as our baseline. Specifically, we select GMT, TopKPool, MinCutPool, EdgePool for flat pooling, node dropping, node clustering, and edge-scoring based pooling respectively.

Memory efficiency We test memory efficiency as described in [Baek et al. (2021)]. Varying numbers of nodes ($n$) were used to generate Erdos-Renyi graphs, while keeping the number of edges constant at $2n$ per graph. Applying pooling to each graph once, we compare our model’s GPU memory usage with baselines (Figure 5). Our parsing-based model shows much better memory efficiency when compared to MinCutPool. This confirms our approach’s memory-efficient nature.

Time efficiency To test time efficiency, we created Erdos-Renyi graphs with $m = n^2/10$, and time one forward graph pooling as in [Baek et al. (2021)]. Our results, as shown in Figure 6, demonstrate that GPN is much more efficient than EdgePool, a model also based on edge-scoring. This is because our model has an upper-bound time complexity of $O(n)$, making it able to handle a large number of edges, whereas EdgePool’s time complexity of $O(m)$ would result in significant delays when processing too many edges.
Figure 5: Memory efficiency test, “x” indicate the out-of-memory error.

Figure 6: Time efficiency test, “x” indicate the out-of-memory error.

Figure 7: Number of pooling layers visualization during model training.

3.5 Visualization

In this section, we verify if our model learns personalized pooling structure for each graph. To achieve this, we have created a visualization in Figure 7 that shows the number of pooling layers learned during training on various datasets. Each data point and its shadow represent the mean and variance of the number of pooling layers for all graphs within a dataset. The star indicate the step with best validation loss. It is evident that the heights of star indicators vary across datasets, indicating our model learns distinct pooling layers for each. The fluctuating curve and its shadows during training confirm that our model effectively learns personalized pooling layers for every graph in a dataset.

3.6 Ablation Study

In this subsection, we conducted an ablation study to assess the contribution of each module that we described in Section 2.1. We use four datasets, including both graph and node levels. We evaluated these variants: for the graph information encoding module, we used GAT or GIN layers instead of GCN layers; we transformed the input graph to a complete graph to check the necessity of graph locality and our proposed graph parsing algorithm; we tested two simple aggregators without parameters, mean pool and max pool, instead of DeepSets for multiset computation. Table 4 shows that overall superior performance was obtained by these variants, except when we broke the locality with a complete graph, the model suffered a significant performance drop, highlighting the importance of graph locality and underscoring the crucial role of our parsing algorithm. The good performance on other variants indicates that our model is robust regarding the selection of GNN encoder and multiset function. Notably, GPN with a GAT backbone get better results on CiteSeer, which reveals the potential of our model.

<table>
<thead>
<tr>
<th>Component</th>
<th>PROTEINS</th>
<th>FRANKENSTEIN</th>
<th>Cora</th>
<th>CiteSeer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>75.62±0.74</td>
<td>70.45±0.53</td>
<td>88.07±0.87</td>
<td>77.02±1.65</td>
</tr>
<tr>
<td>w/GAT</td>
<td>75.41±0.91</td>
<td>69.94±2.56</td>
<td>87.46±1.13</td>
<td>77.16±1.50</td>
</tr>
<tr>
<td>w/GIN</td>
<td>75.02±1.25</td>
<td>70.18±0.44</td>
<td>86.62±1.26</td>
<td>76.50±1.67</td>
</tr>
<tr>
<td>w/Max pool</td>
<td>75.26±1.13</td>
<td>69.75±0.54</td>
<td>87.40±1.02</td>
<td>76.50±1.51</td>
</tr>
<tr>
<td>w/Mean pool</td>
<td>75.12±0.68</td>
<td>69.61±0.46</td>
<td>87.08±1.26</td>
<td>76.23±1.29</td>
</tr>
<tr>
<td>Complete graph</td>
<td>74.12±1.63</td>
<td>67.63±1.06</td>
<td>74.85±1.47</td>
<td>OOM</td>
</tr>
</tbody>
</table>

Table 4: Ablation study on three components of a graph pooling layer, covering both graph dataset and node dataset.

Figure 5: Memory efficiency test, “x” indicate the out-of-memory error.

Figure 6: Time efficiency test, “x” indicate the out-of-memory error.

Figure 7: Number of pooling layers visualization during model training.

In this subsection, we conducted an ablation study to assess the contribution of each module that we described in Section 2.1. We use four datasets, including both graph and node levels. We evaluated these variants: for the graph information encoding module, we used GAT or GIN layers instead of GCN layers; we transformed the input graph to a complete graph to check the necessity of graph locality and our proposed graph parsing algorithm; we tested two simple aggregators without parameters, mean pool and max pool, instead of DeepSets for multiset computation. Table 4 shows that overall superior performance was obtained by these variants, except when we broke the locality with a complete graph, the model suffered a significant performance drop, highlighting the importance of graph locality and underscoring the crucial role of our parsing algorithm. The good performance on other variants indicates that our model is robust regarding the selection of GNN encoder and multiset function. Notably, GPN with a GAT backbone get better results on CiteSeer, which reveals the potential of our model.

4 Conclusion

In this paper, we address two pressing challenges associated with hierarchical graph pooling: the inability to preserve node information while simultaneously maintaining a high degree of memory efficiency, and the lack of a mechanism for learning graph pooling structures for each individual graph. To address these challenges, we introduce a novel graph parsing algorithm. We demonstrate that our model outperforms existing state-of-the-art graph pooling methods in graph classification, while also delivering competitive performance in node classification. Additionally, a graph reconstruction task validates our model’s ability to preserve node information intact, while an efficiency study confirms the effectiveness of our solution in terms of both time and memory consumption. All the proofs, a more detailed introduction to related works, the limitations of our model and broader impacts can be found in Appendix due to the space limit.
5 Ethics Statement

In this work, we propose a novel graph pooling model. Our model is a general graph pooling model that does not make explicit assumptions about data distribution, thus avoiding the occurrence of (social) bias. However, our data-driven pooling mechanism needs to learn from samples with labels, and it is possible to be misled by targeting corrupted graph structures or features, which can have a negative social impact because both internet data and social networks can be represented as graphs. We can develop corresponding ethical guidelines and constraints for the usage of our model to avoid such problems.

6 Reproducibility Statement

We provide an introduction to our used datasets and model configuration in the experimental section of our paper. The dataset is mentioned first in this section. In our Appendix, we report the hardware and software environment used in our experiments, the methods used for hyper-parameter tuning, and the steps for downloading, loading, and pre-processing the datasets. In our released source code, we list the steps required to reproduce the results and provide yaml files that include all the hyper-parameters for the reported performance in this paper.

Acknowledgement

This work was sponsored by the National Key Research and Development Program of China (No. 2023ZD0121402) and National Natural Science Foundation of China (NSFC) grant (No.62106143).

References


Hao Yuan and Shuiwang Ji. Structpool: Structured graph pooling via conditional random fields. In

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and

Liang Zhang, Xudong Wang, Hongsheng Li, Guangming Zhu, Peiying Shen, Ping Li, Xiaoyuan Lu,

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning
architecture for graph classification. In Proceedings of the AAAI conference on artificial intelligence,

Zhen Zhang, Jiajun Bu, Martin Ester, Jianfeng Zhang, Chengwei Yao, Zhi Yu, and Can Wang.

Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gns. International

Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. Beyond
homophily in graph neural networks: Current limitations and effective designs. Advances in Neural
A Proofs

A.1 Proofs regarding Proposition

Proof. Since in the \( k \)-th graph pooling layer, the number of iterations for outer loop in Algorithm is \( n^{(k+1)} \), to prove the time complexity \( \mathcal{O}(\sum_{j=1}^{n^{(k)}} d_{ij}) \), we only need to prove the number of iterations in \( i \)-th inner loop is \( d_i \). Notice that in Definition the DOM \((\cdot)\) performs a edges neighborhood lookup on graph \( \hat{C}^{(k)} \), and this lookup operation can be effectively parallelized for all edges, thus the number of iterations in inner loop equals to the lookup steps needed to traverse all the nodes in that subgraph (a subgraph corresponding to node \( i \) in the output graph), which equals to the diameter \( d_i \) of the subgraph.

Since graph \( \hat{C}^{(k)} \) has \( n^{(k)} \) non-zero entries as we described in Definition the time complexity can never exceed it; then, suppose the input graph \( G^{(k)} \) is a pure set of isolated nodes, without any structure, then the number of iterations in Algorithm equals to \( n^{(k)} \), which is the worst case.

A.2 Proofs regarding Proposition

Proof. Since the commonly used operations like sum \((\cdot)\), argmax \((\cdot)\), union \((\cdot)\) in Algorithm are based on set and are permutation invariant, we only need to prove 3 operators we introduced are also permutation invariant: for DOM \((\cdot)\) operator, it first doing a row-wise max index, which is permutation invariant, then a masking is applied, since the masking is based on the idx from last step, thus it is also permutation invariant; for EXP \((\cdot)\) operator, it performs a neighborhood lookup on matrix in a parallel way (has no need to specify the order), thus it is also permutation invariant; for GEN \((\cdot)\) operator, it’s results has no relationship with ordering since it’s just a matrix filling process, thus it is also permutation invariant.

The above analysis are based on a set of distinct values in edge score matrix \( C^{(k)} \), if there are some duplicate values, the argmax \((\cdot)\) operator cannot guarantee uniqueness. Thus we need to assume the row-wise non-zero entries in \( C \) are distinct, when the assumption breaks, it means there are at least two nodes with common neighbors and their node features are exactly the same.

A.3 Proofs regarding time complexity comparison with SEP

Proof. Suppose we have \( h_{max} \) pooling layers in total, then the overall time complexity is \( \mathcal{O}(h_{max} \sum_{j=1}^{n^{(j)}} d_{ij}) \), \( d_{ij} \) indicate the diameter for subgraph \( i \) in \( j \)-th pooling layer. The worst case is that the input graph has no edge and get time complexity of \( \mathcal{O}(n^{(k)}) \). Thus we have overall time complexity \( \mathcal{O}(\sum_{j=1}^{h_{max}} \sum_{i=1}^{n^{(j)}} d_{ij}) \leq \mathcal{O}(n^{(j)}) \leq \mathcal{O}(h_{max}n^{(0)}) \). Compared to SEP \cite{Wu et al., 2022} that also apply heuristic algorithm, our parsing algorithm has a better time efficiency: SEP has a overall time complexity \( \mathcal{O}(2n^{(0)} + h_{max}(m^{(0)} \log n^{(0)} + n^{(0)})) \) \cite{Wu et al., 2022}, which is much larger than ours.

A.4 Proofs regarding Proposition

Proof. In order to prove the conclusion, we only need to demonstrate that any two nodes in the graph \( G^{(k)} = (V^{(k)}, \mathcal{E}^{(k)}) \), if there is a path connect them, then a path would exist between the corresponding clusters in the graph \( G^{(k+1)} = (V^{(k+1)}, \mathcal{E}^{(k+1)}) \).

Assuming nodes \( i, j \in V^{(k)} \), and there exists a path \( s \) between them: \( s = (i = v_0 - v_1 - \cdots - v_{N-1} - v_N = j) \), where \( v_l \in V^{(k)}, l = 0, \cdots, N \). Suppose the cluster assigned to node \( v_l \in V^{(k)} \) is \( p_{v_l} \in V^{(k+1)} \), now we examine the relationship between the clusters \( p_{v_l}, p_{v_{l+1}} \). If the edge \( (v_l, v_{l+1}) \in \mathcal{E}^{(k)} \) is a dominant edge, meaning \( \hat{C}^{(k)}_{v_l,v_{l+1}} > 0 \). It should be noted that the operator EXP \((\cdot)\) conducts a neighborhood-lookup operation on \( \hat{C}^{(k)} \), regardless of whether the node \( v_l \) or...
$v_{l+1}$ is traversed first in Stage 2 of the parsing algorithm, since these two nodes are adjacent in $\hat{C}^{(k)}$, they will both be assigned to the same cluster, we have $p_{v_l} = p_{v_{l+1}}$.

If the edge $(v_l, v_{l+1}) \in E^{(k)}$ is not a dominating edge, i.e., $\hat{C}^{(k)}_{v_l,v_{l+1}} = 0$, then the two nodes will be assigned to different clusters. According to Equation 2, $A^{(k+1)} = S^{(k)}A^{(k)}S^{(k)}$, then $A^{(k+1)}_{p_{v_l},p_{v_{l+1}}} = (S^{(k)T})_{p_{v_l}}A^{(k)}S^{(k)}_{p_{v_{l+1}}} = \sum_{r=1}^{n(k)} \sum_{t=1}^{n(k)} S^{(k)}_{r,p_{v_l}} A^{(k)}_{t,r} S^{(k)}_{r,p_{v_{l+1}}}$. Noting that $S^{(k)}_{v_{l},p_{v_l}} = 1$, $S^{(k)}_{v_{l+1},p_{v_{l+1}}} = 1$, when $t = v_l$, $r = v_{l+1}$, the corresponding summation term is 1, which means that at least one non-zero term exists among all the summed terms. Since all summation items are non-negative, it follows that $A^{(k+1)}_{p_{v_l},p_{v_{l+1}}} > 0$. We have $(p_{v_l}, p_{v_{l+1}}) \in E^{(k+1)}$.

In conclusion, $p_{v_l}$ and $p_{v_{l+1}}$ are either the same node or adjacent nodes. Therefore, the node sequence $(p_{v_0}, \cdots, p_{v_N})$ is a path on the graph $G^{(k+1)}$. This completes the proof. \qed

B Related Works

Graph pooling Researchers have shifted their focus to hierarchical pooling as flat pooling methods (Atwood & Towsley, 2016; Xu et al., 2019) face difficulties in capturing comprehensive structural information. Early node dropping methods (Gao & Ji, 2019; Lee et al., 2019) only retain nodes with higher computed scores. This explicit dropping risks the irreversible loss of important node information and can break the graph connectivity (Baek et al., 2021; Wu et al., 2022). To better preserve node information, InfoMax-based (Li et al., 2020), memory-based (Khasahmadi et al., 2020) and global prototype-based (Lee et al., 2022) techniques have been suggested; Zhang et al. (2019, 2020); Ranjan et al. (2020) improves graph connectivity by graph structure learning. These attempts alleviate the problem of node dropping, but at cost of additional computing burden. On the other hand, the node clustering approach calculates a soft assignment vector for each node. This method effectively retains the node’s information but can be memory-intensive due to its dense assignment matrix. Some works try to make this matrix sparser using various priors, such as penalty entropy in Ying et al. or Haar wavelet transforms in Wang et al. However, these penalty terms may hurt model training. Another drawback of node clustering methods is that they can easily converge to trivial solutions where each node has uniform soft assignments, due to the ignorance of locality. Several methods inject locality into GNN encoders, such as CRF-based (Yuan & Ji, 2020) or MinCut-based (Bianchi et al., 2020), but they are unable to scale to larger graphs. Unlike the previous mentioned works, Jo et al. (2021) focuses on edges as the center and emphasizes precise learning of both node and edge information. Finally, none of these approaches can optimize the pooling structure for each individual graph.

Structure Optimization Structure optimization on graphs has gained significant attention in recent times. The majority of these methods concentrate on the horizontal level, which involves learning graph structure through techniques such as structure learning (Jiang et al., 2019; Chen et al., 2020); Jin et al., 2020; Suresh et al., 2021) to determine node adjacency with or without a graph prior, or graph rewiring (Gasteiger et al., 2019; Topping et al., 2022; Bo et al., 2023) that adds or removes edges to enhance message passing. Only a few focus on the vertical level, which is concerned with learning pooling structures. EdgePool (Diehl, 2019) is an initial attempt to dynamically learn pooling trees by iteratively contracting edges based on calculated scores, unless one of their nodes has already been part of a contracted edge. Although they achieve end-to-end pooling tree learning through edge scores, their contraction-based strategy limits flexibility as it results in a near-half pooling ratio. On the other hand, SEP (Wu et al., 2022) uses combinatorial optimization to generate pooling trees by defining and optimizing a metric with a greedy algorithm. While this optimizes the pooling tree well for each graph, it is only used as a pre-processing step before training and cannot benefit from downstream supervision signals. Additionally, SEP relies solely on graph topology and fails to capture node features.

Grammar Induction Grammar induction, which aims at learning the latent structure behind natural language in an unsupervised way, can be dated back to Klein & Manning (2004). It has seen many advances recently, such as by using amortized variational inference on a transition-based parser Kim et al. (2019b), by dynamic programming Drozdov et al. (2019), or by maximizing the marginal likelihood of a sentence generated by PCFG Kim et al. (2019a), etc. Among all these various approaches, the most related is the works based on syntactic distance Shen et al. (2018b), which...
is a sequence of continuous scalar values that allows the tree to be inferred from it in a bottom-up fashion. This approach has been successfully shown to work for both recurrent models Shen et al. (2018a, 2019) and Transformer-based models Shen et al. (2021). Different from the above works that induce latent tree structures as a byproduct of some self-supervised training process, our approach lets the latent structure drive graph pooling and engage in gradient backpropagation, resulting in a more task-relevant model.

C DIFFERENCES COMPARED TO CLOSE METHODS

Our model significantly differs from EdgePool. In EdgePool, each cluster is usually composed of two adjacent nodes, and only when all neighbors of a certain node have been assigned, a cluster with a single node will be formed. Therefore, the proportion of pooling in the EdgePool is always close to 50%, which is not learnable. In contrast, in our model, the pooling ratio is not constrained and is entirely inferred by the parsing algorithm, allowing for end-to-end learning. Moreover, the number of pooling layers in EdgePool is set as a hyperparameter, and each pooling layer uses independent parameters. However, we recursively use a single pooling layer to construct our model, the number of which is inferred by the parsing algorithm. This varies continuously during the training process, with all pooling layers sharing the same parameters. Figure 8 visually illustrates the differences between three methods for optimizing pooling tree, i.e., SEP, EdgePool, and our proposed GPN.

D LIMITATION

In Section 3.4, our model demonstrates good time efficiency when processing batched graphs. However, if a batch contains graphs with varying numbers of pooling layers, the model may not be as time-efficient. This is because graphs with fewer pooling layers would have to wait for those with more iterations to finish before proceeding, potentially leading to inefficiencies when dealing with pretty large batches of graph data. For potential extreme cases that the inferred height of pooling tree is too large, we can set a maximum height limit for the pooling tree in the algorithm. This means the algorithm will stop during the parsing process once this height is reached.

E DISCUSSION

When running our model on a large-scale graph with a significant number of nodes, our graph parsing algorithm may infer a high pooling tree. In this case, the model will perform numerous graph convolutions and continuously smooth node features. The model structure for node classification task already includes residual connections from the encoder to the decoder, which helps alleviate potential over-smoothing issues by preserving local neighborhood information and original node features through residual connections.
F \textbf{DETAILS FOR EQUATION 4 AND END-TO-END TRAINING}\n
Here we describe the forward and backward passes of our model to clarify the training process: During the forward pass, we first use GNN and MLP to compute the edge scores (Equation 1). Then we run Algorithm 1 to obtain the assignment matrix (Equation 2) and performs the multiset computation to obtain the coarsened graph (Equation 3). The final step is to compute the edge score mask (Equation 4) to ensure the gradients can flow into GNN and MLP. During the backward pass, Algorithm 1 does not run. Instead, the gradient backpropagation is performed directly. The GNN and MLP can be updated through the gradients of the edge score mask.

We further present the details of Equation 4 to enhance understanding: Induce subgraph $\hat{G}^{(k)}_i \subseteq G^{(k)}$ from node $v_i$ means: each node $v_i$ in graph $G^{(k+1)}$ is a cluster generated by the $k$-th pooling layer, thus it corresponds to a connected subgraph $\hat{G}^{(k)}_i$ in $G^{(k)}$. In order to allow gradients to flow into the MLP used for calculating edge scores, we aggregate the scores of all the edges $\hat{E}^{(k)}_i$ in each subgraph $\hat{G}^{(k)}_i$. We then obtain a vector $y^{(k)}_i \in \mathbb{R}^{n_{(k+1)}}$ represents the score of clusters. This vector is then replicated into a mask matrix $y^{(k)}_i 1^T \in \mathbb{R}^{n_{(k+1)} \times d}$, which matches the shape of the node feature matrix $\hat{X}^{(k+1)}$. Finally, this mask is element-wise multiplied with the node feature matrix. During the backward pass, Algorithm 1 does not run. Instead, the gradient backpropagation is performed directly. The GNN and MLP can be updated through the gradients of the edge score mask.

G \textbf{BROADER IMPACTS}\n
In this study, we suggest a new graph pooling model that can address issues with current methods. Our model derives the pooling structure from both the topology of the graph and the node features, and is not founded on any assumptions regarding the data, which mitigates the risk of social bias. Nonetheless, since the algorithm used for data parsing is data-driven, it is possible that the pooling structure may be misguided by corrupted graph structure or features, which could lead to unfavorable social consequences. Since both internet data and social networks can be presented as graphs, we need to establish ethical guidelines and limit the usage of our model to avert such issues.

H \textbf{EXPERIMENTAL DETAILS}\n
We use NVIDIA GeForce RTX 3090 and NVIDIA GeForce RTX 4090 as the hardware environment, and use PyTorch and PyTorch Geometric \cite{Fey2019} as our software environment. The datasets are downloaded from PyTorch Geometric, we also use dataloader provided by PyTorch Geometric to load and pre-process datasets. We fix the random seed for reproducibility. We report detailed configurations as follows.

H.1 \textbf{GRAPH CLASSIFICATION}\n
\textbf{Dataset details} \hspace{1em} DD and PROTEINS are datasets of protein structures that represent macromolecules. Our task is to distinguish whether they are enzymes or non-enzymes based on their structure. The graph size of DD is larger than the other four datasets, resulting in the highest number of graph-pooling layers. NCI1 and NCI109 are two medium-sized datasets of chemical compounds labeled according to whether they are active against lung cancer cells and/or ovarian cancer cells. FRANKENSTEIN is a chimeric dataset of Mutagenicity and MNIST. It is relatively small and has the lowest average number of nodes among these five datasets, as reflected in its smallest number of learned graph-pooling layers. ogbg-molpcba is a chemical compound dataset used to determine molecular properties. It is a curated dataset from PubChem BioAssay (PCBA) that provides information on the biological activities of small molecules. The performance of ogbg-molpcba is measured using Average Precision, as this dataset is highly biased due to the small number of positive examples. We borrowed the baseline results from \cite{lee2019, ranjan2020, lee2022}. Except for MinCutPool, Hoscpool, GMT and SEP, we reran their code and report the performance, since they did not have public records under our setting. We followed PyTorch Geometric \cite{Fey2019} scripts to reimplement EdgePool for testing.
We keep the same hidden layer for all datasets and limit the maximum number of epochs to 2000. We use an early stopping strategy for all datasets. We also use validation loss as an early stopping criterion. We use the grid search method to find suitable hyperparameters. Table 5 gives the best hyperparameters used in the 5 graph classification datasets. And unlike the first three datasets, PROTEINS, NCI1109, FRANKENSTEIN, and ogbg-molpcba, the node features are 0/1-valued word vectors. CiteSeer, Cora and PubMed are all citation network datasets. They share a common feature in that their node features are 0/1-valued word vectors. CiteSeer consists of 3,312 scientific publications, which contain a total of six categories. It has a citation network consisting of 4,732 links. Its word vector has a dimension of 3,703. Cora consists of 2,708 scientific publications containing a total of seven categories. Its citation network consists of 5,429 links. Its word vector has a dimension of 1,433. PubMed consists of 19,717 scientific publications related to diabetes, which are divided into three different categories. Its citation network consists of 44,338 links. Its word vector has a dimension of 500. We obtain all baseline results from Zhu et al. (2020), Yan et al. (2022), and Bodnar et al. (2022), except for TopKPool and SEP, we rerun their code and perform a grid search on Cora, CiteSeer, and PubMed to ensure a fair comparison.

Implementation details For each experiment, we repeat the experiment on 20 seeds (from 0 to 19), and we use the 10-fold cross-validation method. So for each dataset we end up reporting the average result of 200 runs on the test set, which allows us to effectively avoid noise in the results and makes the results more credible. For all datasets, we use an early stopping strategy to avoid overfitting. We use validation loss as the criterion the early stopping. Also, the maximum number of epochs is set to 500 for all datasets. The Table 5 gives the best hyperparameters used in the 5 graph classification datasets. For the sake of brevity, we use abbreviations in the table. $L_{GNN}$ represents number of GCN layers inside the GNN block, $L_{l_{\text{DeepSets}}}$ represents number of MLP layers of DeepSets, $L_{l_{\text{MLP}}}$ represents number of layers of the MLP for edge score computation, Dropout represents dropout in the MLP, DropEdge represents the ratio of the edge dropout when performing parsing, lr represents learning rate, $H$ represents hidden channel, $B$ represents batch size.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Hyper-parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD</td>
<td>$L_{GNN}: 2$, $L_{l_{\text{DeepSets}}}: 1$, Dropout: 0.2, $lr$: 0.0001, $H$: 128, $B$: 64</td>
</tr>
<tr>
<td>PROTEINS</td>
<td>$L_{GNN}: 3$, $L_{l_{\text{DeepSets}}}: 1$, Dropout: 0.1, $lr$: 0.0005, $H$: 128, $B$: 128</td>
</tr>
<tr>
<td>NCI1</td>
<td>$L_{GNN}: 2$, $L_{l_{\text{DeepSets}}}: 3$, Dropout: 0.4, $lr$: 0.0005, $H$: 256, $B$: 256</td>
</tr>
<tr>
<td>NCI1109</td>
<td>$L_{GNN}: 3$, $L_{l_{\text{DeepSets}}}: 1$, Dropout: 0.2, $lr$: 0.0005, $H$: 128, $B$: 64</td>
</tr>
<tr>
<td>FRANKENSTEIN</td>
<td>$L_{GNN}: 3$, $L_{l_{\text{DeepSets}}}: 1$, Dropout: 0.2, $lr$: 0.001, $H$: 128, $B$: 128</td>
</tr>
<tr>
<td>ogbg-molpcba</td>
<td>$L_{GNN}: 3$, $L_{l_{\text{DeepSets}}}: 1$, Dropout: 0.3, $lr$: 0.001, $H$: 128, $B$: 128</td>
</tr>
</tbody>
</table>

Table 6: Best hyper-parameters for node classification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Hyper-parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cora</td>
<td>$L_{GNN_1}: 1$, $L_{GNN_2}: 2$, $L_{l_{\text{DeepSets}}}: 2$, $L_{l_{\text{MLP}}}: 1$, Dropout: 0.5, DropEdge: 0.5, $lr$: 0.005</td>
</tr>
<tr>
<td>CiteSeer</td>
<td>$L_{GNN_1}: 2$, $L_{GNN_2}: 3$, $L_{l_{\text{DeepSets}}}: 2$, $L_{l_{\text{MLP}}}: 3$, Dropout: 0.6, DropEdge: 0.1, $lr$: 0.005</td>
</tr>
<tr>
<td>PubMed</td>
<td>$L_{GNN_1}: 2$, $L_{GNN_2}: 2$, $L_{l_{\text{DeepSets}}}: 1$, $L_{l_{\text{MLP}}}: 1$, Dropout: 0.7, DropEdge: 0.7, $lr$: 0.001</td>
</tr>
<tr>
<td>Film</td>
<td>$L_{GNN_1}: 2$, $L_{GNN_2}: 2$, $L_{l_{\text{DeepSets}}}: 1$, $L_{l_{\text{MLP}}}: 2$, Dropout: 0.5, DropEdge: 0.5, $lr$: 0.001</td>
</tr>
</tbody>
</table>

Implementation details For the node classification task, we used a ten-fold cross-validation method. We keep the same hidden layer for all datasets and limit the maximum number of epochs to 2000. We use an early stopping strategy for all datasets. We also use validation loss as an early stopping criterion. We use the grid search method to find suitable hyperparameters. Table 6 gives the different hyperparameters used for the four node classification datasets. And unlike the first three datasets, Film is a dataset that describes the collaborative relationships between actors.

H.2 NODE CLASSIFICATION

Dataset details CiteSeer, Cora and PubMed are all citation network datasets. They share a common feature in that their node features are 0/1-valued word vectors. CiteSeer consists of 3,312 scientific publications, which contain a total of six categories. It has a citation network consisting of 4,732 links. Its word vector has a dimension of 3,703. Cora consists of 2,708 scientific publications containing a total of seven categories. Its citation network consists of 5,429 links. Its word vector has a dimension of 1,433. PubMed consists of 19,717 scientific publications related to diabetes, which are divided into three different categories. Its citation network consists of 44,338 links. Its word vector has a dimension of 500. We obtain all baseline results from Zhu et al. (2020), Yan et al. (2022), and Bodnar et al. (2022), except for TopKPool and SEP, we rerun their code and perform a grid search on Cora, CiteSeer, and PubMed to ensure a fair comparison.
Published as a conference paper at ICLR 2024

(a) TopKPool  (b) SAGPool  (c) ASAPPool
(d) DiffPool  (e) minCutPool  (f) GPN

Figure 9: Reconstruction results of ring synthetic graphs, compared to node drop and clustering methods.

H.3 GRAPH RECONSTRUCTION

Implementation details In order to keep the same setting with [Bianchi et al. (2020)], we use two message passing layers before the pooling operation and after the unpooling operation. We choose mean squared error (MSE) as the loss function. We use an early stopping strategy where patience is set to 1,000 epochs. The maximum number of epochs is set to 10,000. We then optimise the network using the Adam optimiser. In the experiment of Bianchi et al. (2020), they retained 25% of the nodes by adjusting the height of the tree. In contrast, our GPN model automatically retains around 30% of the nodes of the grid and ring during pooling.

More reconstruction results Figure 9 and Figure 10 show the results of graphs reconstructed using various pooling methods. We can visually see that the node-drop method suffers from information loss, resulting in the original graph being basically unrecognisable. Also, we can see that GPN preserves the shape of the ring as well as the coordinates of the interior of the mesh very well.
Published as a conference paper at ICLR 2024

Figure 10: Reconstruction results of grid synthetic graphs, compared to node drop and clustering methods.

Figure 11: Sentence parsing build a syntactic tree based on the scores between tokens; graph parsing build a pooling structure based on the scores defined on edges.