Boosting Graph Structure Learning with Dummy Nodes

Xin Liu [1] **Jiayang Cheng** [1] **Yangqiu Song** [1] **Xin Jiang** [2]

Abstract

With the development of graph kernels and graph representation learning, many superior methods have been proposed to handle scalability and oversmoothing issues on graph structure learning. However, most of those strategies are designed based on practical experience rather than theoretical analysis. In this paper, we use a particular dummy node connecting to all existing vertices without affecting original vertex and edge properties. We further prove that such the dummy node can help build an efficient monomorphic edge-to-vertex transform and an epimorphic inverse to recover the original graph back. It also indicates that adding dummy nodes can preserve local and global structures for better graph representation learning. We extend graph kernels and graph neural networks with dummy nodes and conduct experiments on graph classification and subgraph isomorphism matching tasks. Empirical results demonstrate that taking graphs with dummy nodes as input significantly boosts graph structure learning, and using their edge-to-vertex graphs can also achieve similar results. We also discuss the gain of expressive power from the dummy in neural networks.

1. Introduction

Graph structures have been widely used in modeling the interactions and connections in complex systems, such as biological networks, chemical molecules, and social networks. In fact, these data contain rich information in their graph structure beyond vertex and edge attributes. For example, atoms are held together by covalent bonds, and different molecular compounds (usually named isomers) with similar atoms may have distinct properties due to different structures. Thus, there has been a surge of interest in graph similarity, graph comparison, and subgraph matching. In recent years, numerous approaches have been proposed in machine learning and deep learning, among which algorithms based on graph kernels (GKs) (Borgwardt et al., 2005; Shervashidze et al., 2011; Morris et al., 2020) and graph neural networks (GNNs) (Kipf & Welling, 2017; Vashishth et al., 2020) are most notable. GKs tackle the graph comparison by exploring and capturing the semantics inherent in graph-structured data. The main idea behind graph kernels is that graphs with similar properties are highly likely to have similar distributions of substructures. However, most GKs focus on vertex-centric substructures while ignoring the edge similarities. Instead, inductive GNNs automatically extract higher-order information of graphs, sometimes leading to more powerful features compared to hand-crafted features used by GKs (Xu et al., 2019). Nevertheless, there are also disadvantages of GKs and GNNs. The runtime complexity of GKs when considering subgraphs of size up to $k \geq 2$ is usually $\Omega(k \cdot |\mathcal{V}|^2)$ at least and $\mathcal{O}(k \cdot |\mathcal{V}|^{k+1})$ at worst, where $|\mathcal{V}|$ is the number of vertices in the smaller graph for comparison (Kriege et al., 2020). The objective of GNNs is highly non-convex, requiring careful hyper-parameter tuning to stabilize the training procedure and avoid oversmoothing. Besides, message passing in GNNs still faces the limitation of the expressive power and the information vanish upon 0-outdegree vertices in deeper networks.

To address the aforementioned problems, many strategies have been designed. Morris et al. (2020) proposed the local variants of Weisfeiler-Lehman Subtree Kernels (k-WL) to vastly reduce the computation time without performance decline. k-IGNs (Maron et al., 2019), k-GNNs (Morris et al., 2019), and EASN (Bevilacqua et al., 2021) involve high-order tensors in representing high-order substructures with the expressive power as k-WL. Some kernels quantify the similarity based on random walks to reduce the complexity (Zhang et al., 2018). Li et al. (2018) and Rong et al. (2019) addressed the GNNs' oversmoothing by randomly removing edges from graphs to make GNNs robust to various structures. On the other hand, adding reversed edges in heterogeneous graphs is popular in practice (Vashishth et al., 2020; Hu et al., 2020). Some strategies are based on experimental experience rather than theoretical analysis.

This work was done when Xin Liu was an intern at Huawei Noah's Ark Lab. [1]Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong SAR, China [2]Huawei Noah's Ark Lab, Hong Kong SAR, China. Correspondence to: Xin Liu <xliucr@cse.ust.hk>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

To derive a theoretically guaranteed (sub)graph structure modeling that is consistent in original GKs computation and can improve GNNs, we use a particular dummy node and connect it with all existing vertices without affecting original vertex and edge properties. We start from the edge-to-vertex transform to theoretically analyze the role of the dummy in structure preserving. It turns out that an efficient monomorphic transform L_Φ to convert edges to vertices and an epimorphic inverse L_Φ^{-1} to recover the original graph back make the edge-to-vertex transform lossless. It is interesting to observe the transformed graph also contains another dummy node. We extend vertex-centric GKs and GNNs with dummy nodes to boost graph structure learning, with a linear computation cost to the number of edges.

Our main contributions are highlighted as follows:

1. We prove that adding a special dummy node with links to current existing vertices can help build an efficient and lossless edge-to-vertex transform, which also indicates that the edge information can be well preserved during learning.

2. We utilize dummy nodes and edges to extend state-of-the-art machine learning and deep learning models to improve their abilities to capture (sub)graph structures.

3. Extensive experiments are conducted on graph classification and subgraph isomorphism counting and matching, and empirical results reveal the success of learning with graphs with dummy nodes.

Code is publicly released at `https://github.com/HKUST-KnowComp/DummyNode4GraphLearning`.

2. Related Work

Before the deep learning era, graph kernels (GKs) dominated supervised graph structure learning through mapping graphs to Hilbert space and computing Gram matrices. Different kernel functions focus on specific structural properties of graphs. Shortest-path Kernel (Borgwardt & Kriegel, 2005) decomposes graphs into shortest paths and compares graphs according to their shortest paths, such as path lengths and endpoint labels. Instead, Graphlet Kernels (Shervashidze et al., 2009) compute the distribution of small subgraphs under the assumption that graphs with similar graphlet distributions are highly likely to be similar. Another important kernel family considers subtrees. One state-of-the-art kernel is Weisfeiler-Lehman Subtree Kernel (k-WL) (Shervashidze et al., 2011), and some higher-order variants (Morris et al., 2017) and local variants (Morris et al., 2020) further strengthen the expressive power. However, graph kernels are limited by non-inductive learning and the super quadratic time complexity to the training data size.

With the rapid development of heterogeneous computing, neural networks have attracted attention recently. Researchers have successfully used relational inductive biases within deep learning architectures to build graph neural networks (GNNs). End-to-end learning relieves the burden of feature engineering and makes the structure learning sophisticated and flexible (Gilmer et al., 2017; Battaglia et al., 2018). Ideas behind kernels are still referable for the design of GNNs (Morris et al., 2019). k-GNNs (Morris et al., 2019) and EASN (Bevilacqua et al., 2021) align the k-WL hierarchy to enhance the expressive power of GNNs. However, this involves high-order tensor computations. How to efficiently boost the graph structure learning is also one of the research frontiers. Ranking neighbors based on attention scores enables explicit weights for information aggregation (Yuan & Ji, 2021). Besides, it also boosts learning to consider multihop neighbors (Zhu et al., 2020; Teru et al., 2020). On the other hand, dynamic high-order neighbor selection (Yang et al., 2021) and dynamic pointer links (Velickovic et al., 2020) illustrate the power of data-driven manipulations. Moreover, differentiable pooling yields consistent and significant performance improvement for end-to-end hierarchical graph representation learning (Ying et al., 2018; Zhang et al., 2019). Li et al. (2018) and Rong et al. (2019) addressed the GNNs' oversmoothing by randomly removing edges from graphs to make GNNs robust to various structures. However, most of these strategies are based on practical experience rather than theoretical analysis.

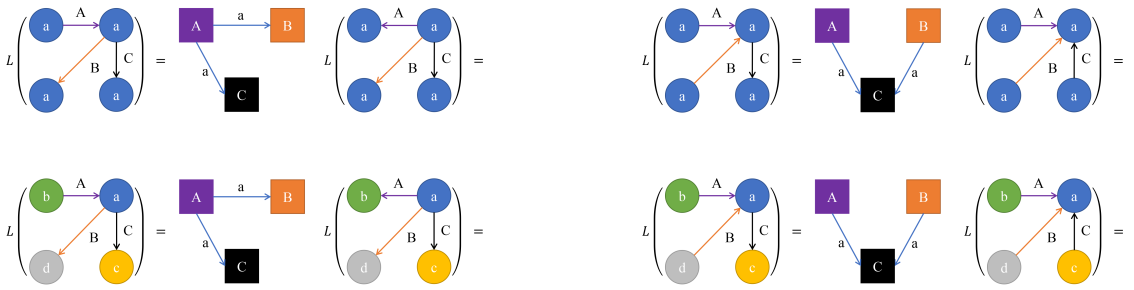
Some literature suggests utilizing "dummy" super-nodes to explicitly learn subgraphs (Scarselli et al., 2009; Hamilton et al., 2017a), but the node is served as a special readout to conduct the representation of the target subgraph. On the contrary, we directly add a dummy node as a part of the target graph to learn representations and capture similarities.

3. Lossless Edge-to-vertex Transforms

3.1. Preliminary

Let $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G}, \mathcal{X}_\mathcal{G}, \mathcal{Y}_\mathcal{G})$ be a *directed connected heterogeneous multigraph* with a vertex set $\mathcal{V}_\mathcal{G}$, an edge set $\mathcal{E}_\mathcal{G} \subseteq \mathcal{V}_\mathcal{G} \times \mathcal{V}_\mathcal{G}$, a label function $\mathcal{X}_\mathcal{G}$ that maps a vertex to a set of *vertex labels*, and a label function $\mathcal{Y}_\mathcal{G}$ that maps an edge to a set of *edge labels*. Under this definition, multiple edges with the same source and the same target can be merged by extending $\mathcal{Y}$, making $\mathcal{G}$ without multi-edges for clarity. To simplify the statement, we also let $\mathcal{Y}_\mathcal{G}((u,v)) = \phi$ if $u, v \in \mathcal{V}_\mathcal{G}$ but $(u,v) \notin \mathcal{E}_\mathcal{G}$. We use d_v^- and d_v^+ to denote the indegree and outdegree of vertex v.

In graph theory, the edge-to-vertex transform converts a graph to its line graph where original vertex properties are stored in edges, and original edge properties are stored in vertices in the transformed graph (Harary, 1969).



(a) one vertex with 0 indegree, two vertices with 0 outdegree

(b) one vertex with 0 indegree, three vertices with 0 outdegree

(c) two vertices with 0 indegree, one vertex with 0 outdegree

(d) three vertices with 0 indegree, one vertex with 0 outdegree

Figure 1. Examples of directed 3-claws and line graphs, where lowercased letters correspond to vertex labels of the original graphs (and edge labels of the line graphs), capitalized letters correspond to edge labels of the original graphs (and vertex labels of the line graphs). Note that line graphs in (b) and (d) are empty.

Definition 3.1 (Edge-to-vertex transform)**.** A *line graph* (also known as *edge-to-vertex graph*) $\mathcal{H}$ of a graph $\mathcal{G}$ is obtained by a bijection $g : \mathcal{E}_\mathcal{G} \to \mathcal{V}_\mathcal{H}$, where g associates a vertex $v' \in \mathcal{V}_\mathcal{H}$ with each edge $e = g^{-1}(v') \in \mathcal{E}_\mathcal{G}$. And two vertices $u', v' \in \mathcal{V}_\mathcal{H}$ are connected as (u', v') if and only if the destination of $d = g^{-1}(u')$ is the source of $e = g^{-1}(v')$. Formally, we have:

- $\forall e = (u, v) \in \mathcal{E}_\mathcal{G}, \mathcal{Y}_\mathcal{G}(e) = \mathcal{X}_\mathcal{H}(g(e))$,
- $\forall v' \in \mathcal{V}_\mathcal{H}, \mathcal{X}_\mathcal{H}(v') = \mathcal{Y}_\mathcal{G}(g^{-1}(v'))$,
- $\forall d, e \in \mathcal{E}_\mathcal{G}, u' = g(d) \in \mathcal{V}_\mathcal{H}, v' = g(e) \in \mathcal{V}_\mathcal{H}$, $(d.\text{target} = e.\text{source} = v) \to (\mathcal{Y}_\mathcal{H}((u', v')) = \mathcal{X}_\mathcal{G}(v)))$,
- $\forall e' = (u', v') \in \mathcal{E}_\mathcal{H}, d = g^{-1}(u') \in \mathcal{V}_\mathcal{G}, e = g^{-1}(v') \in \mathcal{V}_\mathcal{G}$, $(d.\text{target} = e.\text{source}) \wedge (\mathcal{Y}_\mathcal{H}(e') = \mathcal{X}_\mathcal{H}(d.\text{target}))$.

We write $\mathcal{H}$ as $L(\mathcal{G})$ where $L : \mathcal{G} \to \mathcal{H}$ refers to the *edge-to-vertex transform*. Based on the definition, the number of vertices of $\mathcal{H}$ is the same as the number of edges of $\mathcal{G}$, and the number of edges of $\mathcal{H}$ equals to $\sum_{v \in \mathcal{V}_\mathcal{G}} d_v^- \cdot d_v^+$. But there are two worst cases: empty or complete line graphs.

3.2. Non-injective Edge-to-vertex Transforms

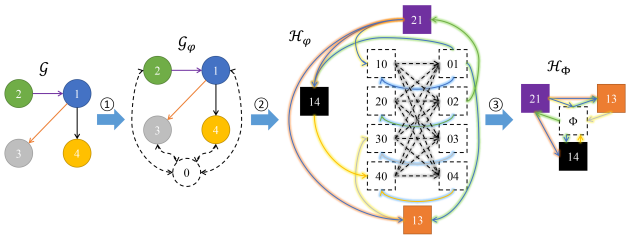
As the vertices of the line graph $\mathcal{H}$ corresponds to the edges of the original graph $\mathcal{G}$, some properties of $\mathcal{G}$ that depend only on adjacency between edges may be preserved as equivalent properties in $\mathcal{H}$ that depend on adjacency between vertices. Intuitively, we have one question of whether we can transform the line graph back to its original graph by a function $L^{-1} : \mathcal{H} \to \mathcal{G}$. The answer is no, because some information may be lost. The reason behind is that some vertices in the original graph are located in claw structures. One classic case is a 3-claw structure and a triangle having the same line graph in undirected scenarios. In fact, directed cases are more general: an extreme case is that all directed 2-path structures (i.e., two vertices are connected by one edge) with the same edge label have the same line graph. Two claws with similar structures but different labels may also result in the same line graph. Figure 1 demonstrates the 3-claw examples, and more complicated structures can be enumerated by extending these claws. All these instances have one thing in common: the information from those vertices with 0 indegree or 0 outdegree gets lost during the edge-to-vertex transform. It is easy to get the Lemma 3.2 from Definition 3.1.

Lemma 3.2. *During the edge-to-vertex transform over a directed graph $\mathcal{G}$, the information of a vertex v is preserved if and only if its indegree d_v^- is nonzero and its outdegree d_v^+ is nonzero. In particular, there are $d_v^- \cdot d_v^+$ copies in the line graph $\mathcal{H} = L(\mathcal{G})$.*

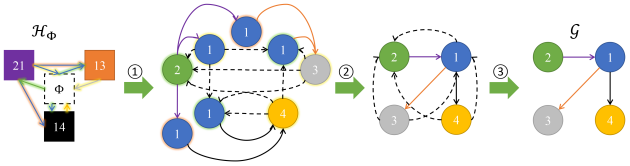
3.3. Injective and Inversive Edge-to-vertex Transforms

Liu & Song (2022) addressed the lost of information by introducing reversed edges with specific edge labels. With the help of reversed edges, all (non-isolated) vertices have nonzero indegrees and nonzero outdegrees. However, this strategy makes the graph and corresponding line graph extremely dense. Assume a graph $\mathcal{G}$ and its line graph $\mathcal{H} = L(\mathcal{G})$, then the modified graph doubles the number of edges, and the corresponding line graph has $2|\mathcal{V}_\mathcal{H}|$ vertices and $\sum_{v \in \mathcal{V}_\mathcal{G}} (d_v^- + d_v^+)^2 \geq 4|\mathcal{E}_\mathcal{H}|$ edges. As a result, the line graph cannot be used directly in practice. To say the least, it doubles the computation of graph convolutions and quadruples that of line graph convolutions. Therefore, we propose our efficient solution to eliminate the 0-indegree vertices and 0-outdegree vertices by adding dummy edges starting from and sinking to one particular dummy node.

Corollary 3.3. *Given a directed graph $\mathcal{G}$ with n vertices, the modified graph $\mathcal{G}_\varphi$ involves one dummy node φ and $2n$ dummy edges where this special dummy node connects every $v \in \mathcal{V}_\mathcal{G}$ by two dummy edges (φ, v) and (v, φ). During the edge-to-vertex transform L over $\mathcal{G}_\varphi$, the information of each vertex $v \in \mathcal{G}$ is preserved. In particular, there are $(d_v^- + 1) \cdot (d_v^+ + 1)$ copies in the line graph $\mathcal{H}_\varphi = L(\mathcal{G}_\varphi)$.*



(a) edge-to-vertex transform L_Φ



(b) inverse edge-to-vertex transform L_Φ^{-1}

Figure 2. Our proposed edge-to-vertex transforms L_Φ over 3-claws and the inverse transform L_Φ^{-1}, where numbers located in circles indicates the original vertex ids, numbers located in squares indicates the pairs of original source id and original target id.

According to Definition 3.1, we have the statistics of the line graph $\mathcal{H}_\varphi$:

$$|\mathcal{V}_{\mathcal{H}_\varphi}| = m + 2n = |\mathcal{V}_{\mathcal{H}}| + 2n, \tag{1}$$

$$\begin{aligned}|\mathcal{E}_{\mathcal{H}_\varphi}| &= \sum_{v\in\mathcal{V}_{\mathcal{G}'}} d_v^- \cdot d_v^+ \\ &= n^2 + \sum_{v\in\mathcal{V}_{\mathcal{G}}} (d_v^- + 1)\cdot(d_v^+ + 1) \\ &= \sum_{v\in\mathcal{V}_{\mathcal{G}}} d_v^- \cdot d_v^+ + n^2 + n + m + m \\ &= |\mathcal{E}_{\mathcal{H}}| + n^2 + n + m + m. \end{aligned} \tag{2}$$

As Corollary 3.3, all original vertices are stored in the line graph. But the costs are not negligible. The number of vertices in the line graph obviously increases by $2n$. But the worst thing is dramatical growth of the number of edges in an additional complexity $\mathcal{O}(n^2)$. To investigate further, we transform a directed 3-claw for demonstration. As shown in Figure 2a, the step ② is the conventional edge-to-vertex transform over $\mathcal{G}_\varphi$. The edges in its line graph $\mathcal{H}_\varphi$ can be divided into five categories:

- $|\mathcal{E}_{\mathcal{H}}|$ edges are connections through the original n vertices, which can be preserved by the transform without the help of dummy node;
- n^2 edges are connections between dummy edges through the dummy node;
- n edges are connections between dummy edges through the original n vertices;
- m edges start from the original edges and sink to the dummy edges;
- m edges start from the dummy and sink to the original.

In fact, the n^2 edges plotted as dashed grey lines are useless in original structure preserving since these edges does not store any graph-specific properties. On the contrary, each of the n edges marked in blue maintains original vertex properties. However, Corollary 3.3 confirms that there are $(d_v^- + 1)\cdot(d_v^+ + 1) = (d_v^- \cdot d_v^+ + d_v^- + d_v^+) + 1$ copies for an original vertex $v \in \mathcal{V}_\mathcal{G}$. Since $d_v^- \cdot d_v^+ + d_v^- + d_v^+ > 0$ holds for connected components except an isolated point, it is also safe to remove these n edges when $|\mathcal{E}_\mathcal{G}| = m > 0$.

After deleting the $n^2 + n$ edges, we find there is no connections between dummy edges anymore. Thus, we merge the corresponding $2n$ vertices as one dummy Φ and finally get the new transformed graph $\mathcal{H}_\Phi$ with

$$|\mathcal{V}_{\mathcal{H}_\Phi}| = m + 1 = |\mathcal{V}_{\mathcal{H}}| + 1, \tag{3}$$

$$|\mathcal{E}_{\mathcal{H}_\Phi}| = \sum_{v\in\mathcal{V}_\mathcal{G}} (d_v^- \cdot d_v^+ + d_v^- + d_v^+) = |\mathcal{E}_\mathcal{H}| + 2m, \tag{4}$$

where $\mathcal{H}$ is the line graph of the original graph $\mathcal{G}$. The new edge-to-vertex transform L_Φ is shown in Figure 2a. And we provide Algorithm 1 in Appendix A for details.

Since no vertex information or edge information gets lost, the next is to find the inverse transform L_Φ^{-1}. Because both G_φ and H_Φ contain a special dummy node, respectively, we consider the same strategy to merge vertices. Before that, original vertex ids are assigned as edge ids for $\mathcal{H}_\Phi$. We are surprised to find that it is easy to get vertices and edges of $\mathcal{G}$ from $L(\mathcal{H}_\Phi)$ after removing dummy edges. The process is shown in Figure 2b, and the algorithm is described in Algorithm 2 in Appendix A. Theorem 3.4 shows that the *inverse* of L_Φ always exists.

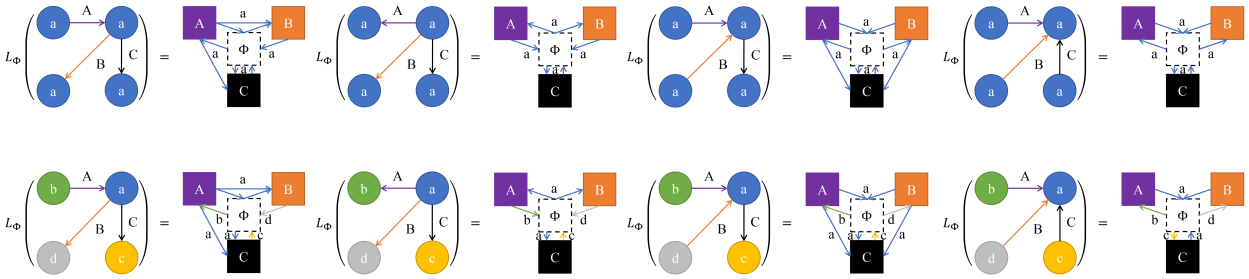
Theorem 3.4. *For any $\mathcal{H}_\Phi$ transformed by L_Φ such that $\mathcal{H}_\Phi = L_\Phi(\mathcal{G})$, L_Φ^{-1} can always transform $\mathcal{H}_\Phi$ back to $\mathcal{G}$, i.e., $L_\Phi^{-1}(L_\Phi(\mathcal{G})) = \mathcal{G}$.*

Proof. Let $\mathcal{G}'$ be $L_\Phi^{-1}(L_\Phi(\mathcal{G}))$, $\mathcal{G}'_{①}$ be the resulting graph of the step ① (i $\in \{1, 2, 3\}$) of L_Φ^{-1}.

Each vertex w of $\mathcal{H}_\Phi$ except the dummy Φ corresponds to one edge (e.g., $e = (u, v)$) of $\mathcal{G}$, and Φ connects w with two edges where (Φ, w) is associated with label $\mathcal{X}_\mathcal{G}(u)$ and (w, Φ) is associated with label $\mathcal{X}_\mathcal{G}(v)$. After the step ①, (Φ, w) is transformed as one vertex with id u and label $\mathcal{X}_\mathcal{G}(u)$, and (w, Φ) is converted as one vertex with id v and label $\mathcal{X}_\mathcal{G}(v)$. Similarly, other edges not connecting Φ can also be transformed as vertices. We have:

$$|\mathcal{V}_{\mathcal{G}'_①}| = \sum_{v\in\mathcal{V}_\mathcal{G}} (d_v^- \cdot d_v^+ + d_v^- + d_v^+),$$

where vertex v in $\mathcal{G}$ appears $d_v^- \cdot d_v^+ + d_v^- + d_v^+$ times (that is nonzero) in $\mathcal{G}'_{①}$. On the other hand, the indegree and the outdegree of w are $d_u^- + 1$ and $d_v^+ + 1$, respectively. Thus, we get:



(a) one vertex with 0 indegree, two vertices with 0 outdegree (b) one vertex with 0 indegree, three vertices with 0 outdegree (c) two vertices with 0 indegree, one vertex with 0 outdegree (d) three vertices with 0 indegree, one vertex with 0 outdegree

Figure 3. Examples of directed 3-claws and new transformed edge-to-vertex graphs by L_Φ.

$$
\begin{aligned}
|\mathcal{E}_{\mathcal{G}'_{①}}| &= \sum_{w\in\mathcal{H}_\Phi} d_w^- \cdot d_w^+ \\
&= m^2 + \sum_{(u,v)\in\mathcal{E}_\mathcal{G}} (d_u^- + 1)\cdot(d_v^+ + 1) \\
&= 2\sum_{v\in\mathcal{V}_\mathcal{G}} d_v^- \cdot d_v^+ + \sum_{(u,v)\in\mathcal{E}_\mathcal{G}} d_u^- \cdot d_v^+ + m^2 + m,
\end{aligned}
$$

where $2\sum_{v\in\mathcal{V}_\mathcal{G}} d_v^- \cdot d_v^+$ says inedge $(?, v)$ is copied d_v^+ times thanks to (w, Φ) and outedge $(v, ?)$ is copied d_v^+ times due to (Φ, w), $\sum_{(u,v)\in\mathcal{E}_\mathcal{G}} d_u^- \cdot d_v^+$ refers to the transformed edges by $L(L(\mathcal{G}))$, m^2 corresponds to dummy edges, m indicates original edges recovered by (Φ, w) and (w, Φ) directly.

As we store vertex ids and vertex labels from $\mathcal{G}$ in edges of $\mathcal{H}_\Phi$, we get $\mathcal{V}_{\mathcal{G}'_{②}}$ is exactly the same as $\mathcal{V}_\mathcal{G}$ after merging vertices with same information. Since the step ③ does not remove vertices further, for $\mathcal{V}_{\mathcal{G}'} = \mathcal{V}_\mathcal{G}$ holds.

Edges are merged based on source ids and target ids in the step ②, and additional dummy edges are removed in the step ③. Therefore, $\mathcal{E}_\mathcal{G} \subseteq \mathcal{E}_{\mathcal{G}'} \subseteq \mathcal{E}_{\mathcal{G}'_{①}}$. Next, we prove $\mathcal{E}_\mathcal{G} = \mathcal{E}_{\mathcal{G}'}$ by $\forall(u'', v'') \in \mathcal{E}_{L(L(\mathcal{G}))}, \mathcal{Y}_{L(L(\mathcal{G}))}((u'', v'')) \subseteq \mathcal{Y}_\mathcal{G}((u''.id, v''.id))$. Beineke (1968) and Beineke & Zamfirescu (1982) discussed $L(L(\mathcal{G}))$ in more detail. We simply prove in another direction. Let $\mathcal{S}_\mathcal{G} = \{u' \in \mathcal{V}_\mathcal{G} | d_{u'}^- = 0\}$, $\mathcal{T}_\mathcal{G} = \{v' \in \mathcal{V}_\mathcal{G} | d_{v'}^+ = 0\}$, and $\mathcal{U}_\mathcal{G} = \{v | d_v^- \cdot d_v^+ > 0\} = \mathcal{V}_\mathcal{G} - \mathcal{S}_\mathcal{G} \cup \mathcal{T}_\mathcal{G}$ then any edge e in $\mathcal{S}_\mathcal{G} \times \mathcal{T}_\mathcal{G}$ corresponds to a isolated vertex in $L(\mathcal{G})$ so that it gets lost in $L(L(\mathcal{G}))$, i.e., $\phi \subseteq \mathcal{Y}_\mathcal{G}(e)$. For other vertices $\mathcal{U}_\mathcal{G} = \{v | d_v^- + d_v^+ > 0\}$, any edge $e = (u, v) \in \mathcal{U}_\mathcal{G} \times \mathcal{U}_\mathcal{G} \cap \mathcal{E}_\mathcal{G}$ is located in at least one *3-diwalk*, e.g., $s \to u \to v \to t$ where $s, t \in \mathcal{V}_\mathcal{G}$. Then $L(\mathcal{G})$ converts this 3-diwalk to a *2-diwalk* $su' \to uv' \to vt'$, and $L(L(\mathcal{G}))$ finally results in a *1-diwalk* $u'' \to v''$ where u'' corresponds to u and v'' corresponds to v, i.e., $\mathcal{Y}_{L(L(\mathcal{G}))}((u'', v'')) = \mathcal{Y}_\mathcal{G}((u, v))$. Hence, $\forall(u'', v'') \in \mathcal{E}_{L(L(\mathcal{G}))}, \mathcal{Y}_{L(L(\mathcal{G}))}((u'', v'')) \subseteq \mathcal{Y}_\mathcal{G}((u''.id, v''.id))$. That is to say, both $2\sum_{v\in\mathcal{V}_\mathcal{G}} d_v^- \cdot d_v^+$ edges and $\sum_{(u,v)\in\mathcal{E}_\mathcal{G}} d_u^- \cdot d_v^+$ edges are going to be merged into m edges in the step ② and $|\mathcal{E}_{\mathcal{G}'}| = m$. Considering $\mathcal{E}_\mathcal{G} \subseteq \mathcal{E}_{\mathcal{G}'}$ and $m = \mathcal{E}_\mathcal{G}$, we have $\mathcal{E}_{\mathcal{G}'} = \mathcal{E}_\mathcal{G}$. Clearly, $\mathcal{G}' = \mathcal{G}$. And this finishes the proof. □

Considering the symmetry between $\mathcal{G}_\varphi$ and $\mathcal{H}_\Phi$ where the dummy node serves as a hub to connect all other vertices, we call $\mathcal{G}_\varphi$ and $\mathcal{H}_\Phi$ *conjugate* of each other.

Figure 3 illustrates the new transformed edge-to-vertex graphs over 3-claws. As observed, all vertex information is preserved in new transformed graphs, the union of line graphs and six heterogeneous edges connecting dummy Φ.

3.4. Transforms and Morphisms

As L_Φ is a structure-preserving map, we are also interested in the connections between L_Φ and morphisms. In graph theory, one of the most important bijective morphisms is the isomorphism.

Definition 3.5 (Isomorphism)**.** A graph $\mathcal{G}_1$ is *isomorphic* to a graph $\mathcal{G}_2$ if there is a bijection $f : \mathcal{V}_{\mathcal{G}_1} \to \mathcal{V}_{\mathcal{G}_2}$ such that:

- $\forall v \in \mathcal{V}_{\mathcal{G}_1}, \mathcal{X}_{\mathcal{G}_1}(v) = \mathcal{X}_{\mathcal{G}_2}(f(v))$,
- $\forall v' \in \mathcal{V}_{\mathcal{G}_2}, \mathcal{X}_{\mathcal{G}_2}(v') = \mathcal{X}_{\mathcal{G}_1}(f^{-1}(v'))$,
- $\forall (u, v) \in \mathcal{E}_{\mathcal{G}_1}, \mathcal{Y}_{\mathcal{G}_1}((u, v)) = \mathcal{Y}_{\mathcal{G}_2}((f(u), f(v)))$,
- $\forall (u', v') \in \mathcal{E}_{\mathcal{G}_2}, \mathcal{Y}_{\mathcal{G}_2}((u', v')) = \mathcal{Y}_{\mathcal{G}_1}((f^{-1}(u'), f^{-1}(v')))$.

We write $\mathcal{G}_1 \simeq \mathcal{G}_2$ for such the isomorphic property, and F is named as an *isomorphism*. For two isomorphic graphs, they are also regarded as both *permutation*.

L_Φ and L_Φ^{-1} are also morphisms from graphs to graphs. In particular, L_Φ^{-1} is an *epimorphism* based on Theorem 3.4. The following Propositions 3.6 shows that L_Φ is a *monomorphism*.

Proposition 3.6. *L_Φ is a monomorphism.*

Proof. To prove that L_Φ is a monomorphism, we seek to show $L_\Phi(\mathcal{G}_1) = L_\Phi(\mathcal{G}_2) \to \mathcal{G}_1 = \mathcal{G}_2$ for any $\mathcal{G}_1, \mathcal{G}_2$. Theorem 3.4 shows that L_Φ^{-1} always exists, and $L_\Phi^{-1}(L_\Phi(\mathcal{G}_1)) =$

$\mathcal{G}_1$, $L_\Phi^{-1}(L_\Phi(\mathcal{G}_2)) = \mathcal{G}_2$ always holds. Given $L_\Phi(\mathcal{G}_1) = L_\Phi(\mathcal{G}_2)$, we have $L_\Phi^{-1}(L_\Phi(\mathcal{G}_1)) = L_\Phi^{-1}(L_\Phi(\mathcal{G}_2))$, which implies $\mathcal{G}_1 = \mathcal{G}_2$. And this finishes the proof. □

Propositions 3.6 is important because we can apply the edge-to-vertex transform L_Φ before other morphisms and functions without breaking properties, such as Corollary 3.7.

Corollary 3.7. *Isomorphisms hold after L_Φ.*

More generally, permutation-invariant functions are expected to acquire the same outputs among permutations. In view of Corollary 3.7, we get the following consequence:

Corollary 3.8. *If a function h is permutation-invariant, then $h \circ L_\Phi$ is also permutation-invariant.*

This helps us to apply vertex-centric graph kernel functions and graph neural networks to learn edge-centric representations. And all above indicate that the graph with a dummy node $\mathcal{G}_\varphi$ is better to learn structure information than $\mathcal{G}$.

4. Methodology

In this section, we extend effective machine learning kernel functions and deep graph neural networks with dummy nodes and our proposed edge-to-vertex transform L_Φ.

4.1. Extensions of Graph Kernel Functions

A graph kernel (GK) is a symmetric, positive semi-definite function defined on the graph space. It is usually expressed as an inner product in Hilbert space (Kriege et al., 2020) such that $k(\mathcal{G}_1, \mathcal{G}_2) = \langle h(\mathcal{G}_1), h(\mathcal{G}_2)\rangle$, where k is the kernel function and h is the permutation-invariant function from graph space to Hilbert space. In general, k measures the similarity between two graphs, and the graph similarity is directly related to graph comparison in machine learning. In this paper, we aim to explore the power of dummy nodes, so we adopt graph-structure-sensitive and attribute-sensitive kernels, like Weisfeiler-Lehman Subtree Kernel (WL) (Shervashidze et al., 2011). We generalize these kernels with dummy nodes (denoted as k_φ) and the edge-to-vertex transform (denoted as k_Φ):

$$\begin{aligned} k_\varphi(\mathcal{G}_1, \mathcal{G}_2) &= k(\mathcal{G}_1, \mathcal{G}_2) + k(\mathcal{G}_{\varphi_1}, \mathcal{G}_{\varphi_2}) \\ &= \langle h(\mathcal{G}_1), h(\mathcal{G}_2)\rangle + \langle h(\mathcal{G}_{\varphi_1}), h(\mathcal{G}_{\varphi_2})\rangle, \quad (5) \\ k_\Phi(\mathcal{G}_1, \mathcal{G}_2) &= k(\mathcal{G}_1, \mathcal{G}_2) + k(\mathcal{H}_{\Phi_1}, \mathcal{H}_{\Phi_2}) \\ &= \langle h(\mathcal{G}_1), h(\mathcal{G}_2)\rangle + \langle h(L_\Phi(\mathcal{G}_1)), h(L_\Phi(\mathcal{G}_2))\rangle, \quad (6) \end{aligned}$$

where $\mathcal{G}_{\varphi_1}$ and $\mathcal{G}_{\varphi_2}$ respectively correspond to graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with a dummy node φ and dummy edges, and $\mathcal{H}_{\Phi_1} = L_\Phi(\mathcal{G}_1)$ and $\mathcal{H}_{\Phi_2} = L_\Phi(\mathcal{G}_2)$ are transformed by the proposed L_Φ, each of which contains a dummy node Φ. We add the $k(\mathcal{G}_1, \mathcal{G}_2)$ term in k_φ and k_Φ to enforce the kernel functions to pay more attention to original structures; otherwise, the dummy node and dummy edges may bring some side effects.

4.2. Extensions of Graph Neural Networks

Many graph neural networks (GNNs) have been proposed to learn graph structures, such as GCN (Kipf & Welling, 2017), GraphSAGE (Hamilton et al., 2017b), GIN (Xu et al., 2019). Most can be unified in the *Message Passing* framework (Gilmer et al., 2017):

$$\begin{aligned} \boldsymbol{\Delta}_v^{(t+1)} &= Aggregate(\{Message(\boldsymbol{x}_v^{(t)}, \boldsymbol{x}_u^{(t)}, \boldsymbol{y}_{(u,v)}) | u \in \mathcal{N}_v\}), \\ \boldsymbol{x}_v^{(t+1)} &= Update(\boldsymbol{x}_v, \boldsymbol{\Delta}_v^{(t+1)}), \end{aligned}$$

where $\boldsymbol{x}_v^{(t)}$ is the hidden state of vertex v at the t-th layer network, $\mathcal{N}_v$ is v's neighbor collection, $\boldsymbol{y}_{(u,v)}$ is the edge tensor for (u, v), $\boldsymbol{\Delta}_v^{(t+1)}$ is the aggregated message from neighbors, $Aggregate$ is a permutation-invariant functions (e.g., Sum) to aggregate all messages as one, and $Update$ is a combination function (e.g., Add) to fuse $\Delta_v^{(t+1)}$ and $\boldsymbol{x}_v^{(t)}$ as the new state $\boldsymbol{x}_v^{(t+1)}$. After introducing a dummy node to $\mathcal{G}$, the neighbor collection is extended with a dummy node φ, and the dummy node serves to aggregate the global graph information in turn. Similarly, we feed $L_\Phi(\mathcal{G})$ as the input to learn the graph representation, where such neural networks actually model the edges in $\mathcal{G}$.

4.3. Efficiency of Extensions

4.3.1. Learning with $\mathcal{G}_\varphi$

Even though we introduce one particular dummy node and $2n$ dummy edges to $\mathcal{G}$ (where n is the vertex size of $\mathcal{G}$), kernels' complexities when the size respect graphlets or tuples is not too large, denoted as k in Sec. 1. GNNs yield additional computation of the $2n$ dummy edges, but it is still efficient because we do not involve additional operations upon existing n vertices and $2n$ is usually much less than the existing m edges.

4.3.2. Learning with $\mathcal{H}_\Phi$

The overhead of utilizing $\mathcal{H}_\Phi$ include the cost to obtain and the cost to use. Sec. 3.3 analyzes the former part: $\mathcal{H}$ is constructed in $\sum_{v \in \mathcal{V}_\mathcal{G}} d_v^- \cdot d_v^+ = \mathcal{O}(\tilde{d} \cdot m)$, and building $\mathcal{H}_\Phi$ also requires $\mathcal{O}((\tilde{d} + 2) \cdot m) = \mathcal{O}(\tilde{d} \cdot m)$, where $\tilde{d}$ is a coefficient depending on graph structures, which is usually small and bounded by the maximum outdegree of $\mathcal{G}$. After transforming $\mathcal{G}$ to $\mathcal{H}_\Phi$, kernels' complexities increase from $\mathcal{O}(k \cdot n^{k+1})$ to $\mathcal{O}(k \cdot m^{k+1})$, and k-layer GNNs' complexities increase from $\mathcal{O}(k \cdot m)$ to $\mathcal{O}(k \cdot \tilde{d} \cdot m)$. Kernel complexities dramatically increase, but using $\mathcal{H}_\Phi$ looks more acceptable compared with increasing k; GNNs benefit from parallelization, so the running time also increases linearly. Overall, using $\mathcal{H}_\Phi$ as input is still efficient.

	Models	PROTEINS			D&D			NCI109			NCI1		
		$\mathcal{G}$	$\mathcal{G}_\varphi$	$\mathcal{H}_\Phi$	$\mathcal{G}$	$\mathcal{G}_\varphi$	$\mathcal{H}_\Phi$	$\mathcal{G}$	$\mathcal{G}_\varphi$	$\mathcal{H}_\Phi$	$\mathcal{G}$	$\mathcal{G}_\varphi$	$\mathcal{H}_\Phi$
Kernel	SP	73.48±3.93	**74.20±3.23**	73.39±3.04	80.50±3.66	79.58±3.91	**81.51±3.91**	73.65±2.34	73.84±2.07	74.11±2.22	74.18±1.67	74.70±1.74	74.40±1.74
	GR	70.45±6.54	74.20±4.44	73.66±4.00	78.82±3.83	79.66±5.18	78.82±3.87	66.45±2.14	72.46±2.51	71.81±2.69	65.16±2.30	73.04±1.81	71.07±1.47
	WLOA	72.59±2.46	73.84±3.29	74.02±3.47	79.24±3.61	79.24±3.81	78.57±3.59	85.43±1.51	84.61±1.52	84.81±1.11	85.96±1.82	86.33±1.77	**86.37±1.75**
	1-WL	71.79±4.52	73.30±4.14	73.48±5.02	80.50±4.43	81.26±4.08	80.42±3.85	**85.54±1.34**	83.74±0.94	84.37±1.02	85.13±1.69	84.87±1.77	85.38±1.21
	2-WL	74.11±5.19	75.27±4.67	OOM	OOM	OOM	OOM	68.09±1.55	68.38±1.21	72.24±1.85	67.71±1.33	67.49±1.45	69.00±2.34
	δ-2-WL	74.20±4.98	74.82±4.16	OOM	OOM	OOM	OOM	68.00±1.94	68.26±1.59	70.34±1.87	67.32±1.34	67.37±1.40	69.20±2.18
	δ-2-LWL	73.66±5.10	74.37±3.34	74.11±3.72	77.06±5.99	77.31±5.98	79.41±5.28	84.20±1.44	83.12±1.34	83.82±1.06	85.40±1.28	84.06±1.54	85.40±1.51
	δ-2-LWL$^+$	78.12±4.75	83.48±4.34	**84.55±3.62**	77.14±6.05	77.56±6.30	**79.58±6.24**	88.79±0.94	**89.42±1.37**	88.57±0.97	91.92±1.93	**93.67±0.84**	91.65±1.96
Network	GraphSAGE	73.48±5.66	73.93±5.68	-	77.73±4.66	78.91±4.59	-	73.38±2.68	74.13±2.30	-	73.82±2.17	74.31±2.27	-
	GCN	72.95±3.88	74.02±3.82	-	72.77±4.62	**80.76±5.37**	-	50.34±2.69	51.67±5.52	-	61.75±11.1	68.95±10.8	-
	GIN	73.84±4.46	74.11±4.12	-	76.97±3.87	77.65±3.46	-	72.61±2.37	73.82±2.50	-	73.50±1.80	75.16±1.49	-
	RGCN	73.30±4.90	74.98±4.50	**75.09±4.03**	69.16±9.97	69.24±10.0	78.47±5.24	50.29±2.08	51.52±4.37	71.71±7.59	52.75±4.75	57.27±9.49	74.04±1.15
	RGIN	68.75±6.59	70.54±5.03	74.20±2.93	77.65±4.62	78.15±4.60	77.73±4.42	64.20±2.85	64.52±2.58	**75.43±3.50**	66.11±1.77	66.11±1.69	**76.18±2.03**
	DiffPool	75.62±5.17	**75.98±3.89**	-	81.41±5.11	80.25±4.69	-	75.29±1.85	**75.44±1.90**	-	76.62±1.93	**77.08±1.33**	-
	HGP-SL	71.25±7.13	74.46±3.77	-	74.62±3.19	**82.07±2.11**	-	74.78±2.37	74.32±1.84	-	74.94±0.88	76.08±1.94	-
	Average	*73.13±2.10*	*74.77±2.60*	*75.31±3.53*	*77.20±3.26*	*78.59±3.05*	*79.31±1.13*	*72.07±11.20*	*72.62±10.53*	*77.72±6.51*	*73.48±10.16*	*75.10±8.98*	*78.27±7.77*

Table 1. Accuracies on graph classification, where "OOM" means out-of-memory, the best results are highlighted in bold, and the average results are italicized.

5. Experiment

To evaluate the effectiveness of dummy nodes, we conduct two (sub)graph-level tasks. On graph classification, we use both kernel functions and graph neural networks; on subgraph isomorphism counting and matching, we only employ neural methods upon the generalization. Appendix B provides more details of experiments.

5.1. Graph Classification

Datasets. We select four benchmarking datasets where current state-of-the-art models face overfitting problems: PROTEINS (Borgwardt et al., 2005), D&D (Dobson & Doig, 2003), NCI109, and NCI1 (Wale et al., 2008). The highest accuracies on these datasets are around $70\% \sim 90\%$, and we want to explore the gain of dummy nodes.

Graph Kernels. We use the open-source GKs for a fair comparison. Morris et al. (2020) released a toolkit with an efficient C++ implementation of current best-performed GKs.[1] Based on their experiments, we choose eight kernel functions: Shortest-path Kernel (SP) (Borgwardt & Kriegel, 2005), Graphlet Kernel (GR) (Shervashidze et al., 2009), Weisfeiler-Lehman Optimal Assignment Kernel (WLOA) (Kriege et al., 2016), Weisfeiler-Lehman Subtree Kernels (1-WL and 2-WL) (Shervashidze et al., 2011), their proposed δ-2-WL, δ-2-LWL, and δ-2-LWL$^+$. The kernel functions for $\mathcal{G}_\varphi$ and $\mathcal{H}_\Phi$ are described in Sec. 4.1. Specifically, to handle $\mathcal{G}$, $\mathcal{G}_\varphi$ and $\mathcal{H}_\Phi$, GKs are equipped with their original kernel functions, Eq. (5), and Eq. (6), respectively.

Graph Neural Networks. We adopt the PyG library (Fey & Lenssen, 2019) to implement the neural baselines. We consider the three most-famous networks, GraphSAGE (Hamilton et al., 2017b), GCN (Kipf & Welling, 2017) and GIN (Xu et al., 2019), as well as two state-of-the-art pooling-based networks, DiffPool (Ying et al., 2018) and HGP-SL (Zhang et al., 2019). Seeing that edge-to-vertex-transformed graphs ($\mathcal{H}_\Phi$) involve edge labels, we use two relational GNNs, RGCN (Schlichtkrull et al., 2018) and RGIN (Liu et al., 2020), to better utilize edge type information. Note that we do not concatenate features from $\mathcal{G}$ for fairness, and we hope GNNs can learn from data.

Results and Discussion. Experimental results on graph classification benchmarks are shown in Table 1. Note that some models (GCN, GIN, GraphSAGE, DiffPool, and HGP-SL) are not designed to handle edge types. For a fair comparison, we do not evaluate these models' performance on the transformed graph $\mathcal{H}_\Phi$. We observe consistent improvement in performance after adding dummy nodes ($\mathcal{G}_\varphi$) to the original graphs ($\mathcal{G}$) for most classifiers. When the input is the graphs ($\mathcal{H}_\Phi$) transformed by L_Φ, we also see the further improvement on average. Any progress on graph kernels is not easy, but our kernel modification helps almost all kernels over four datasets. And kernels with the 2-order structures nearly surpass the 1-order kernels and classical graph neural networks with the expressive power no more than 1-WL. Obtaining global high-order structure statistics is time-consuming, but the local variants δ-2-LWL, and δ-2-LWL$^+$ achieves the best tradeoff between efficiency and effectiveness.[2] GNNs with pooling capture the hierarchical graph structures and outperform simple GNNs. GNNs with $\mathcal{G}_\varphi$ can compete against DiffPool with $\mathcal{G}_\varphi$, but DiffPool and HGP-SL can further benefit from the artificial dummy nodes. On the other hand, relational GNNs can handle various edge types in $\mathcal{H}_\Phi$. Accuracies get boosted again when the input changes from $\mathcal{G}_\varphi$ to $\mathcal{H}_\Phi$. We also notice the unstable performance in GCN and RGCN on D&D and NCI1. Using $\mathcal{G}_\varphi$ instead of $\mathcal{G}$ cannot solve their oversmoothing

[1] https://www.github.com/chrsmrrs/sparsewl

[2] Even for the fastest δ-3-LWL$^+$, it either requires over 20,000 seconds (for NCI109 and NCI1) or causes the out-of-memory issue (for PROTEINS and D&D), but there is no further improvement.

Models		Homogeneous						Heterogeneous					
		Erdős-Renyi			Regular			Complex			MUTAG		
		RMSE	MAE	GED	RMSE	MAE	GED	RMSE	MAE	GED	RMSE	MAE	GED
RGCN	$\mathcal{G}$	9.386	5.829	28.963	14.789	9.772	70.746	28.601	9.386	64.122	0.777	0.334	1.441
	$\mathcal{G}_\varphi$	7.764	4.654	24.438	14.077	9.511	71.393	26.389	7.110	55.600	0.534	0.191	1.052
RGIN	$\mathcal{G}$	6.063	3.712	22.155	13.554	8.580	56.353	20.893	4.411	56.263	0.273	0.082	0.329
	$\mathcal{G}_\varphi$	**4.769**	**2.898**	**15.219**	**10.871**	**6.874**	**43.537**	19.436	**3.846**	**41.337**	**0.193**	**0.064**	0.277
HGT	$\mathcal{G}$	24.376	14.630	104.000	26.713	17.482	191.674	34.055	8.336	70.080	1.317	0.526	3.644
	$\mathcal{G}_\varphi$	5.969	3.691	23.401	13.813	8.813	64.926	20.841	4.707	47.409	0.876	0.345	2.973
CompGCN	$\mathcal{G}$	6.706	4.274	25.548	14.174	9.685	64.677	22.287	5.127	57.082	0.300	0.085	0.278
	$\mathcal{G}_\varphi$	4.981	3.019	16.263	11.450	7.443	46.802	20.786	4.048	56.269	0.321	0.089	**0.262**
DMPNN	$\mathcal{G}$	5.330	3.308	23.411	11.980	7.832	56.222	18.974	3.992	56.933	0.232	0.088	0.320
	$\mathcal{G}_\varphi$	5.220	3.130	23.285	11.259	7.136	49.179	**18.885**	3.892	73.161	0.259	0.101	0.623
Deep-LRP	$\mathcal{G}$	0.794	0.436	2.571	1.373	0.788	5.432	27.490	5.850	56.772	0.260	0.094	0.437
	$\mathcal{G}_\varphi$	0.710	0.402	2.218	1.145	0.718	4.611	24.458	5.094	57.398	0.356	0.115	0.849
DMPNN-LRP	$\mathcal{G}$	**0.475**	0.287	1.538	**0.617**	0.422	2.745	20.425	4.173	**32.200**	0.196	0.062	**0.210**
	$\mathcal{G}_\varphi$	0.477	**0.260**	**1.457**	0.633	**0.413**	**2.538**	**18.127**	**4.112**	39.594	**0.186**	**0.057**	0.265

Table 2. Performance on subgraph isomorphism counting and matching.

problem, although their performance becomes slightly better to a certain degree. As a comparison, GraphSAGE using Max and GIN using Sum can provide the steady criterion whenever the input graphs with dummy or not. One negative observation is the over-parameterization problem in RGCN and RGIN. They can perform as expected only when feeding $\mathcal{H}_\Phi$ with numerous edge types. How to help relational GNNs handle dummy edges is one of our future work. We also think the combination of pooling with heterogeneous information is promising.

Relevance to Over-smoothing. Adding dummy nodes to graphs can help alleviate the over-smoothing issue. For instance, on the NCI1 dataset, a 2-layer GIN achieves 73.50±1.80 (75.16±1.49) accuracy on $\mathcal{G}$ ($\mathcal{G}_\varphi$); a 4-layer model achieves 71.97±1.46 (75.11±1.88) accuracy on $\mathcal{G}$ ($\mathcal{G}_\varphi$). We can see that: (1) on $\mathcal{G}$, the model performs worse while its number of layers increases, indicating the over-smoothing does exist, and (2) on $\mathcal{G}_\varphi$, the model achieves comparable performance after stacking layers, meaning adding dummy nodes help overcome the over-smoothing.

To understand why, let's take a look at a 2-layer GNN. The first layer helps every vertex receive one-hop information. At this time, the dummy node aggregates all vertex information. The second layer helps every vertex receive two-hop information (from its neighbors) and global information (from the dummy node). The dummy provides every vertex v with additional information of all vertices, helping v differentiate the intersection of the one-hop and the two-hop, and other vertices beyond two hops. In this way, with the help of the "shortcut" by the dummy, the learned vertex representations of are more distinct from each other. This also applies for deeper layers, and we observe the models with dummy nodes are more robust against over-smoothing.

5.2. Subgraph Isomorphism Counting and Matching

Datasets. We evaluate neural methods on two synthetic homogeneous datasets with 4 graphlet patterns (Chen et al., 2020), one synthetic heterogeneous dataset with 75 random patterns up to eight vertices, and one mutagenic compound dataset MUTAG with 24 artificial patterns (Liu et al., 2020).

End-to-end Framework Considering the complexity and the generalization, we follow the neural framework proposed by Liu et al. (2020) and employ their released implementation.[3] The end-to-end framework includes four parts: encoding, representation, fusion, and prediction. It supports sequence models (e.g., RNN) and graph models (e.g., RGCN). Based on their practical experience, we only consider the effective graph models, including RGCN, RGIN, CompGCN, DMPNN, Deep-LRP, and DMPNN-LRP (Liu & Song, 2022). We also implement HGT (Hu et al., 2020) to see whether relation-specific attention performs well and whether dummy nodes can help it. The framework utilizes one feed-forward network to make predictions at the graph level, i.e., $\text{FFN}_{\text{counting}}(Concat(\boldsymbol{x}_v, \boldsymbol{p}, \boldsymbol{x}_v - \boldsymbol{p}, \boldsymbol{x}_{\mathcal{G}_v} \odot \boldsymbol{p}))$, and another one feed-forward network to make predictions at the vertex level, i.e., $\text{FFN}_{\text{matching}}(Concat(\boldsymbol{g}, \boldsymbol{p}, \boldsymbol{g} - \boldsymbol{p}, \boldsymbol{g} \odot \boldsymbol{p}))$, where $\boldsymbol{p}$ is the pattern representation by applying pooling over the pattern's vertex representations, $\boldsymbol{x}_v$ is the graph vertex representation, and $\boldsymbol{g}$ is the sum of $\{\boldsymbol{x}_v | v \in \mathcal{V}_\mathcal{G}\}$.

Evaluation Metric. The counting prediction is modeled as regression, so RMSE and MAE are adopted to estimate global inference skills. To evaluate the ability of local decision making, we require models to predict the frequency of each node that how many times it appears in all isomorphic subgraphs, so graph edit distance (GED) serves as a metric.

[3] https://github.com/HKUST-KnowComp/DualMessagePassing

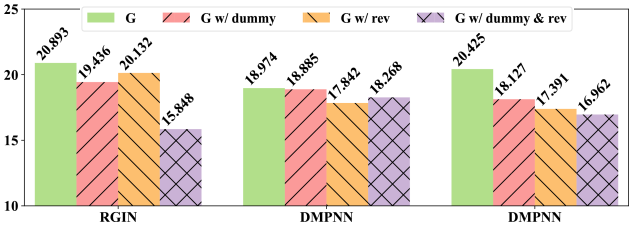


Figure 4. RMSE of RGIN, DMPNN, and DMPNN-LRP with different input graph structures on the *Complex* dataset.

Results and Discussion. Table 2 lists the performance on subgraph isomorphism counting and matching. Almost all graph models consistently benefit from the dummy nodes. In particular, RGIN outperforms other GNNs. Both RGCN and RGIN use relation-specific matrices to transform neighbor messages, but we still observe that RGIN has greater average relative error reductions (19.35% of RMSE and 24.06% of GED) than RGCN (15.28% of RMSE and 13.75% of GED). One interesting observation is that HGT has the most prominent performance boost by a 49.02% relative error reduction of RMSE, a 50.58% relative error reduction of MAE, and a 49.59% relative error reduction of GED on average. We can explain that the dummy nodes provide an option to drop all pattern-irrelevant messages. Without such dummy nodes and edges, irrelevant messages would always be aggregated as side-effects. DMPNN, as the previous state-of-the-art model, has minor overall improvement and even a slight drop of GED on heterogeneous data. The possible reason is the star topology, making the edge representation learning based on line graphs difficult. CompGCN fixes edge presentations and focuses on graph structures, yielding a 17.79% reduction of GED. On the other hand, DMPNN-LRP combines dual message passing with pooling on explicit neighbor permutations and obtains the lowest errors on homogeneous data, but it still faces the same problem that matching errors increase on heterogeneous graphs.

We also consider adding reversed edges and dummy nodes and edges simultaneously. Figure 4 illustrates the counting error changes on the *Complex* data. We note that the two strategies work together well on RGIN, but the boost of DMPNN and DMPNN-LRP mainly comes from reversed edges. We believe this issue can be addressed in the future since we see their cooperation in DMPNN-LRP and the success of graph classification with $\mathcal{H}_\Phi$ in Sec. 5.1.

6. Expressive Power of MPNNs with Dummy Nodes and Transformers with CLS Tokens

In fact, the edge-to-vertex transform L_Φ corresponds to the construction of local 2-tuples in δ-2-LWL$^+$ (Morris et al., 2020). And we can conclude that message passing neural networks (MPNNs) with $\mathcal{H}_\Phi$ have the same expressive power of δ-2-LWL$^+$ with $\mathcal{G}$. And it has been proven that δ-k-LWL$^+$ is strictly more powerful than k-WL. That is, MPNNs with $\mathcal{H}_\Phi$ are more powerful than 2-WL with $\mathcal{G}$. And we also know that MPNNs are no more powerful than 2-WL (Chen et al., 2020) and usually as same powerful as 1-WL (Xu et al., 2019), so we have successfully empowered the MPNNs to surpass 2-WL with the help of $\mathcal{H}_\Phi$. The inverse transform L_Φ^{-1} can always recover $\mathcal{G}$ back if the vertex id information is provided. This implies that MPNNs with $\mathcal{G}_\varphi$ and vertex id information should have the same expressive power as MPNNs with $\mathcal{H}_\Phi$.

Graph attention networks (GATs) (Velickovic et al., 2018) and heterogeneous graph transformers (HGTs) (Hu et al., 2020) also belong to the message passing framework. Therefore, the above discussions are applicable to them. Transformer-based encoders (Vaswani et al., 2017) regard the input as a fully-connected graph and adopt attention to explicitly learn pair-wise connections and implicitly learn global patterns. For each vertex (token), all other vertices and itself are served as its neighbors, indicating a stronger discriminative capability than GATs and HGTs. Thus, transformers with dummy nodes (usually named CLS tokens) should be more powerful than 2-WL. Based on the analyses in (Chen et al., 2020), a 12-layer transformer encoder can capture patterns size of at most $3 \cdot 2^{12} = 12288$, and a 24-layer model can almost learn any patterns. And that is the reason why transformers dominate the sequence encoding.

7. Conclusion

In this paper, we analyze the role of dummy nodes in the lossless edge-to-vertex transform. We further prove that a dummy node with connections to all existing vertices can preserve the graph structure. Specifically, we design an efficient monomorphic edge-to-vertex transform and find its inverse to recover the original graph back. We extend graph kernels and graph neural networks with dummy nodes. Experiments demonstrate the success of performance boost on graph classification and subgraph isomorphism counting and matching. Last, we discuss the capability of MPNNs and Transformers with special dummy elements.

Acknowledgements

The authors of this paper were supported by the NSFC Fund (U20B2053) from the NSFC of China, the RIF (R6020-19 and R6021-20) and the GRF (16211520) from RGC of Hong Kong, the MHKJFS (MHP/001/19) from ITC of Hong Kong and the National Key R&D Program of China (2019YFE0198200) with special thanks to HK-MAAC and CUSBLT, and the Jiangsu Province Science and Technology Collaboration Fund (BZ2021065). We also thank the support from the UGC Research Matching Grants (RMGS20EG01-D, RMGS20CR11, RMGS20CR12, RMGS20EG19, RMGS20EG21).

References

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gülçehre, Ç., Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y., and Pascanu, R. Relational inductive biases, deep learning, and graph networks. *arXiv*, abs/1806.01261, 2018.

Beineke, L. W. Derived graphs and digraphs. *Beiträge zur Graphentheorie*, pp. 17–33, 1968.

Beineke, L. W. and Zamfirescu, C. M. Connection digraphs and second-order line digraphs. *Discrete Mathematics*, 39(3):237–254, 1982.

Bevilacqua, B., Frasca, F., Lim, D., Srinivasan, B., Cai, C., Balamurugan, G., Bronstein, M. M., and Maron, H. Equivariant subgraph aggregation networks. *arXiv*, abs/2110.02910, 2021.

Borgwardt, K. M. and Kriegel, H. Shortest-path kernels on graphs. In *ICDM*, pp. 74–81, 2005.

Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S. V. N., Smola, A. J., and Kriegel, H. Protein function prediction via graph kernels. In *NeurIPS*, pp. 47–56, 2005.

Chen, Z., Chen, L., Villar, S., and Bruna, J. Can graph neural networks count substructures? In *NeurIPS*, 2020.

Dobson, P. D. and Doig, A. J. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, 330(4):771–783, 2003.

Errica, F., Podda, M., Bacciu, D., and Micheli, A. A fair comparison of graph neural networks for graph classification. In *ICLR*, 2020.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML*, volume 70, pp. 1263–1272, 2017.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, 40(3):52–74, 2017a.

Hamilton, W. L., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *NeurIPS*, pp. 1024–1034, 2017b.

Harary, F. *Graph theory*. Addison-Wesley, 1969. ISBN 978-0-201-02787-7.

Hu, Z., Dong, Y., Wang, K., and Sun, Y. Heterogeneous graph transformer. In *WWW*, pp. 2704–2710, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Kriege, N. M., Giscard, P., and Wilson, R. C. On valid optimal assignment kernels and applications to graph classification. In *NeurIPS*, pp. 1615–1623, 2016.

Kriege, N. M., Johansson, F. D., and Morris, C. A survey on graph kernels. *Applied Network Science*, 5(1):6, 2020.

Li, Q., Han, Z., and Wu, X. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*, pp. 3538–3545, 2018.

Liu, X. and Song, Y. Graph convolutional networks with dual message passing for subgraph isomorphism counting and matching. In *AAAI*, 2022.

Liu, X., Pan, H., He, M., Song, Y., Jiang, X., and Shang, L. Neural subgraph isomorphism counting. In *KDD*, pp. 1959–1969, 2020.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR*, 2019.

Maron, H., Fetaya, E., Segol, N., and Lipman, Y. On the universality of invariant networks. In *ICML*, volume 97, pp. 4363–4371, 2019.

Morris, C., Kersting, K., and Mutzel, P. Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In *ICDM*, pp. 327–336, 2017.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI*, pp. 4602–4609, 2019.

Morris, C., Rattan, G., and Mutzel, P. Weisfeiler and leman go sparse: Towards scalable higher-order graph embeddings. In *NeurIPS*, 2020.

Rong, Y., Huang, W., Xu, T., and Huang, J. The truly deep graph convolutional networks for node classification. *arXiv*, abs/1907.10903, 2019.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Schlichtkrull, M. S., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *ESWC*, volume 10843, pp. 593–607, 2018.

Shervashidze, N., Vishwanathan, S. V. N., Petri, T., Mehlhorn, K., and Borgwardt, K. M. Efficient graphlet kernels for large graph comparison. In *AISTATS*, volume 5, pp. 488–495, 2009.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

Teru, K. K., Denis, E., and Hamilton, W. Inductive relation prediction by subgraph reasoning. In *ICML*, volume 119, pp. 9448–9457, 2020.

Vashishth, S., Sanyal, S., Nitin, V., and Talukdar, P. P. Composition-based multi-relational graph convolutional networks. In *ICLR*, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NeurIPS*, pp. 5998–6008, 2017.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *ICLR*, 2018.

Velickovic, P., Buesing, L., Overlan, M. C., Pascanu, R., Vinyals, O., and Blundell, C. Pointer graph networks. In *NeurIPS*, 2020.

Wale, N., Watson, I. A., and Karypis, G. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14 (3):347–375, 2008.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Yang, T., Wang, Y., Yue, Z., Yang, Y., Tong, Y., and Bai, J. Graph pointer neural networks. *arXiv*, abs/2110.00973, 2021.

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W. L., and Leskovec, J. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*, pp. 4805–4815, 2018.

Yuan, H. and Ji, S. Node2seq: Towards trainable convolutions in graph neural networks. *arXiv*, abs/2101.01849, 2021.

Zhang, Z., Wang, M., Xiang, Y., Huang, Y., and Nehorai, A. Retgk: Graph kernels based on return probabilities of random walks. In *NeurIPS*, pp. 3968–3978, 2018.

Zhang, Z., Bu, J., Ester, M., Zhang, J., Yao, C., Yu, Z., and Wang, C. Hierarchical graph pooling with structure learning. *arXiv*, abs/1911.05954, 2019.

Zhu, J., Yan, Y., Zhao, L., Heimann, M., Akoglu, L., and Koutra, D. Beyond homophily in graph neural networks: Current limitations and effective designs. In *NeurIPS*, 2020.

A. Algorithms for L_Φ and L_Φ^{-1}

Algorithm 1 Edge-to-vertex transform L_Φ

input a connected directed graph $\mathcal{G}$ with n vertices and m edges ($m > 0$), a special vertex label for dummy nodes x_φ
let y_φ as the special edge label for dummy edges
add one dummy node φ with label x_φ
for each v in $\mathcal{V}_\mathcal{G} - \{\varphi\}$ **do**
  add two dummy edges (v, φ) and (φ, v) with label y_φ
end for // {end of step ①}
get the line graph $\mathcal{H}_\Phi = L(\mathcal{G})$
assign edge ids with original vertex ids // {end of step ②}
add a single dummy node Φ
for each $e = (u, v)$ in $\mathcal{E}_{\mathcal{H}_\Phi}$ **do**
  if u is associated with label y_φ **and** v is associated with label y_φ **then**
    delete e
  else if u is associated with label y_φ **then**
    add one edge (Φ, v) with e's id and labels $\mathcal{Y}_{\mathcal{H}_\Phi}(e)$
    delete e
  else if v is associated with label y_φ **then**
    add one edge (u, Φ) with e's id and labels $\mathcal{Y}_{\mathcal{H}_\Phi}(e)$
    delete e
  end if
end for
for each v in $\mathcal{V}_{\mathcal{H}_\Phi}$ **do**
  if v is associated with label y_φ **then**
    delete v
  end if
end for // {end of step ③}
output the transformed graph $\mathcal{H}_\Phi$

Algorithm 2 Inverse edge-to-vertex transform L_Φ^{-1}

input a transformed graph $\mathcal{H}_\Phi$ obtained by L_Φ, a special vertex label for dummy nodes x_φ
get the line graph $\mathcal{G} = L(\mathcal{H}_\Phi)$
assign vertex ids with the original edge ids // {end of step ①}
create an empty mapping $\mathcal{I}$
for each $e = (u, v)$ in $\mathcal{E}_\mathcal{G}$ **do**
  if $u.id$ not in $\mathcal{I}$ **and** $v.id$ not in $\mathcal{I}$ **then**
    set $\mathcal{I}(u.id) \leftarrow u$ and $\mathcal{I}(v.id) \leftarrow v$
  else if $u.id$ not in $\mathcal{I}$ **then**
    add one edge $(u, \mathcal{I}(v.id))$ with e's id and labels
    delete e
    set $\mathcal{I}(u.id) \leftarrow u$
  else if $v.id$ not in $\mathcal{I}$ **then**
    add one edge $(\mathcal{I}(u.id), v)$ with e's id and labels
    delete e
    set $\mathcal{I}(v.id) \leftarrow v$
  else if $u \neq \mathcal{I}(u.id)$ **or** $v \neq \mathcal{I}(v.id)$ **then**
    add one edge $(\mathcal{I}(u.id), \mathcal{I}(v.id))$ with e's id and labels
    delete e
  end if
end for
for each v in $\mathcal{V}_\mathcal{G}$ **do**
  if $v \neq \mathcal{I}(v.id)$ **then**
    delete v
  end if
end for // {end of step ②}
for each e in $\mathcal{E}_\mathcal{G}$ **do**
  if e is associated with label x_φ **then**
    delete e
  end if
end for // {end of step ③}
output the transformed graph $\mathcal{G}$

B. Details of Experiments

B.1. Environment

We conduct our experiments on one CentOS 7 server with 2 Intel Xeon Gold 5215 CPUs and 4 NVIDIA GeForce RTX 3090 GPUs. The software versions are: GNU C++ Compiler 5.2.0, Python 3.7.3, PyTorch 1.7.1, torch-geometric 2.0.2, and DGL 0.6.0.

B.2. Graph Classification

B.2.1. Datasets

Dataset statistics are listed in Table 3. To construct the dataset with dummy node information, we add an extra dummy node to each graph and connect it with all the other vertices in the graph with bidirectional edges. When one graph is undirected, we employ the common practice to replace one undirected edge with one directed edge and its reverse. Following previous work (Zhang et al., 2019), we randomly split each dataset into the training set (80%), the validation set (10%), and the test set (10%) in each run.

B.2.2. Implementation Details

- **Kernel Methods.** We compile kernel functions with C++11 features and -O2 flag. After obtaining normalized Gram matrices, SVM classifiers are trained based on LibSVM[4] and wrapped by sklearn.
- **Graph Neural Network Based Methods.** We implement and evaluate all our graph neural network based models with the PyG library. For GraphSAGE, GIN and DiffPool, we adapt the implementation by Errica et al. (2020).[5] To implement our RGCN, we modify the PyG implementation[6] by adding 3 fully connected layers after the convolutional layers. The RGIN convolutional layer can be adapted from a RGCN convolutional layer, by setting the aggregation function as *Sum*, and followed by a multi-layer perceptron. For HGP-SL, we use the official code.[7] We observe a performance drop

[4] https://www.csie.ntu.edu.tw/~cjlin/libsvm
[5] https://github.com/diningphil/gnn-comparison
[6] https://github.com/pyg-team/pytorch_geometric/blob/master/examples/rgcn.py
[7] https://github.com/cszhangzhen/HGP-SL

Dataset		# Graphs	# Classes	Avg. $\lvert\mathcal{V}_\mathcal{G}\rvert$	Avg. $\lvert\mathcal{E}_\mathcal{G}\rvert$	$\lvert\mathcal{X}_\mathcal{G}\rvert$	$\lvert\mathcal{Y}_\mathcal{G}\rvert$
PROTEINS	$\mathcal{G}$	1,113	2	39.1	145.6	3	1
	$\mathcal{G}_\varphi$	1,113	2	40.1	223.7	4	2
	$\mathcal{H}_\Phi$	1,113	2	146.6	885.9	2	4
D&D	$\mathcal{G}$	1,178	2	284.3	1431.3	82	1
	$\mathcal{G}_\varphi$	1,178	2	285.3	2000.0	83	2
	$\mathcal{H}_\Phi$	1,178	2	1432.3	10875.5	2	83
NCI109	$\mathcal{G}$	4,127	2	29.7	64.3	38	1
	$\mathcal{G}_\varphi$	4,127	2	30.7	123.6	39	2
	$\mathcal{H}_\Phi$	4,127	2	65.3	285.8	2	39
NCI1	$\mathcal{G}$	4,110	2	29.9	64.6	37	1
	$\mathcal{G}_\varphi$	4,110	2	30.9	124.3	38	2
	$\mathcal{H}_\Phi$	4,110	2	65.6	287.0	2	38

Table 3. Dataset statistics on graph classification.

compared with the results reported in their paper, either when it runs with a newer version of torch-sparse (in our setting, torch-sparse=0.6.9), or with the version reported in their GitHub page (torch-sparse=0.4.0). For a fair comparison with other baseline models, we choose to use our current software versions.

B.2.3. Hyper-parameter Settings

For reproducibility, we run all the experiments 10 times with random seeds $\{2020, 2021, \cdots, 2029\}$, and report the sample mean and standard deviation of test accuracies.

- **Kernel Methods.** For kernel methods, we search for the regularization parameter C within $\{10^{-7}, 10^{-6}, \cdots, 10^{3}\}$ for each seed and corresponding training data. The best hyper-parameter setting is chosen by the best validation performance.

- **Graph Neural Network Based Methods.** We use the Adam optimizer (Kingma & Ba, 2015) to optimize the models. Following Zhang et al. (2019), an early stopping strategy with patience 100 is adopted during training, i.e., training would be stopped when the loss on validation set does not decrease for over 100 epochs. For GraphSAGE, GIN and DiffPool, the optimal hyper-parameters are found using grid search within the same search ranges in (Errica et al., 2020). For HGP-SL, we follow the official hyper-parameters reported in their GitHub repository. For models using graph convolutional operator (Kipf & Welling, 2017) (GCN, DiffPool, HGP-SL), we additionally impose a learnable weight γ on the dummy edges. The weights for all the other edges is set as 1, and γ is initialized with different values in $\{0.01, 0.1, 1, 10\}$. For relational models RGCN and RGIN, we search for the learning rate within $\{1e-2, 1e-3, 1e-4\}$, batch size $\in \{128, 512\}$, hidden dimension $\in \{32, 64\}$, dropout ratio $\in \{0, 0.5\}$, and number of layers $\in \{2, 4\}$.

	Erdős-Renyi		Regular		Complex		MUTAG	
# Train	6,000		6,000		358,512		1,488	
# Valid	4,000		4,000		44,814		1,512	
# Test	10,000		10,000		44,814		1,512	
	Max	Avg.	Max	Avg.	Max	Avg.	Max	Avg.
$\lvert\mathcal{V}_\mathcal{P}\rvert$	4	3.8±0.4	4	3.8±0.4	8	5.2±2.1	4	3.5±0.5
$\lvert\mathcal{E}_\mathcal{P}\rvert$	10	7.5±1.7	10	7.5±1.7	8	5.9±2.0	3	2.5±0.5
$\lvert\mathcal{X}_\mathcal{P}\rvert$	1	1±0	1	1±0	8	3.4±1.9	2	1.5±0.5
$\lvert\mathcal{Y}_\mathcal{P}\rvert$	1	1±0	1	1±0	8	3.8±2.0	2	1.5±0.5
$\lvert\mathcal{V}_\mathcal{G}\rvert$	10	10±0	30	18.8±7.4	64	32.6±21.2	28	17.9±4.6
$\lvert\mathcal{E}_\mathcal{G}\rvert$	48	27.0±6.1	90	62.7±17.9	256	73.6±66.8	66	39.6±11.4
$\lvert\mathcal{X}_\mathcal{G}\rvert$	1	1±0	1	1±0	16	9.0±4.8	7	3.3±0.8
$\lvert\mathcal{Y}_\mathcal{G}\rvert$	1	1±0	1	1±0	16	9.4±4.7	4	3.0±0.1

Table 4. Dataset statistics on subgraph isomorphism experiments. $\mathcal{P}$ and $\mathcal{G}$ corresponds to patterns and graphs.

B.3. Subgraph Isomorphism Counting and Matching

B.3.1. Datasets

The statistics of datasets on the subgraph isomorphism counting and matching task are listed in Table 4.

B.3.2. Implementation Details

We adapt the DGL implementation provided by Liu & Song (2022)[8] to evaluate the effect of dummy nodes on neural subgraph isomorphism counting and matching. Apart from that, we add a new implementation of HGT (Hu et al., 2020). We learn from experience and lessons to jointly train models for counting and matching in a multitask setting.

B.3.3. Hyper-parameter Settings

We follow the same paradigm for the training and evaluation in the multitask learning setting. The best model over validation data among three random seeds $\{0, 2020, 2022\}$ is reported.

The embedding dimensions and hidden sizes are set as 64 for all 3-layer networks. Deep-LRP and DMPNN-LRP enumerate neighbor subsets by 3-truncated BFS. The HGT model is also set as the same hyper-parameters. Residual connections and Leaky ReLU are added between two layers. We use the AdamW optimizer (Loshchilov & Hutter, 2019) to optimize the models with a learning rate $1e-3$ and a weight decay $1e-5$.

[8] https://github.com/HKUST-KnowComp/DualMessagePassing