Removing Redundancy in Graph Neural Networks

Franka Bause*[1,2], Samir Moustafa*[1,2],
Wilfried N. Gansterer[1], and Nils M. Kriege[1,3]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
[2] UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria
[3] Research Network Data Science, University of Vienna, Vienna, Austria
{firstname.lastname}@univie.ac.at
* Both authors contributed equally to this article.

Abstract. Message passing graph neural networks iteratively compute node embeddings by aggregating messages from all neighbors. This procedure can be viewed as a neural variant of the Weisfeiler-Leman method, which naturally limits their expressive power. Moreover, oversmoothing and oversquashing limit the number of layers effectively utilized by these networks, restricting their expressive power further. We identify the repeated exchange and encoding of identical information as a weak point and propose a non-redundant aggregation scheme to alleviate the problem. We develop a neural tree canonization technique and apply it to neighborhood trees, which represent node neighborhoods similarly to Weisfeiler-Leman unfolding trees, but restrict the repetition of nodes. Our method is provably more powerful than the Weisfeiler-Leman method, potentially less susceptible to oversquashing and oversmoothing than message passing neural networks, and provides high classification accuracy on widely-used benchmark datasets.

Keywords: Graph neural networks · Message passing · Weisfeiler-Leman.

1 Introduction

Graph neural networks have recently emerged as the dominant technique for machine learning with graphs. The class of message passing neural networks [5] is widely-used and updates node embeddings layer-wise by combining the current embedding of a node with the embeddings of its neighbors involving learnable parameters. Suitable neural architectures admit a parametrization such that each layer represents an injective function encoding its input uniquely by the new embedding [16]. In this case the message passing neural network has the same expressive power as the Weisfeiler-Leman algorithm [16]. The Weisfeiler-Leman algorithm can distinguish two nodes if and only if the unfolding trees representing their neighborhoods are non-isomorphic. This unfolding tree corresponds to the computational tree of message passing neural networks [15,6]. These results prove the existence of parameters such that two nodes obtain different embeddings after layer n if they are distinguished after n iterations of the Weisfeiler-Leman algorithm meaning their unfolding trees of height n are non-isomorphic. However,

in practice typically shallow message passing neural networks are employed. Two phenomena have been identified explaining the poor performance of deep message passing neural networks. First, node representations are observed to converge to the same values for deep architecture instead of being able to distinguish more vertices, a phenomenon referred to as *oversmoothing* [9,10]. Second, *oversquashing* [4] refers to the problem that the neighborhood of a node grows exponentially with the number of layers and aggregation steps and, therefore, cannot be supposed to be accurately represented by a fixed-sized embedding. We argue that oversquashing can be alleviated by removing the encoding of repeated information. Consider a node u with an edge $e = \{u, v\}$ in an undirected graph. In a first step, u will send information to v over the edge e. In the next step, u will receive a message form v via e that incorporates the information that u has previously sent to v. Clearly, this information is redundant. In the context of walk-based graph learning method this problem is well-known and referred to as *tottering* [11].

Our Contribution. We systematically investigate the information redundancy in message passing neural networks and develop principled techniques to avoid superfluous messages. Fundamental to our consideration is the tree representation implicitly used by message passing neural networks and the Weisfeiler-Leman method. First, we develop a neural tree canonization approach processing trees systematically in a bottom-up fashion. The method leads to a message passing neural network encoding unfolding trees, but achieves this without redundancy. Second, we apply the canonization technique to *neighborhood trees*, which are obtained from unfolding trees by deleting nodes that appear multiple times. We show that the neighborhood trees allow to distinguish nodes and graphs that cannot be distinguished by the Weisfeiler-Leman method, rendering our technique more expressive than message passing neural networks. While our approach removes information redundancy on a node level, the structure of subtrees now depends on the selected root leading to computational challenges. We address these by compact representations of neighborhood trees using directed acyclic graphs allowing to reuse canonical representations of subtrees. Our method is shown to achieve high accuracy on several graph classification tasks.

2 Related Work

The graph isomorphism network (GIN) [16] is a graph neural network that generalizes the Weisfeiler-Leman algorithm and reaches its expressive power. The embedding of a vertex v in layer i of GIN is defined as follows:

$$x_i(v) = \mathrm{MLP}_i\left((1+\epsilon_i)\cdot x_{i-1}(v) + \sum_{u\in N(v)} x_{i-1}(u)\right), \tag{1}$$

and the first representation $x_0(v)$ is usually acquired by applying a multi-layer perceptron (MLP) to the vertex features. The limited expressiveness of simple

message passing neural networks has lead to an increased interest in researching the expressiveness of GNNs and finding more powerful architectures, for example by encoding graph structure as additional features or altering the message passing procedure. Shortest Path Networks [1] are closely related to our approach. This method uses multiple aggregation functions for different shortest path lengths: One for each k for the k-hop neighbors. While this allows the target node to directly communicate with nodes further away and in turn possibly might help mitigate oversquashing, some information about the structure of the neighborhood can still not be represented adequately and the gain in expressiveness is limited. In Distance Encoding GNNs [8], the distances of the nodes to a set of target nodes are encoded. While this approach also is provably more expressive than the standard WL method, it is limited to solving node-level tasks, since the encoding depends on a fixed set of target nodes, and has not been employed for graph-level tasks. MixHop [2] employs an activation function for each neighborhood and concatenates their results. However, in contrast to [1], the aggregation is based on normalized powers of the adjacency matrix, not shortest paths, which does not solve the problem of redundant messages. SPAGAN [17] proposes a path-based attention mechanism. Although the idea is very similar, shortest paths are only sampled and the feature aggregation differs. Only one layer is used and the paths are used as features. This approach has not been investigated theoretically. IDGNN [18] keeps track of the identity of the root node in the unfolding tree, which allows for more expressiveness than 1-WL. Their variant ID-GNN-Fast works by using cycles lengths as additional node features. Both variants however, do not reduce the amount of redundant information that is aggregated over multiple layers.

For many of these architectures no thorough investigation on their expressiveness and connections to other approaches is provided. Moreover, these works do not explicitly investigate redundancy in message passing neural networks.

3 Fundamental Techniques

In this section, we give an overview of the necessary definitions and the notation used throughout the article and introduce fundamental techniques.

Graph Theory. A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices V, a set of edges $E \subseteq V \times V$ between them and functions $\mu\colon V \to X$ and $\nu\colon E \to X$ assigning arbitrary attributes to the vertices and edges, respectively. We refer to an edge from u to v by uv, and in case of undirected graphs $uv = vu$. The vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively, and the *neighbors* (or in-neighbors) of a vertex $u \in V$ are denoted by $N(u) = \{v \mid vu \in E\}$. The *out-neighbors* of a vertex $u \in V$ are denoted by $N_o(u) = \{v \mid uv \in E\}$. Two graphs G and H are isomorphic, denoted by $G \simeq H$, if there exists a bijection $\phi\colon V(G) \to V(H)$, so that $\forall u, v \in V(G)\colon \mu(v) = \mu(\phi(v)) \wedge uv \in E(G) \Leftrightarrow \phi(u)\phi(v) \in E(H) \wedge \forall uv \in E(G)\colon \nu(uv) = \nu(\phi(u)\phi(v))$. We call ϕ an *isomorphism* between G and H.

Fig. 1: Example of an in-tree (left) and a DAG (right).

An *in-tree* T is a connected, directed, acyclic graph with a distinct vertex $r \in V(T)$ with no outgoing edges referred to as *root*, denoted by $r(T)$, in which $\forall v \in V(T) \backslash r(T) : |N_o(v)| = 1$. For $v \in V(T) \backslash r(T)$ the *parent* $p(v)$ is defined as the unique vertex $u \in N_o(v)$, and $\forall v \in V(T)$ the *children* are defined as $\mathrm{chi}(v) = N(v)$. We refer to all vertices with no incoming edges as *leaves* denoted by $l(T) = \{v \in V(T) \mid \mathrm{chi}(v) = \emptyset\}$. Conceptually it is a directed tree, in which there is a unique directed path from each vertex to the root [12]. In our paper, we only cover in-trees and will thereby just refer to them as *trees*. In-trees are generalized by directed, acyclic graphs (DAGs). The *leaves* of a DAG D and the *children* of a vertex are defined as in trees. However, there can be multiple roots and a vertex may have more than one parent. We refer to all vertices in D with no outgoing edges as *roots* denoted by $r(D) = \{v \in V(D) \mid N_o(v) = \emptyset\}$ and define the *parents* $p(v)$ of a vertex v as $p(v) = N_o(v)$. For clarity we refer to the vertices of a DAG as nodes to distinguish them from the graphs that are the input of a graph neural network.

Weisfeiler-Leman Unfolding Trees. The 1-dimensional Weisfeiler-Leman (WL) algorithm or *color refinement* starts with all vertices having a color representing their label (or a uniform coloring in case of unlabeled vertices). In each iteration the color of a vertex is updated based on the multiset of colors of its neighbors according to

$$c_{i+1}(v) = h\left(c_i(v), \{\!\{c_i(u) \mid u \in N(v)\}\!\}\right) \quad \forall v \in V(G),$$

where h is an injective function.

The color of a vertex of G encodes its neighborhood by a tree T that may contain multiple representatives of each vertex in G. Let $\phi\colon V(T) \to V(G)$ be a mapping such that $\phi(n) = v$ if the node n in $V(T)$ represents the vertex v in $V(G)$. The *unfolding tree* F_i^v with height i of the vertex $v \in V(G)$ consists of a root n_v with $\phi(n_v) = v$ and child subtrees F_{i-1}^u for all $u \in N(v)$, where $F_0^v = (\{n_v\}, \emptyset)$. The attributes of the original graph are preserved, see Figure 2 for an example. The unfolding trees F_i^v and F_i^w of two vertices v and w are isomorphic if and only if $c_i(v) = c_i(w)$.

Fig. 2: Graph G and its unfolding trees F_2^v for all $v \in V(G)$.

(a) Graph G (b) F_2^v (c) $T_{2,0}^v$ (d) $T_{2,1}^v$

Fig. 3: Graph G and the unfolding, 0- and 1-redundant neighborhood trees of height 2 of vertex v (vertex in the upper left of G).

***k*-redundant Neighborhood Trees.** A k-redundant neighborhood tree (k-NT) $T_{i,k}^v$ can be constructed from the unfolding tree F_i^v by deleting all subtrees with its roots, that already occurred more than k levels before (seen from root to leaves). Let $\text{depth}(v)$ denote the length of the path from v to the root and $\phi(v)$ again denote the original vertex in $V(G)$ represented by v in the unfolding or neighborhood tree.

Definition 1 (k-redundant Neighborhood Tree). *For $k \geq 0$, the* k-redundant neighborhood tree *(k-NT) of a vertex $v \in V(G)$ with height i, denoted by $T_{i,k}^v$, is defined as the subtree of the unfolding tree F_i^v induced by the nodes $u \in V(F_i^v)$ satisfying*

$$\forall w \in V(F_i^v) \colon \phi(u) = \phi(w) \Rightarrow \text{depth}(u) \leq \text{depth}(w) + k.$$

Figure 3 shows an example of unfolding and neighborhood trees. Note that for $k \geq i$ the k-redundant neighborhood tree is equivalent to the WL unfolding tree.

4 Non-Redundant Graph Neural Networks

We propose to restrict the information flow in message passing to control redundancy using k-redundant neighborhood trees. We first develop a neural tree canonization technique and obtain an MPNN via its application to unfolding trees. Then we investigate computational methods on a graph-level reusing information computed for subtrees and derive a customized GNN architecture. Finally, we prove that 1-redundant neighborhood trees are strictly more expressive than unfolding trees on both node- and graph-level.

4.1 A Tree Canonization Perspective on GNNs

It is well-known that two nodes obtain the same WL color if and only if their unfolding trees are isomorphic and this concept directly carries over to message passing neural networks and their computational tree [15,6]. However, unfolding trees were mainly used as a tool in expressivity analysis and as a concept explaining mathematical properties in graph learning [7,14]. We develop a tree canonization perspective on MPNNs which will extend to a novel non-redundant GNN architecture in the following.

The AHU Algorithm for Tree Isomorphism and Canonization. Aho, Hopcroft and Ullman describe a linear time isomorphism test for rooted unordered trees in their classical text book [3, Section 3.2] based on radix sort. We describe the general procedure of the algorithm abstractly in order to lay the foundations for our neural variant without focusing on the running time. The algorithm proceeds in a bottom-up fashion and assigns integers $c(v)$ to each node v of the two trees. Initially, $c(v) = 0$ for all leaves. Then, the internal nodes are processed level-wise in a bottom-up fashion guaranteeing that for every considered node all its children have been processed. In step i, we assign $c(v) = f_i(\{\!\{c(u) \mid u \in \mathrm{chi}(v)\}\!\})$ to each internal node v on level i. Here, the function f_i assigns integers starting with 1 to the sorted list of multisets of level i using the next integer whenever it differs from the one processed before. The algorithm identifies the trees as non-isomorphic whenever the integers on some level of the two trees differ; they are isomorphic when eventually their roots are assigned the same integer. The multisets on the same level are sorted efficiently by maintaining and scanning sorted lists of lower-level nodes for both trees, sorting tuples using radix sort. In this case, the overall algorithm achieves linear running time. As noted in [3], the algorithm can be extended to labeled trees by adding node labels as first element to the tuples, assuming that the node label $\mu(v)$ is in $\{1, \ldots, n\}$ for all nodes v.

The idea easily extends to tree canonization assuming that a function f is computable that assigns a pair consisting of an integer and a multiset of integers injectively to a new (unused) integer. In this case, we have

$$c(v) = \begin{cases} f(\mu(v), \emptyset) & \text{if } v \text{ is a leaf,} \\ f(\mu(v), \{\!\{c(u) \mid u \in \mathrm{chi}(v)\}\!\}) & \text{otherwise.} \end{cases} \tag{2}$$

Here, we ignore the running time allowing us to discard the requirements regarding the initial labels $\mu(v)$, and neglect the fact that such a function f is non-trivial to realize for the sake of illustrating the overall concept and linking to neural approaches.

Canonization of Unfolding Trees. We combine Eq. (2) with the definition of unfolding trees and denote the root of an unfolding tree of height i of a vertex v by n_v^i. Then, we obtain

$$c(n_v^i) = f(\mu(n_v^i), \{\!\{c(n_u^{i-1}) \mid n_u^{i-1} \in \mathrm{chi}(n_v^i)\}\!\}) = f(\mu(v), \{\!\{c(n_u^{i-1}) \mid u \in N(v)\}\!\}). \tag{3}$$

Realizing f using a suitable neural architecture and replacing integers by embeddings in $\mathbb{R}^d$ we immediately obtain a GNN from our canonization approach. The only notable difference to standard GNNs is that the first component of the pair in Eq. (3) is $\mu(v)$ instead of $x_{i-1}(v)$. We can derive a GNN based on unfolding tree canonization from Eq. (1) by replacing the first addend with the initial embedding according to

$$x_i(v) = \mathrm{MLP}_i\left((1 + \epsilon_i) \cdot x_0(v) + \sum_{u \in N(v)} x_{i-1}(u)\right). \tag{4}$$

We refer to the above equation as *unfolding tree canonization MPNN*. It is known that MPNNs cannot distinguish two nodes with the same WL color or unfolding tree. Since the function $c(n_v^i)$ uniquely represents the unfolding tree for an injective function f which is realized by Eq. (4) using the same technique as GIN [16], we conclude the following.

Proposition 1. *Unfolding tree canonization MPNNs given by Eq.* (4) *are as expressive as GIN given by Eq.* (1).

However, since $c(n_v^{i-1})$ or $x_{i-1}(v)$ may represent the whole unfolding tree rooted at v of height $i-1$, it clearly may contain redundant information to a large extent, which is avoided by the canonization-based approach. We proceed by investigating redundancy in unfolding trees themselves.

4.2 Removing Redundancy in Unfolding Trees

The computation DAG of an MPNN can be represented by merging all isomorphic substructures of the unfolding trees of the vertices. We will first describe, how to merge trees in general and then look at the consequences for the computation DAG in case of using unfolding trees, as well as neighborhood trees.

Merge DAGs. The neural tree canonization approach developed in the last section can directly be applied to DAGs. Given a DAG D, it computes an embedding for each node n in D that represents the tree F_n obtained by recursively following its children similar as in unfolding trees, cf. Section 3. Since D is acyclic the height of F_n is bounded. Given a set of trees $\mathcal{T} = \{T_1, \dots, T_n\}$, a *merge DAG* of $\mathcal{T}$ is a pair (D, ξ), where D is a DAG, $\xi\colon \{1, \dots n\} \to V(D)$ is a mapping and for all $i \in \{1, \dots, n\}$ we have $T_i \simeq F_{\xi(i)}$. The definition guarantees that the neural tree canonization approach applied to the merge DAG produces the same result for the nodes in the DAG as for the nodes in the original trees. A trivial merge DAG is the disjoint union of the trees with $\xi(i) = r(T_i)$. However, depending on the structure of the given trees, we can identify the subtrees they have in common and represent them only once, such that two nodes of different trees share the same child, resulting in a DAG.

We can identify the shared substructures by adapting the AHU algorithm, cf. Section 4.1. For this work, we used a straightforward algorithm that inserts the nodes of a tree in a bottom-up fashion into a DAG and identifies common substructures in passing, see Algorithm 1 in the appendix. When two siblings that are the roots of isomorphic subtrees are merged this leads to parallel edges in the DAG. We avoid this by introducing a node labeling function extending the original labels of the tree. Let $V_T = \bigcup_{i \in \{1,\dots,n\}} V(T_i)$, then a suitable labeling function $L\colon V_T \to \mathcal{O}$ with $\mathcal{O}$ some arbitrary labeling satisfies $\forall u, v \in V_T\colon L(u) = L(v) \implies \mu(u) = \mu(v) \wedge p(u) \neq p(v)$. The algorithm extends to the case where the input is a set of DAGs allowing parallel pairwise merging, see Section C.

(a) Merged unfolding trees (b) Merged 0-NTs

Fig. 4: Computation DAGs for unfolding and neighborhood trees of height 2.

Merging Unfolding Trees. Merging the Weisfeiler-Leman unfolding trees of a graph with the labeling $L = \phi$ leads to the computation DAG of GNNs. Figure 4a shows the computation DAG for the example graph from Figure 2. The roots in this DAG correspond to the representation of the vertices after aggregating information from the lower layers. Every node occurs once at every layer of the DAG and the links between any two consecutive layers are given by the adjacency matrix of the original graph. While this makes computation based on the adjacency matrix widely used for MPNNs simple, a lot of redundant information is incorporated which can be avoided using neighborhood trees.

Merging Neighborhood Trees. Merging the k-redundant neighborhood trees in the same way using the labeling $L = \phi$ leads to a computation DAG having a less regular structure, see Figure 4b for an example. First, there might be multiple nodes on the same level representing the same original vertex. Second, the adjacency matrix of the original graph cannot be used to propagate the information. We apply the neural tree canonization approach to the merge DAG it in a bottom-up fashion from the leaves to the roots. Each edge is used exactly once in this computation. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the computation DAG of the graph $G = (V, E)$. The edge set $\mathcal{E}$ is partitioned into subsets $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$, with $\bigcup_{i \in \{1,n\}} \mathcal{E}_i = \mathcal{E}$ and $\forall i \neq j : \mathcal{E}_i \cap \mathcal{E}_j = \emptyset$, where subset $\mathcal{E}_i$ represents layer i. Edges are assigned to the layers, so that

$$\mathcal{E}_n = \{uv \in \mathcal{E} \mid N_o(v) = \emptyset\}$$
$$\mathcal{E}_i = \{uv \in \mathcal{E} \mid i = \min\{j \mid \exists vw \in \mathcal{E}_j\} - 1\}$$

This results in a unique partition, in which all edges with the same end node v are in the same layer and all edges pointing to children of the starting node u are in some previous layer. The pseudocode of the algorithm can be found in the appendix.

Fig. 5: Graph G and its 0-NTs $T^v_{2,0}$ for all $v \in V(G)$.

(a) First layer (b) Second layer

Fig. 6: Edges in the different layers of the merge DAG based on 0-NTs.

Let $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ be the resulting edge partition. All incoming edges of a node $v \in \mathcal{V}$ are in the same edge set $\mathcal{E}_i$ and, thus, the edge partition induces the partition $\{\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_n\}$ of the nodes $\mathcal{V}$ as follows:

$$\begin{aligned}\mathcal{L}_0 &= \{v \in \mathcal{V} \mid N(v) = \emptyset\}\\ \mathcal{L}_i &= \{v \in \mathcal{V} \mid \exists u \in \mathcal{V} \colon uv \in \mathcal{E}_i\}\end{aligned}$$

4.3 Non-Redundant Neural Architecture (DAG-MLP)

We use the k-redundant computation DAG and develop a neural canonization technique that is suitable for implementations accelerated by standard deep learning frameworks and hardware. In a pre-processing step we transform the node labels using MLP_0 to a embedding space of a fixed dimension. We use an MLP_i to processes the nodes on the DAG layer $\mathcal{L}_i$.

$$\begin{aligned}\mu'(v) &= \mathrm{MLP}_0\left(\mu(v)\right) && \forall v \in \mathcal{V}\\ x(v) &= \mu'(v) && \forall v \in \mathcal{L}_0\\ x(v) &= \mathrm{MLP}_i\left((1+\epsilon_i)\cdot\mu'(v) + \sum_{\forall u\colon uv \in \mathcal{E}_i} x(u)\right) && \forall v \in \mathcal{L}_i, i \in \{1,\ldots,n\}\end{aligned} \tag{5}$$

The DAG-MLP can be computed through iterated matrix vector multiplication, similar to the standard GNNs. Let $\mathbf{L}_i$ be the square matrix with ones on the diagonal at position j if $v_j \in \mathcal{L}_i$, and zeros elsewhere. Let $\mathbf{E}_i$ be the adjacency matrix of $(\mathcal{V}, \mathcal{E}_i)$, and $\mathbf{F}$ the node features of $\mathcal{V}$ corresponding to the initial labels of V. The transformed features $\mathbf{F}'$ are obtained using the pre-processing MLP, and $\mathbf{X}^{[i]}$ represents the updated embeddings at layer i of the DAG.

$$\begin{aligned}\mathbf{F}' &= \mathrm{MLP}_0\left(\mathbf{F}\right)\\ \mathbf{X}^{[0]} &= \mathbf{L}_0\mathbf{F}'\\ \mathbf{X}^{[i]} &= \mathrm{MLP}_i\left((1+\epsilon_i)\cdot\mathbf{L}_i\mathbf{F}' + \mathbf{E}_i\mathbf{X}^{[i-1]}\right) + \mathbf{X}^{[i-1]},\end{aligned} \tag{6}$$

where MLP_i is applied to the rows associated with nodes in $\mathcal{L}_i$. Here, $\mathbf{X}^{[i]}$ are the embeddings of the nodes of the DAG, which are initially all zero for the inner

nodes and computed level-wise. To preserve the embeddings of all previous layers $\mathbf{X}^{[i-1]}$ is added in the computation of $\mathbf{X}^{[i]}$. The final embeddings are the rows of $\mathbf{X}^{[n]}$ associated with the roots of the DAG.

4.4 Expressiveness of 1-NTs

Let φ be an isomorphism between G and H. We call two nodes u and v (or edges uw and vx) corresponding in an isomorphism φ, if $\varphi(u) = v$ (for edges $\varphi(u)\varphi(w) = vx$). We denote two nodes u and v (or edges uw and vx) by $u \cong v$ ($uw \cong vx$, respectively), if there exists an isomorphism in which u and v (uw and vx) are corresponding.

For isomorphism testing the (multi-)sets of unfolding trees of two graphs (and k-redundant neighborhood trees, respectively) can be compared. The sets are denoted with $wl_i(G)$ and $nt_{i,k}(G)$ for iteration i and defined as $wl_i(G) = \{\!\{F_i^v | v \in V(G)\}\!\}$ and $nt_{i,k}(G) = \{\!\{T_{i,k}^v | v \in V(G)\}\!\}$.

If two graphs G and H are isomorphic, then by definition of the trees, we can find a bijection ψ between their tree sets $wl_\infty(G)$ and $wl_\infty(H)$ (and $nt_{\infty,k}(H)$ and $nt_{\infty,k}(G)$, respectively), with $\forall T\colon T \simeq \psi(T)$, which we denote by $wl_\infty(G) = wl_\infty(H)$ ($nt_{\infty,k}(G) = nt_{\infty,k}(H)$). However, $wl_\infty(G) = wl_\infty(H) \not\Rightarrow G \simeq H$ (and also $nt_{\infty,k}(G) = nt_{\infty,k}(H) \not\Rightarrow G \simeq H$). We focus on 1-redundant neighborhood trees from now on.

Theorem 1. *The 1-NT isomorphism test is more powerful than the Weisfeiler-Leman isomorphism test, i.e.,*

1. $\forall G, H\colon wl_\infty(G) \neq wl_\infty(H) \Rightarrow nt_{\infty,1}(G) \neq nt_{\infty,1}(H)$
2. $\exists G, H\colon G \not\simeq H \wedge wl_\infty(G) = wl_\infty(H) \wedge nt_{\infty,1}(G) \neq nt_{\infty,1}(H)$.

Proof. 1. We proof the first statement by contradiction. Assume $u \in V(G), v \in V(H)$, two nodes with $u \ncong v$, and let i be the first iteration in which $F_i^u \not\simeq F_i^v$, but $T_{i,1}^u \simeq T_{i,1}^v$. It is easy to see that $\forall v\colon F_0^v \simeq T_{0,1}^v$, and also $\forall v\colon F_1^v \simeq T_{1,1}^v$, so $i \geq 2$.

Since i is the first iteration they differed, $F_{i-1}^u \simeq F_{i-1}^v$. Any isomorphism between F_i^u and F_i^v can only be generated from extending an isomorphism between F_{i-1}^u and F_{i-1}^v. Let φ be an arbitrary isomorphism between F_{i-1}^u and F_{i-1}^v, then, no matter how we extend it, there exists an edge in the last layer of F_i^u, that has no corresponding edge in the last layer of F_i^v (or vice versa).

If this edge is in $T_{i,1}^u$ in the last layer, then (since $T_{i,1}^u \simeq T_{i,1}^v$) there is also a corresponding edge in $T_{i,1}^v$, which implies it is also in F_i^v. This would imply $F_i^u \simeq F_i^v$, contradicting the assumption.

If this edge is not in $T_{i,1}^u$ in the last layer, the same edge must have already occurred in a previous layer in $T_{i,1}^u$. Let l be the layer that this edge first occurred. Then, $l \leq i-2$ must hold (because $k = 1$), and this edge must also occur in F_i^u, with a corresponding edge in $T_{i,1}^v$ and most importantly in F_i^v in that layer (since the trees up to $i-1$ were the same). But in unfolding trees, an edge from the original graph will be present in every layer after its first occurrence. If the

(a) Hexagon (b) Two triangles (c) Unfolding trees (d) 1-NTs

Fig. 7: Two graphs (a), (b) that cannot be distinguished by unfolding trees, but by 1-NTs. Figure (c) shows the unfolding tree F_3, which is the same for all vertices of both graphs, while (d) shows the 1-NTs of the vertices in the hexagon (left) and the triangle (right).

corresponding edge occurs in F_i^v in layer l, it also has to occur in layer i again (implying $F_i^u \simeq F_i^v$), which implies $T_{i,1}^u \not\simeq T_{i,1}^v$ and thereby contradicts the initial assumption. So $\forall G, H : wl_\infty(G) \neq wl_\infty(H) \Rightarrow nt_{\infty,1}(G) \neq nt_{\infty,1}(H)$.

2. For two triangles and a hexagon the unfolding trees look the same, but the 1-NTs do not (see Figure 7), therefore

$$\exists G, H. G \not\simeq H \wedge wl_\infty(G) = wl_\infty(H) \wedge nt_{\infty,1}(G) \neq nt_{\infty,1}(H). \qquad \square$$

The 1-NTs can also distinguish the molecules decalin and bicyclopentyl, which WL cannot distinguish. We investigate the expressiveness of 0-NTs in Appendix D.

5 Experimental Evaluation

This section provides an overview of the methods, their setup, and the datasets used in our experimental evaluation. All results were obtained using cross-validation with 10-folds. The reported accuracy and confidence intervals are averaged over three runs. We compare our method with GIN [16], ensuring fair evaluation with consistent setups and diverse datasets.

Setup. According to Equation (6), the DAG-MLP architecture is similar to GIN. The primary distinction between them is that DAG-MLP operates over DAG levels, allowing control over k-redundancy, and includes a pre-processing step for the initial feature. To ensure a fair comparison, the GIN architecture employed in our experiments will also incorporate the same pre-processing layer for the initial feature. However, it will operate directly on the original graph.

Implementation. Python was used for compatibility and implementing the approach. PyTorch and PyTorch Geometric were employed for DAG-MLP and GIN architectures respectively, with a pre-processing layer for initial node feature. The construction of the DAG was performed in a parallel manner as described in Algorithm 3, and using the hardware mentioned in Appendix I within a time range of 10 to 1500 seconds, as depicted in Figures 10 and 11.

Table 1: The accuracy and confidence interval on MUTAG for different numbers of layers and redundancy parameters k (best accuracy for each layer in bold).

Layers	k-Redundancy 0	1	2	3	4	5	6	7	8
2	**90.99**$_{\pm 3.3}$	**90.99**$_{\pm 3.3}$	87.23$_{\pm 2.9}$	-	-	-	-	-	-
3	**93.22**$_{\pm 3.9}$	92.91$_{\pm 3.7}$	90.71$_{\pm 3.4}$	90.71$_{\pm 3.4}$	-	-	-	-	-
4	92.51$_{\pm 5.6}$	92.44$_{\pm 5.0}$	**92.91**$_{\pm 3.7}$	**92.91**$_{\pm 3.7}$	**92.91**$_{\pm 3.7}$	-	-	-	-
5	93.13$_{\pm 4.6}$	**94.16**$_{\pm 5.1}$	93.42$_{\pm 3.6}$	93.64$_{\pm 3.5}$	91.89$_{\pm 3.5}$	92.40$_{\pm 3.5}$	-	-	-
6	94.14$_{\pm 4.5}$	**94.18**$_{\pm 5.1}$	93.46$_{\pm 5.0}$	92.20$_{\pm 4.8}$	92.64$_{\pm 3.3}$	92.12$_{\pm 3.4}$	92.34$_{\pm 3.5}$	-	-
7	92.97$_{\pm 5.2}$	93.70$_{\pm 4.7}$	93.92$_{\pm 4.5}$	**94.45**$_{\pm 5.4}$	93.64$_{\pm 3.7}$	94.10$_{\pm 3.1}$	93.35$_{\pm 3.1}$	93.35$_{\pm 3.1}$	-
8	**94.43**$_{\pm 5.5}$	93.11$_{\pm 5.0}$	94.38$_{\pm 4.3}$	93.19$_{\pm 5.9}$	92.91$_{\pm 4.7}$	92.91$_{\pm 4.7}$	93.13$_{\pm 3.5}$	92.86$_{\pm 3.4}$	93.33$_{\pm 3.4}$

Results. The redundancy parameter k and the number of layers l on accuracy are systematically investigated to obtain insights of their impact on the proposed method's performance.

Influence of Parameters. As shown in Table 1, the influence of k and l on accuracy is investigated through a case study on the MUTAG dataset. Combinations where $k > l$ can be disregarded, as the computation DAG remains the same as when $k = l$. The results reveal that 0 and 1-redundancy empirically provide the highest accuracy, except for cases where the number of layers is 4 and 7. In these instances, a negligible drop in accuracy, less than 1%, was observed compared to the highest accuracy achieved in each layer. This observation supports our expressivity results discussed in Section 4.4 and Appendix D.

Discussion. Table 2 presents a comparison between GIN and DAG-MLP architectures with 0-redundancy, spanning across [2, 8] layers. A notable distinction between the two architectures becomes apparent as the number of layers increases for the NCI1, ENZYMES, and MUTAG datasets. In the case of IMDB-MULTI, PROTEINS and IMDB-BINARY, both architectures exhibit similar performance, except that DAG-MLP is better as the number of layers increases.

6 Conclusion

We proposed a neural tree canonization technique and apply it to neighborhood trees, which are pruned and more expressive versions of unfolding trees used by standard MPNNs. By merging trees in a DAG, we derive compact representations that form the basis for our neural architecture DAG-MLP, which learns across DAG levels. It inherits the properties of the GIN architecture, but is provably more expressive than 1-WL when based on neighborhood trees.

Future Work. The merging of trees can be accelerated by using more refined algorithms and more compact representations can be achieved using less restrictive labeling functions. Our approach uses techniques proposed for the GIN architecture. However, other existing GNNs can be modified accordingly to operate on neighborhood trees using the concepts and algorithms established in this work.

Table 2: The accuracy and confidence intervals of GIN and DAG-MLP with 0-redundancy on graph-level classification tasks (best accuracy for each dataset in bold).

Layers	Architecture	NCI1	IMDB-MULTI	PROTEINS	IMDB-BINARY	ENZYMES	MUTAG
2	GIN	$79.60_{\pm 1.2}$	$\mathbf{54.73}_{\pm 1.5}$	$79.34_{\pm 1.8}$	$\mathbf{78.84}_{\pm 2.2}$	$49.37_{\pm 3.3}$	$83.04_{\pm 4.1}$
	DAG-MLP	$79.18_{\pm 1.1}$	$53.93_{\pm 2.3}$	$\mathbf{79.40}_{\pm 1.8}$	$77.20_{\pm 1.5}$	$49.33_{\pm 3.9}$	$90.99_{\pm 3.3}$
3	GIN	$80.17_{\pm 1.1}$	$54.62_{\pm 1.6}$	$79.16_{\pm 1.7}$	$78.72_{\pm 2.2}$	$51.03_{\pm 2.9}$	$85.97_{\pm 4.2}$
	DAG-MLP	$82.705_{\pm 0.95}$	$54.31_{\pm 1.9}$	$79.16_{\pm 1.9}$	$76.93_{\pm 2.3}$	$53.67_{\pm 3.0}$	$93.22_{\pm 3.9}$
4	GIN	$80.94_{\pm 1.2}$	$54.53_{\pm 1.6}$	$79.07_{\pm 1.5}$	$77.96_{\pm 2.0}$	$50.90_{\pm 3.2}$	$86.45_{\pm 4.2}$
	DAG-MLP	$\mathbf{83.91}_{\pm 0.8}$	$54.36_{\pm 2.2}$	$78.71_{\pm 1.8}$	$77.47_{\pm 2.0}$	$56.50_{\pm 4.0}$	$92.51_{\pm 5.6}$
5	GIN	$81.38_{\pm 1.1}$	$54.07_{\pm 1.7}$	$79.07_{\pm 1.7}$	$77.26_{\pm 2.0}$	$50.93_{\pm 3.5}$	$86.43_{\pm 4.6}$
	DAG-MLP	$83.09_{\pm 1.1}$	$\mathbf{54.73}_{\pm 3.4}$	$78.86_{\pm 1.9}$	$77.00_{\pm 1.9}$	$\mathbf{57.94}_{\pm 2.8}$	$93.13_{\pm 4.6}$
6	GIN	$80.97_{\pm 1.0}$	$52.93_{\pm 1.6}$	$78.26_{\pm 1.7}$	$77.60_{\pm 1.5}$	$49.23_{\pm 3.3}$	$88.15_{\pm 4.1}$
	DAG-MLP	$82.51_{\pm 0.8}$	$54.36_{\pm 2.4}$	$78.98_{\pm 2.1}$	$77.23_{\pm 2.4}$	$55.94_{\pm 3.4}$	$94.14_{\pm 4.5}$
7	GIN	$81.38_{\pm 1.1}$	$53.00_{\pm 1.3}$	$77.84_{\pm 1.7}$	$76.42_{\pm 2.0}$	$48.87_{\pm 2.8}$	$86.92_{\pm 4.6}$
	DAG-MLP	$81.82_{\pm 0.9}$	$53.69_{\pm 3.0}$	$78.86_{\pm 2.6}$	$77.23_{\pm 2.5}$	$54.61_{\pm 3.7}$	$92.97_{\pm 5.2}$
8	GIN	$81.40_{\pm 0.9}$	$52.00_{\pm 1.75}$	$76.44_{\pm 2.0}$	$76.74_{\pm 1.9}$	$47.73_{\pm 3.6}$	$86.67_{\pm 4.2}$
	DAG-MLP	$81.97_{\pm 0.6}$	$54.64_{\pm 1.9}$	$78.92_{\pm 2.0}$	$77.30_{\pm 2.2}$	$54.33_{\pm 3.8}$	$\mathbf{94.43}_{\pm 5.5}$

Acknowledgements. This work was supported by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG19009].

References

1. Abboud, R., Dimitrov, R., Ceylan, İ.İ.: Shortest path networks for graph property prediction. In: LoG 2022. Proceedings of Machine Learning Research, vol. 198 (2022), https://proceedings.mlr.press/v198/abboud22a.html
2. Abu-El-Haija, S., Perozzi, B., Kapoor, A., Alipourfard, N., Lerman, K., Harutyunyan, H., Steeg, G.V., Galstyan, A.: MixHop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In: ICML 2019. Proceedings of Machine Learning Research, vol. 97 (2019), http://proceedings.mlr.press/v97/abu-el-haija19a.html
3. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
4. Alon, U., Yahav, E.: On the bottleneck of graph neural networks and its practical implications. In: ICLR 2021. OpenReview.net (2021), https://openreview.net/forum?id=i80OPhOCVH2
5. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: International Conference on Machine Learning (2017)
6. Jegelka, S.: Theory of graph neural networks: Representation and learning. CoRR **abs/2204.07697** (2022)
7. Kriege, N.M., Giscard, P.L., Wilson, R.C.: On valid optimal assignment kernels and applications to graph classification. In: International Conference on Neural Information Processing Systems. NIPS (2016)
8. Li, P., Wang, Y., Wang, H., Leskovec, J.: Distance encoding: Design provably more powerful neural networks for graph representation learning. In: NeurIPS (2020), https://proceedings.neurips.cc/paper/2020/hash/2f73168bf3656f697507752ec592c437-Abstract.html

9. Li, Q., Han, Z., Wu, X.: Deeper insights into graph convolutional networks for semi-supervised learning. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16098
10. Liu, M., Gao, H., Ji, S.: Towards deeper graph neural networks. In: KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. ACM (2020). https://doi.org/10.1145/3394486.3403076
11. Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P.: Extensions of marginalized graph kernels. In: Proceedings of the twenty-first international conference on Machine learning. ACM (2004), http://doi.acm.org/10.1145/1015330.1015446
12. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer (2008), https://doi.org/10.1007/978-3-540-77978-0
13. Morris, C., Kriege, N.M., Bause, F., Kersting, K., Mutzel, P., Neumann, M.: Tudataset: A collection of benchmark datasets for learning with graphs. In: ICML 2020 Workshop on Graph Representation Learning and Beyond (2020), http://www.graphlearning.io
14. Nikolentzos, G., Chatzianastasis, M., Vazirgiannis, M.: Weisfeiler and leman go hyperbolic: Learning distance preserving node representations. In: International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 206 (2023), https://proceedings.mlr.press/v206/nikolentzos23a.html
15. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: Computational capabilities of graph neural networks. IEEE Transactions on Neural Networks **20**(1), 81–102 (2009). https://doi.org/10.1109/TNN.2008.2005141
16. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: 7th International Conference on Learning Representations, ICLR (2019)
17. Yang, Y., Wang, X., Song, M., Yuan, J., Tao, D.: SPAGAN: shortest path graph attention network. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. IJCAI (2019), https://doi.org/10.24963/ijcai.2019/569
18. You, J., Selman, J.M.G., Ying, R., Leskovec, J.: Identity-aware graph neural networks. In: Thirty-Fifth AAAI Conference on Artificial Intelligence. AAAI Press (2021), https://ojs.aaai.org/index.php/AAAI/article/view/17283

Algorithm 1 Merging trees

function MERGE(set of trees $\mathcal{T}$, labeling L)
  $D \leftarrow$ empty DAG ▷ start with empty DAG
  for each $T \in \mathcal{T}$ **do**
    $\Psi \leftarrow \emptyset$ ▷ map from nodes of T to D
    $\mathcal{R} \leftarrow l(T)$ ▷ start with leaves
    while $\mathcal{R} \neq \emptyset$ **do**
      for each $v \in \mathcal{R}$ **do**
        ADD(D, v, Ψ, L) ▷ add node
      $\mathcal{N} \leftarrow \bigcup_{v \in \mathcal{R}} \{p \mid p \in p(v) \wedge \forall c \in \text{chi}(p) : \Psi(c) \text{ is defined}\}$
      ▷ add processable ancestors next
      $\mathcal{R} \leftarrow \mathcal{N}$
  return D

function ADD(DAG D, node v, map Ψ, labeling L)
  if $\Psi(v)$ is defined **then** ▷ node has already been processed once
    return
  find $v_2 \in V(D)$ with $L(v) = L(v_2) \wedge \{\Psi(c) \mid c \in \text{chi}(v)\} = \text{chi}(v_2)$
  ▷ try to find (unique) corresponding node in D
  if $\nexists v_2$ **then**
    add new node v_2 with $L(v) = L(v_2)$ to D ▷ add new node if needed
  $\Psi(v) \leftarrow v_2$

A Merging Trees - Algorithm

Algorithm 1 describes how to merge a set of trees $\{T_1, \ldots, T_n\}$ into a DAG under a labeling function $L\colon \bigcup_{i \in \{1,\ldots,n\}} V(T_i) \to \mathcal{O}$, where $\mathcal{O}$ is some arbitrary labeling, that fulfills $\forall v \in \bigcup_{i \in \{1,\ldots,n\}} V(T_i)\colon \forall c_j, c_k \in \text{chi}(v)\colon j \neq k \Rightarrow L(c_j) \neq L(c_k)$. Each tree is merged into the DAG separately.

All substructures that are isomorphic under L are merged, by merging nodes bottom-up: Starting at the leaves of the tree, that is added, each leaf is either merged into an existing node in the DAG or (if no node matches) the leaf is added to the DAG as a new node. After a node is added to the DAG, all parents of that node in the original tree are added to a list containing the next nodes to be added, provided all their children have already been added to the DAG.

The set of possible matches can be restricted to the set of nodes, that are parents of all the nodes the children were mapped to.

B Generating the Edge Partition

Algorithm 2 extracts the edges needed for the layers in reverse order. The algorithm takes a DAG as input and then starts by adding all edges that go to a root to the first edge partition (layer). The roots are then "processed" and can be disregarded for the remaining iterations. For each of those edges, if the source node has no parents left that need to be processed, it is added to the list

Algorithm 2 Creation of the edge partition

```
function buildEdgePartitions(DAG D)
    E ← {}                                              ▷ list of edge partitions
    C ← r(D)                                            ▷ start with roots
    P ← ∅                                               ▷ nodes processes so far
    while ∃c ∈ C with c ∉ l(D) do
        E ← ∅                                           ▷ set of edges in this layer
        C' ← ∅                                          ▷ set of next nodes to process
        for each c ∈ C do
            P = P ∪ {c}
            E = E ∪ {uc | u ∈ chi(c)}
            C' = C' ∪ {u ∈ chi(c) | ∀v ∈ p(u) ⇒ v ∈ P}   ▷ add if all parents processed
        C ← C'
        E append E
    reverse E                                           ▷ layers were generated from the last to first
    return E
```

of nodes to process in the next iteration. This is repeated, until no nodes that have incoming edges need to be processed. Each edge is used exactly once in the computation.

C Parallel Construction of DAG

The algorithm constructs a DAG in parallel using a set of nodes, V, and edges, E, wherein the k-NTs were built concurrently for each node in the graph. Subsequently, the pairs of the k-NT are merged in parallel, resulting in a merged set of k-NTs. This process was iteratively repeated, with the merged k-NTs being divided into two pairs, and the aforementioned steps being applied until only two merged k-NTs remained, which were subsequently merged sequentially.

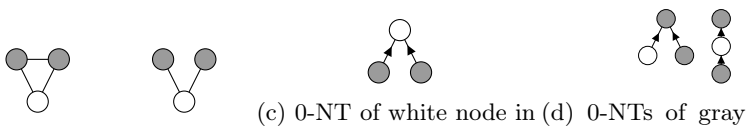
D Expressiveness of 0-NTs as Node Invariants

Theorem 1 shows that 1-NTs are more expressive as a node invariant (and in turn graph invariant) than unfolding trees. While the 0-NTs can also distinguish the nodes of the two graphs in the example of Figure 7 (and the famous example of decalin and bicyclopentyl), as a node invariant, they are not more expressive than unfolding trees in every case. Figure 8 shows an example, where the unfolding trees of two (non-isomorphic) vertices differ, but their 0-NTs do not. Looking at the tree sets, however, $nt_{\infty,0}$ can also distinguish those two graphs. It remains future work to delineate the expressivity of $nt_{\infty,0}$ and wl_{∞}.

Algorithm 3 Parallel Reduction Algorithm for DAG Construction

function ConstructDAG(set of nodes V, set of edges E)
  DAG $\leftarrow$ {}
  in parallel for each $v \in V$ **do**
    $\text{DAG}_v \leftarrow$ Build k-NT for node v ▷ using Definition 1
  pDAG $\leftarrow$ BuildPairsDAG(V, DAG)
  while || pDAG || ≥ 2 **do**
    in parallel for each $\text{DAG}_{v,u} \in$ pDAG **do**
      $\text{DAG}_{v,u} \leftarrow$ merge(DAG_v, DAG_u)
    pDAG $\leftarrow$ BuildPairsDAG(V, DAG)
  DAG $\leftarrow$ merge(pDAG_0, pDAG_1) ▷ using Algorithm 1
  return DAG

function BuildPairsDAG(set of nodes V, set of k-NT *DAG*)
  pDAG $\leftarrow$ {}
  for each two nodes $(v, u) \in V$ **do**
    $\text{pDAG}_{v,u} \leftarrow (\text{DAG}_v, \text{DAG}_u)$
  return pairs



(a) A triangle (b) A star (c) 0-NT of white node in either graph. (d) 0-NTs of gray nodes (triangle left, star right).

Fig. 8: Graphs with vertices, not distinguishable by 0-NTs, but by unfolding trees.

E Datasets

Table 3 provides an overview of the datasets utilized in our evaluation, along with their corresponding characteristics. These datasets are sourced from the TUDataset [13].

F Efficiency

The size of the DAG has a huge influence on the computation efficiency of our method. Tables 4 and 5 show the number of nodes and edges in the computation DAG of differing size and k-redundancy on dataset MUTAG. As the number of layers increases, the DAG grows in size initially. This is expected, since a larger neighborhood around the vertices is explored. For a larger number of layers (the lower the k, the earlier) the size of the DAG decreases. This means a more efficient computation and can be explained, by whole subgraphs of the original graph being fully explored. To illustrate this Figure 9 depicts an example graph G and its 0-NT for three layers. In the case of a single layer, the 0-NT is identical to the original unfolding tree. With two layers, the number of nodes and edges increases,

Table 3: Summary of dataset characteristics from TUDataset [13]. The table provides information on the dataset name, number of graphs ($|\mathbf{G}|$), average number of nodes ($\overline{|\boldsymbol{V}|}$), average number of edges ($\overline{|\boldsymbol{E}|}$), number of classes, number of initial features, and maximum diameter (Max Diam.) for each dataset.

Name	$\|\mathbf{G}\|$	$\overline{\|\boldsymbol{V}\|}$	$\overline{\|\boldsymbol{E}\|}$	# **Classes**	# **Feature**	**Max Diam.**
NCI1	4110	29.87	32.3	2	37	45
IMDB-MULTI	1500	13	65.94	3	89	2
PROTEINS	1113	39.06	72.82	2	3	64
IMDB-BINARY	1000	19.77	96.53	2	136	2
ENZYMES	600	32.63	62.14	6	3	37
MUTAG	188	17.93	19.79	2	7	15

since for example for the blue vertex of the original graph, the neighborhood in the direction of the green node is fully explored, while for the yellow and red vertices, this neighborhood is still missing the purple vertex. However, with three layers the number of nodes decreases, since the neighborhoods are fully explored then. This leads to a more efficient representation for a higher number of layers.

G Running Time

The process of generating merge DAGs for neighborhood trees with varying depths (layers) for IMDB-MULTI and IMDB-BINARY datasets reveals an intriguing finding: regardless of the number of layers involved, the execution time appears to converge to a similar value. This phenomenon can be attributed to the remarkable property shared by both datasets, namely their equal diameter of 2.

H Hyper-Parameter

The hyper-parameters were fixed as following: hidden dimension: 64, batch size: 32, learning rate (lr): 0.001, learning rate decay factor: 0.5, learning rate decay step size: 50, epochs: 500, and weight decay: 0.0005.

I Hardware

The hardware used includes an Intel Xeon Gold 6130 processor (x86_64 architecture) with 64 cores (2 sockets, 2 threads per core) and 251GB system memory, and two Tesla P100-PCIE-12GB GPUs.

Table 4: The average number of nodes in the computation DAG for the MUTAG dataset was compared across different numbers of layers and k-redundancy.

		MUTAG Average Number of Nodes								
Number of layers	**2**	75.6	75.6	**56.6**	-	-	-	-	-	-
	3	108.9	108.2	**75.4**	**75.4**	-	-	-	-	-
	4	143.4	147.1	214.3	216.1	**94.3**	-	-	-	-
	5	159.7	168.9	297.7	301.9	126.7	**113.1**	-	-	-
	6	156.4	167.6	371.5	383.9	428.7	425.5	**132.0**	-	-
	7	**143.9**	157.7	401.8	429.3	569.7	571.1	208.5	150.8	-
	8	**131.6**	143.7	393.1	428.9	684	706	724.1	703.1	169.7
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
		k-Redundancy								

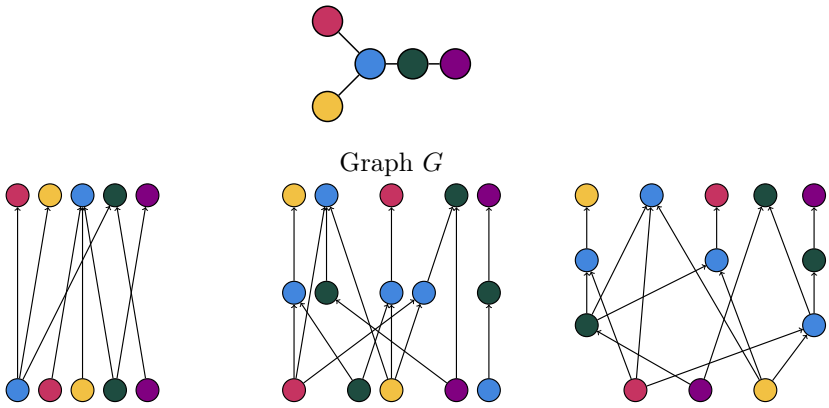


Fig. 9: 0-NT DAGs for the original graph G, shown with three different layers.

Table 5: The average number of edges in the computation DAG for the MUTAG dataset was compared across different numbers of layers and k-redundancy.

	MUTAG Average Number of Edges								
Number of layers									
2	102.7	102.7	**83.3**	-	-	-	-	-	-
3	153.3	153.1	**125.0**	**125.0**	-	-	-	-	-
4	202	209.7	430	434.9	**166.7**	-	-	-	-
5	224.5	241.1	605	618.8	243.6	**208.4**	-	-	-
6	**221.6**	241.3	751.4	783.1	935.5	929.3	250.0	-	-
7	**206.9**	229	809.2	871.9	1238.8	1254.1	440.9	291.7	-
8	**190.9**	210.1	792.8	873.6	1471.7	1535.3	1630.3	1580.1	333.4
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	k-Redundancy								

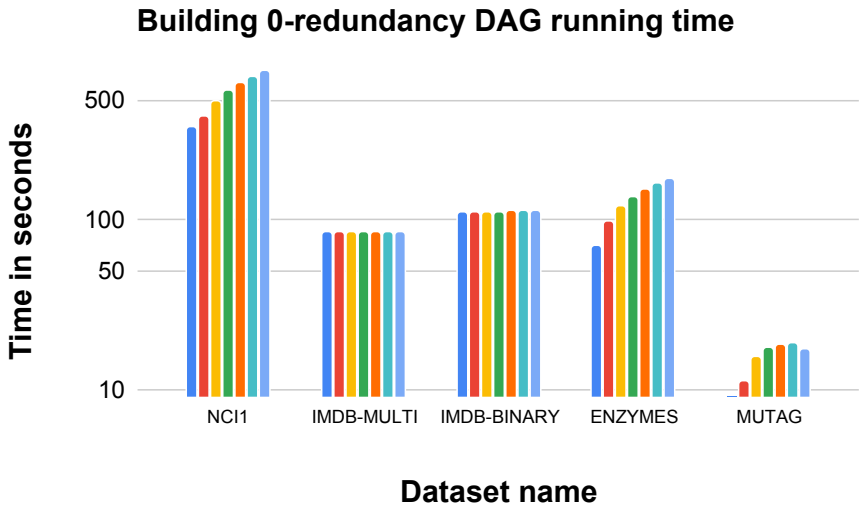


Fig. 10: Running time for building the 0-redundant DAG in parallel.

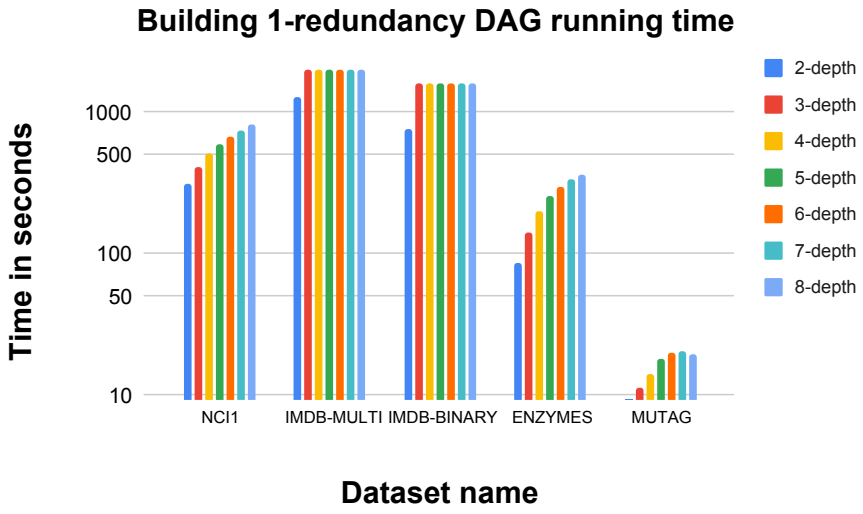


Fig. 11: Running time for building the 1-redundant DAG in parallel.