

On the Two Sides of Redundancy in Graph Neural Networks

Franka Bause^{1,2}, Samir Moustafa^{1,2}, Johannes Langguth³,
Wilfried N. Gansterer¹, and Nils M. Kriege^{1,4}

¹Faculty of Computer Science, University of Vienna, Vienna, Austria

²UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria

³Simula Research Laboratory, Oslo, Norway

⁴Research Network Data Science, University of Vienna, Vienna, Austria
{firstname.lastname}@univie.ac.at

Abstract

Message passing neural networks iteratively generate node embeddings by aggregating information from neighboring nodes. With increasing depth, information from more distant nodes is included. However, node embeddings may be unable to represent the growing node neighborhoods accurately and the influence of distant nodes may vanish, a problem referred to as oversquashing. Information redundancy in message passing, i.e., the repetitive exchange and encoding of identical information amplifies oversquashing. We develop a novel aggregation scheme based on neighborhood trees, which allows for controlling redundancy by pruning redundant branches of unfolding trees underlying standard message passing. While the regular structure of unfolding trees allows the reuse of intermediate results in a straightforward way, the use of neighborhood trees poses computational challenges. We propose compact representations of neighborhood trees and merge them, exploiting computational redundancy by identifying isomorphic subtrees. From this, node and graph embeddings are computed via a neural architecture inspired by tree canonization techniques. Our method is less susceptible to oversquashing than traditional message passing neural networks and can improve the accuracy on widely used benchmark datasets.

1. Introduction

Graph neural networks (GNNs) have emerged as the dominant approach for machine learning on graph data, with the class of message passing neural networks (MPNNs) [12] being widely-used. These networks update node embeddings layer wise by combining the current embedding of a node with those of its neighbors, involving learnable parameters. Suitable

neural architectures, which admit a parametrization such that each layer represents an injective function uniquely encoding the input, have the same expressive power as the Weisfeiler-Leman algorithm [37]. The Weisfeiler-Leman algorithm distinguishes two nodes if and only if the unfolding trees representing their neighborhoods are non-isomorphic. These unfolding trees correspond to the computational trees of MPNNs [30, 15]. Hence, nodes with isomorphic unfolding trees will obtain the same embedding, while for nodes with non-isomorphic unfolding trees, there exist parameters such that their embeddings differ. This implies that deeper unfolding trees lead to more expressive methods. Despite this theoretical connection, shallow MPNNs are often favored in practice. Challenges arise from the observed convergence of node embeddings for deep architectures, referred to as *oversmoothing* [20, 21], and the issue of *oversquashing* [5], where the neighborhood of a node grows exponentially with the number of layers, and therefore, cannot be supposed to be accurately represented by a fixed-sized embedding. Recently, oversquashing has been investigated by analyzing the sensitivity of node embeddings to the initial features of distant nodes, relating the phenomenon to the *graph curvature* [34], the *effective resistance* [7] and the *commute time* [13]. On this basis several graph rewiring strategies have been proposed to mitigate oversquashing [34, 7].

We address the problem of oversquashing by modifying the message passing scheme for eliminating the encoding of repeated information. For example, in an undirected graph, when a node sends information to its neighbour, future messages sent back via the same edge will contain the exact information previously sent, leading to redundancy. In the context of walk-based graph learning this problem is well-known and referred to as *tottering* [22]. Recent work by Chen et al. [8] established a first result formalizing the relation between redundancy and oversquashing by sensitivity analysis. Several recent GNNs replace the walk-based aggregation by mechanisms based on simple or shortest paths reporting promising results [2, 26, 16]. PathNNs [26] and RFGNN [16] are closely related approaches, defining path-based trees for nodes and employing custom aggregation schemes. However, these methods suffer from high computational costs compared to standard MPNNs and often have an exponential time complexity. The crucial advantage of MPNNs is the regular structure of aggregations applied through all layers, while reducing information redundancy leads to a less regular structure, rendering it challenging to exploit computational redundancy.

Our contribution. We systematically explore the issue of information redundancy within MPNNs and introduce principled techniques to eliminate superfluous messages. Our investigation is based on the implicit tree representation used by both MPNNs and the Weisfeiler-Leman algorithm. We first develop a neural tree canonization approach that systematically processes trees in a bottom-up fashion and extend it to directed acyclic graphs (DAGs). To exploit computational redundancy, we merge multiple trees representing node neighborhoods into a single DAG identifying isomorphic subtrees. Our approach, termed DAG-MLP, recovers the computational graph of MPNNs for unfolding trees, while avoiding redundant computations in the presence of symmetries. We employ the canonization technique on *neighborhood trees*, which are derived from unfolding trees by eliminating nodes that appear multiple times. We show that neighborhood trees allow distinguishing nodes and graphs that are indistinguishable by

Table 1: Time complexity of preprocessing, size of computation graph and expressivity of our method compared to related work. n : number of nodes, m : number of edges, b : maximum node degree, K : path length, h : tree height, L : number of layers, and $m_2 = 0.5 \sum_{v \in V} |N_2(v)|$.

Method	Preprocessing	Size Comp. Graph/Runtime	Expressivity
PathNet	$O(mb)$	$O(2^L(m + m_2))$	n/a
PathNN-SP	$O(nb^K)$	$O(nbK)$	incomparable
PathNN-SP+	$O(nb^K)$	$O(nbK)$	> 1-WL
RFGNN	$O(n!/(n-h-1)!)$	$O(n!/(n-h-1)!)$	incomparable
DAG-MLP (0-/1-NTs)	$O(nm)$	$O(nm)$	incomparable

the Weisfeiler-Leman algorithm. The DAGs derived from neighborhood trees have size at most $O(nm)$ for input graphs with n nodes and m edges making the approach computational feasible. We formally show by sensitivity analysis that our approach reduces oversquashing. Our approach achieves high accuracy across various node and graph classification tasks.

2. Related Work

The graph isomorphism network (GIN) [37] is an MPNN that generalizes the Weisfeiler-Leman algorithm, achieving its expressive power. The limited expressivity of simple MPNNs has led to an increased interest in researching more powerful architectures, such as encoding graph structure as additional features or modifying the message passing procedure. Shortest Path Networks [2] use multiple aggregation functions for different shortest path lengths, allowing direct communication with distant nodes. While this might help mitigate oversquashing, information about the structure of the neighborhood can still not be represented adequately and the gain in expressivity is limited. Distance Encoding GNNs [19] encode the distances of nodes to a set of target nodes. While being provably more expressive than the standard WL algorithm, the approach is limited to solving node-level tasks, as the encoding depends on a fixed set of target nodes and has not been employed for graph-level tasks. MixHop [3] concatenates results from activation functions for each neighborhood, but in contrast to Shortest Path Networks [2], the aggregation is based on normalized powers of the adjacency matrix, not shortest paths, which fails to solve the problem of redundant messages. SPAGAN [38] proposes a path-based attention mechanism, sampling shortest paths and using them as features. However, a theoretical investigation is lacking and the approach utilizes only one layer. Short-rooted random walks in [33] capture long-range dependencies, but have notable limitations due to sampling paths. The evaluation is restricted to node classification datasets and an extensive study of their expressive power is lacking. IDGNN [39] tracks the identity of the root node in unfolding trees, achieving higher expressivity than 1-WL, but failing to reduce redundant information aggregation. PathNNs [26] define path-based trees and a custom aggregation scheme, but overlook exploiting computational redundancy. RFGNNs [8] aim to reduce redundancy by altering the

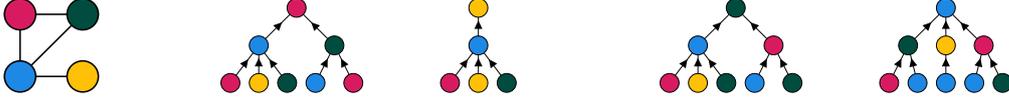


Figure 1: Graph G and its unfolding trees F_2^v for all $v \in V(G)$.

message flow to only include each node (except for the root node) at most once in each path of the computational tree. While this reduces redundancy to some extent, nodes and even the same subpaths may repeatedly occur in the computational trees. The redundancy in computation is not addressed resulting in a highly inefficient preprocessing and computation, limiting the method to a maximum of 3 layers in the experiments. We discuss further differences between our approach and RFGNN in Appendix A.

These architectures lack thorough investigation of their expressivity and connections to other approaches. Importantly, they do not explicitly investigate both types of redundancy in MPNNs – redundancy in the information flow and in computation. We compare the time complexity, as well as the expressivity of our method DAG-MLP and other relevant methods in Table 1 and further discuss it in Section 4.5.

3. Preliminaries

In this section, we provide an overview of essential definitions and the notation used throughout this article, accompanied by the introduction of 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 $\mu: V \rightarrow X$ and $\nu: E \rightarrow X$ assigning arbitrary attributes to the vertices and edges, respectively.¹ An edge from u to v is denoted by uv , and in undirected graphs $uv = vu$. The vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The *neighbors* (or in-neighbors) of a vertex $u \in V$ are denoted by $N(u) = \{v \mid vu \in E\}$, and the *out-neighbors* of a vertex $u \in V$ are denoted by $N_o(u) = \{v \mid uv \in E\}$. A *multigraph* is a graph, where E is a multiset, allowing multiple edges between a pair of vertices. Two graphs G and H are isomorphic, denoted by $G \simeq H$, if there exists a bijection $\phi: V(G) \rightarrow V(H)$, such that $\forall u, v \in V(G): \mu(v) = \mu(\phi(v)) \wedge uv \in E(G) \Leftrightarrow \phi(u)\phi(v) \in E(H) \wedge \forall uv \in E(G): \nu(uv) = \nu(\phi(u)\phi(v))$. We refer to ϕ as an *isomorphism* between G and H .

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* ($r(T)$), in which $\forall v \in V(T) \setminus r(T) : |N_o(v)| = 1$. For $v \in V(T) \setminus r(T)$ the *parent* $p(v)$ is the unique vertex $u \in N_o(v)$, and $\forall v \in V(T)$ the *children* are defined as $\text{chi}(v) = N(v)$. We refer to vertices without incoming edges as *leaves*, denoted by $l(T) = \{v \in V(T) \mid \text{chi}(v) = \emptyset\}$. Conceptually an in-tree is a directed tree, in which there is a

¹Edge attributes are not considered in the following for clarity of presentation, though the proposed methods can be extended to incorporate them.

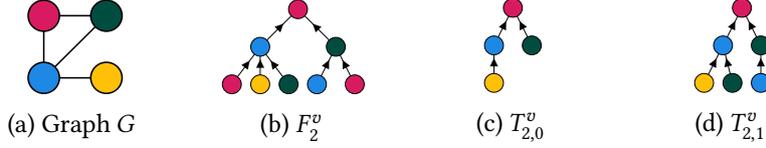


Figure 2: Graph G and the unfolding, 0- and 1-redundant neighborhood trees of height 2 of vertex v (vertex in the upper left of G).

unique directed path from each vertex to the root [25]. In our paper, we only consider in-trees and will therefore refer to them simply 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 without 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 $p(v) = N_o(v)$. The height hgt of a node v is the length of the longest path from any leaf to v : $\text{hgt}(v) = 0$, if $v \in l(D)$ and $\text{hgt}(v) = \max_{c \in \text{chi}(v)} \text{hgt}(c) + 1$, otherwise. The height of a DAG D is defined as $\text{hgt}(D) = \max_{v \in V(D)} \text{hgt}(v)$. 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 and Message Passing Neural Networks. The 1-dimensional Weisfeiler-Leman (WL) algorithm, also known as *color refinement*, starts with vertices having a color corresponding to their label (or a uniform coloring for unlabeled vertices). In each iteration the vertex color is updated based on the multiset of colors of its neighbors according to

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

where h is an injective function, typically using integers to represent colors.

The color of a vertex encodes its neighborhood through a tree T , which may contain multiple representatives of each vertex. Let $\phi: V(T) \rightarrow V(G)$ 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, as illustrated in Figure 1. The unfolding trees F_i^v and F_i^w of two vertices v and w are isomorphic if and only if $c_{\text{wl}}^{(i)}(v) = c_{\text{wl}}^{(i)}(w)$.

Message passing neural networks such as GIN [37] can be seen as a neural variant of the Weisfeiler-Leman algorithm. The embedding of a vertex v in layer i of GIN is defined as

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

where the initial features $x_0(v)$ are usually acquired by applying a multi-layer perceptron (MLP) to the vertex features.

4. Non-Redundant Graph Neural Networks

We propose to restrict the information flow in message passing to regulate redundancy through the use of k -redundant neighborhood trees. We first develop a neural tree canonization technique, and obtain an MPNN via its application to unfolding trees. Subsequently, we explore computational methods on graph level, reusing information computed for subtrees and derive a customized GNN architecture. Finally, we prove that k -redundant neighborhood trees and unfolding trees are incomparable regarding their expressivity on node-level.

4.1. Removing Information Redundancy

As previously discussed, two vertices obtain the same WL color if and only if their unfolding trees are isomorphic. This concept directly carries over to message passing neural networks and their computational tree [30, 15]. However, unfolding trees were mainly used as tools in expressivity analysis and as a conceptual framework for explaining mathematical properties in graph learning [18, 29]. We propose a novel perspective on MPNNs through tree canonization. From this perspective, we derive a non-redundant GNN architecture based on neighborhood trees.

In their classical textbook, Aho, Hopcroft, and Ullman [4, Section 3.2] describe a linear time isomorphism test for rooted unordered trees, detailed in Appendix C. We give a high-level description to establish the foundation for our neural variant without focusing on the running time. The algorithm proceeds in a bottom-up manner, assigning integers $c_{\text{ahu}}(v)$ to each node v in the tree. The function f injectively maps a pair consisting of an integer and a multiset of integers to a new, unused integer. Initially, all leaves v are assigned integers $c_{\text{ahu}}(v) = f(\mu(v), \emptyset)$ based on their label $\mu(v)$. Then internal nodes are processed level-wise in a bottom-up manner, ensuring that whenever a node is processed, all its children have been considered. Hence, the algorithm computes for all nodes v of the tree

$$c_{\text{ahu}}(v) = f(\mu(v), \{\{c_{\text{ahu}}(u) \mid u \in \text{chi}(v)\}\}). \quad (2)$$

This process ensures the unique representation of non-isomorphic trees, serving as the foundation of our neural tree canonization technique.

GNNs via unfolding tree canonization. We combine Eq. (2) and the definition of unfolding trees, denoting the root of an unfolding tree of height i of a vertex v by n_v^i . This yields

$$c_{\text{ahu}}(n_v^i) = f(\mu(n_v^i), \{\{c_{\text{ahu}}(n_u^{i-1}) \mid n_u^{i-1} \in \text{chi}(n_v^i)\}\}) = f(\mu(v), \{\{c_{\text{ahu}}(n_u^{i-1}) \mid u \in N(v)\}\}). \quad (3)$$

By implementing the function f using a suitable neural architecture and replacing its codomain with embeddings in \mathbb{R}^d , we readily obtain a GNN based on our canonization approach. The key difference to standard GNNs is that the first component of the pair in Eq. (3) is the initial vertex feature instead of the embedding from the previous iteration. Utilizing the technique proposed by Xu et al. [37] and replacing the first addend in Eq. (1) with the initial embedding, we formulate the *unfolding tree canonization GNN*

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

It is established that MPNNs cannot distinguish two vertices with the same WL color or unfolding tree. Given that the function $c_{\text{ahu}}(n_v^i)$ uniquely represents the unfolding tree for an injective function f , realizable by Eq. (4) [37], we infer the following proposition.

Proposition 1. Unfolding tree canonization GNNs, as defined in Eq. (4), are as expressive as GIN, as defined in Eq. (1).

Despite the equivalence in expressivity, the canonization-based approach avoids redundancy since $x_{i-1}(v)$ represents the entire unfolding tree rooted at v of height $i - 1$, while using the initial vertex features $x_0(v)$ is sufficient. We proceed by investigating redundancy within unfolding trees themselves.

GNNs via neighborhood tree canonization. We leverage the concept of neighborhood trees to manage redundancy in unfolding trees.² A k -redundant neighborhood tree (k -NT) $T_{i,k}^v$ is derived from the unfolding tree F_i^v by removing all subtrees with roots that occurred more than k levels before (seen from root to leaves). Here, $\text{depth}(v)$ denotes the length of the path from v to the root, and $\phi(v)$ denotes the original vertex in $V(G)$ represented by v in the unfolding or neighborhood tree.

Definition 2 (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

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

Figures 2 and 3 provide examples of unfolding and neighborhood trees. It is worth noting that for $k \geq i$ the k -redundant neighborhood tree is equivalent to the WL unfolding tree.

We can directly apply the neural tree canonization technique to neighborhood trees. However, a simplifying expression based on the neighbors in the input graph, as given by Eq. (3) for unfolding trees, is not possible for neighborhood trees. Therefore, we explore techniques to systematically exploit computational redundancy.

²In a parallel work, neighborhood trees were investigated for approximating the graph edit distance [6].

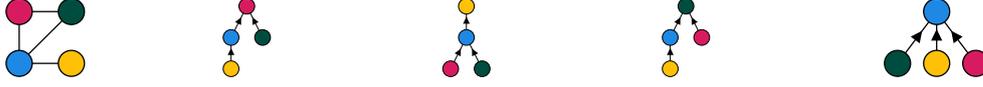


Figure 3: Graph G and its 0-NTs $T_{2,0}^v$ for all $v \in V(G)$.

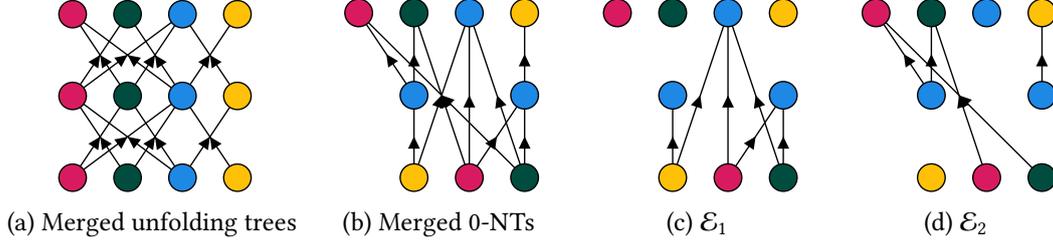


Figure 4: Computation DAGs for unfolding (a) and 0-NTs (b) of height 2 of graph G . And edges in the different layers of the merge DAG of 0-NTs (c), (d).

4.2. Removing Computational Redundancy

The computation DAG of an MPNN involves the embedding of a set of trees representing the vertex neighborhoods of a single or multiple graphs. Results computed for one tree can be reused for others by identifying isomorphic substructures, thereby minimizing computational redundancy. We first describe how to merge trees in a general context and then discuss its application to unfolding and neighborhood trees.

Merging trees into a DAG. The neural tree canonization approach developed in the last section can be directly 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. A detailed description of a neural architecture is postponed to Section 4.3.

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, $\xi: \{1, \dots, n\} \rightarrow V(D)$ 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 instead of a forest.

We propose an algorithm that builds a merge DAG by successively adding trees to an initially empty DAG, creating new nodes only when necessary. Our approach maintains a canonical labeling for each node of the DAG and computes a canonical labeling for each node of the tree to be added using the AHU algorithm (cf. Appendix C). Then, the tree is processed starting at the root. If the canonical labeling of the root is present in the DAG, then the algorithm terminates.

Otherwise, the subtrees rooted at its children are inserted into the DAG by recursive calls. Finally, the root is created and connected to the representatives of its children in the DAG. We introduce a node labeling $L: V_T \rightarrow \mathcal{O}$ used for tree canonization, where $V_T = \bigcup_{i=1}^n V(T_i)$ and \mathcal{O} an arbitrary set of labels, refining the original node attributes, i.e., $L(u) = L(v) \Rightarrow \mu(u) = \mu(v)$ for all u, v in V_T . When \mathcal{O} consists of integers from the range 1 to $|V_T|$, the algorithm runs in $O(|V_T|)$ time (see Appendix E for details). When two siblings that are the roots of isomorphic subtrees are merged, this leads to parallel edges in the DAG. Parallel edges can be avoided by using a labeling satisfying $L(u) = L(v) \Rightarrow \mu(u) = \mu(v) \wedge p(u) \neq p(v)$ for all u, v in V_T .

Unfolding trees and k -NTs can grow exponentially in size with increasing height. However, this is not case for merge DAGs obtained by the algorithm described above, as we will show below. Moreover, we can directly generate DAGs of size $O(m \cdot (k + 1))$ representing individual k -NTs with unbounded height in a graph with m edges (see Appendix D for details).

Merging unfolding trees. Merging the 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 graph from Figure 1. The roots in this DAG correspond to the representation of the vertices after aggregating information from the lower layers. Each vertex 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 allows computation based on the adjacency matrix widely-used for MPNNs, it involves the encoding of redundant information. Our method has the potential to compress the computational DAG further by using the less restrictive labeling $L = \mu$, leading to a DAG where at layer i all vertices u, v with $c_{\text{wl}}^{(i)}(u) = c_{\text{wl}}^{(i)}(v)$ are represented by the same node. This compression appears particularly promising for graphs with symmetries.

Merging neighborhood trees. When merging k -redundant neighborhood trees in the same way using the labeling $L = \mu$ (or $L = \phi$ to avoid parallel edges), it results in a computation DAG with a more irregular structure, as illustrated in Figure 4b. Firstly, there might be multiple nodes at the same level representing the same original vertex. Secondly, the adjacency matrix of the original graph cannot be used to propagate the information. A straightforward upper bound on the size of the merge DAG for a graph with n nodes and m edges is $O(nmk + nm)$.

We apply the neural tree canonization approach to the merge DAG in a bottom-up fashion, starting from the leaves and progressing to the roots. Each edge is used exactly once in this computation. Let $D = (\mathcal{V}, \mathcal{E})$ be a merge DAG. The nodes can be partitioned based on their height, leading to $\mathcal{L}_i = \{v \in \mathcal{V} \mid \text{hgt}(v) = i\}$. This induces an edge partition $\mathcal{E}_i = \{uv \in \mathcal{E} \mid v \in \mathcal{L}_i\}$, where all edges with the same end node v are in the same layer, and all incoming edges of children of v belong to a previous layer. Note that, since \mathcal{L}_0 contains all leaves of the DAG, there is no \mathcal{E}_0 . Figures 4c and 4d depict the edge sets \mathcal{E}_1 and \mathcal{E}_2 for the example merge DAG illustrated in Figure 4b.

4.3. Non-Redundant Neural Architecture (DAG-MLP)

We propose a neural architecture to compute embeddings for nodes in a merge DAG, allowing to retrieve embeddings of the contained trees from their roots. The process involves a preprocessing step that transforms the node labels, using MLP_0 to map them to an embedding space of fixed dimensions. Subsequently, an MLP_i is used to process nodes at each layer \mathcal{L}_i .

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

The DAG-MLP can be computed through iterated matrix-vector multiplication analogous to standard GNNs. Let \mathbf{L}_i be a square matrix with ones on the diagonal at position j if $v_j \in \mathcal{L}_i$, and zeros elsewhere. Let \mathbf{E}_i represent the adjacency matrix of $(\mathcal{V}, \mathcal{E}_i)$, and let \mathbf{F} denote the node features of \mathcal{V} , corresponding to the initial node labels. The transformed features \mathbf{F}' are obtained through MLP_0 , and $\mathbf{X}^{[i]}$ represents the updated embeddings at layer i of the DAG.

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

In the above equation, MLP_i is applied to the rows associated with nodes in \mathcal{L}_i . The embeddings $\mathbf{X}^{[i]}$ are initialized to zero for inner nodes and computed level-wise. To preserve embeddings from all previous layers, we add $\mathbf{X}^{[i-1]}$ during the computation of $\mathbf{X}^{[i]}$. Suppose the merge DAG (D, ξ) contains the trees $\{T_1, \dots, T_n\}$. We obtain a node embedding $\mathbf{X}_{\xi(i)}^{[n]}$ for each tree T_i with $i \in \{1, \dots, n\}$. This approach allows for obtaining the final embedding for a vertex by using a single tree (Fixed Single-Height) or combining trees of different heights, for example all NTs of size up to a certain maximum (Combine Heights). Further details on the resulting architecture are described in Appendix F.

4.4. Expressivity of k -NTs

Here, we investigate how expressive k -NTs are compared to unfolding trees. While it is evident that k -NTs are a node invariant, providing the same result for nodes that can be mapped to each other by an isomorphism or automorphism, they might also produce the same results for nodes that cannot. This means that, similar to unfolding trees, they are not a complete node invariant.

We show that there are vertices that 1-WL cannot distinguish, but k -NTs can, and vice versa, proving that both methods are incomparable regarding their expressivity on node level. We

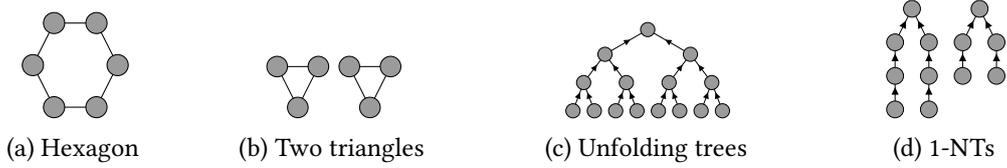


Figure 5: Two graphs (a), (b) that cannot be distinguished by unfolding trees, but by k -NTs. Figure (c) shows the unfolding tree F_3 , which is the same for all vertices of both graphs, while (d) shows the 1-NTs of the vertices in the hexagon (left) and the triangle (right).

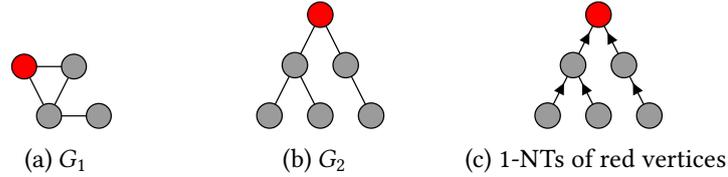


Figure 6: Two graphs (a), (b) in which the red vertices can be distinguished by unfolding trees, but not by k -NTs. Figure (c) shows the 1-NTs of the red vertices, which are the same. However, G_1 and G_2 can be distinguished by their multisets of 1-NTs.

further conjecture that, while k -NTs cannot distinguish certain vertices that 1-WL can, they can still distinguish the graphs containing such vertices, making k -NTs more expressive on the graph level.

Theorem 3. The expressivity of k -NT and unfolding trees is incomparable, i.e.,

1. $\exists u, v: F_\infty^u = F_\infty^v \wedge T_{\infty,k}^u \neq T_{\infty,k}^v$.
2. $\exists u, v: F_\infty^u \neq F_\infty^v \wedge T_{\infty,k}^u = T_{\infty,k}^v$

Proof. We prove the statement by giving concrete examples. In Figure 5 an example for 1. is given: as commonly known, the Weisfeiler-Leman algorithm is unable to distinguish the two graphs, indicating identical unfolding trees of the vertices in the two graphs. However, for any k the k -NT of the vertices will differ for $h \geq k + 2$. In Figure 6 an example for 2. (for 1-NTs) is given: the unfolding trees of the two marked vertices differ, while the 1-NTs are identical. This serves as an instance where 1-WL can distinguish vertices that k -NTs cannot. \square

We have shown that on node level, the expressivity of k -NTs and unfolding trees is incomparable. However, the examples, where k -NTs fail to distinguish nodes that 1-WL can, can actually be distinguished on the graph level. This arises from the fact, that the graphs have a different number of vertices and the k -NTs of the other nodes differ.

4.5. Computational Complexity, Expressivity and Oversquashing

The DAG representing the k -NT of a single vertex has a size in $O(mk + m)$, where m is the number of edges in the graph. The lexicographic encoding and merging of k -NTs to generate the DAG can be done in time linear in its size. A trivial upper bound on the size of the merge DAG of a graph with n nodes and m edges is $O(nmk + nm)$. Overall, this means that preprocessing can be done in $O(nmk)$ time, where k can be considered constant. For 0- and 1-NTs, we obtain time $O(nm)$. Table 1 compares the complexity and expressivity of our method to related work.

Understanding and relating the expressivity of different approaches is non-trivial. In the case of PathNet, its expressivity concerning the WL-hierarchy remains unexplored. PathNN-SP+ has been shown to be more expressive than 1-WL. While it is claimed that RFGNNs are maximally expressive, the proof claiming higher expressivity on node level as presented in Chen et al. [8, Lemma 7] is not correct (rather it is incomparable on node level, as detailed in Appendix A). Consequently, it remains uncertain, whether RFGNN is strictly more expressive on graph level. PathNN-SP [26] states that it can only disambiguate graphs at least as well as 1-WL and is not strictly more powerful. This is because, due to sampling, isomorphic graphs could be mapped to different representations, indicating that it is not a graph invariant.

RFGNN and PathNN-SP+ involve enumerating all possible paths and all shortest paths, respectively. A straightforward example of a graph consisting of a chain of joined 4-cycles, shows that there is an exponential number of such paths, and that these paths still contain redundancy. In our approach, we avoid this redundancy by not explicitly building NTs, but instead generate DAGs, resulting in a much more compact representation (refer to Appendix D).

In Appendix B, we investigate the theoretical connection of our method and RFGNN in terms of relative influence, and show that 0- and 1-NTs can address the oversquashing problem more effectively.

To summarize, our method is the first to address both types of redundancy in GNNs, informational and computational redundancy, with polynomial running time, and can mitigate oversquashing better than comparable approaches. Therefore, our method addresses a gap in GNN research that has not been previously considered.

5. Experimental Evaluation

We assess the performance of DAG-MLP with k -NTs on a range of synthetic [1, 28] and real-world datasets [9, 24, 11, 31, 27] (additional details can be found in Appendix H).³

³Our implementation is available (anonymized) at <https://anonymous.4open.science/r/k-RedundancyGNNs/> and will be uploaded to GitHub upon publication.

Table 2: Average classification accuracy for EXP-Class and CSL across k -folds (4-folds and 5-folds), and the number of indistinguishable pairs of graphs in EXP-Iso. Best results are highlighted in **gray**, best results from methods with polynomial time complexity are highlighted in **bold**.

Model	EXP-Class \uparrow	EXP-Iso \downarrow	CSL \uparrow
GIN [37]	50.0 \pm 0.0	600	10.0 \pm 0.0
3WLGNN [23]	100.0 \pm 0.0	0	97.8 \pm 10.9
PathNN- \mathcal{SP}^+ [26]	100.0 \pm 0.0	0	100.0 \pm 0.0
PathNN- \mathcal{AP} [26]	100.0 \pm 0.0	0	100.0 \pm 0.0
DAG-MLP (0-NTs)	100.0 \pm 0.0	0	100.0 \pm 0.0
DAG-MLP (1-NTs)	100.0 \pm 0.0	0	100.0 \pm 0.0

Experimental setup. For synthetic datasets, we determine the number of layers in DAG-MLP based on the average graph diameter, ensuring effective aggregation during message propagation. The embeddings at each layer are obtained using readouts, concatenated, and then fed through two learnable linear layers for prediction. In the evaluation of TUDataset, we adopt the 10-fold splits proposed by [10], allowing a grid search for optimal hyper-parameters on each dataset. The architecture for combined heights involves using each “readout $_i$ ” to extract the embeddings for each layer, with mean of the average-pooled embeddings being passed to a final MLP layer responsible for prediction (see Appendix F). For the fixed single-height architecture, only the last readout is used, pooled, and passed to the final MLP layer. Further details on hyper-parameters can be found in Appendix J.

Graph classification. Table 2 presents the results on synthetic expressivity datasets. In line with our theoretical expectations, the experimental results support our hypothesis: NTs exhibit greater expressivity than GIN on a graph level. However, a formal theoretical proof of this observation remains a direction for future work.

In Table 3, we investigate the impact of the parameter k and the number of layers l on the accuracy for the EXP-Class dataset. Cases where $k > l$ can be disregarded, as the computation for NTs remains the same as when $k = l$. Empirically, 0- and 1-NTs yield the highest accuracy. This observation aligns with our discussions on expressivity in Section 4.4. The decrease in accuracy with increasing k indicates that information redundancy leads to oversquashing. We investigate this theoretically in Appendix B.

For TUDataset, we report the accuracy compared to related work in Table 4. We report only the best results for the different parameter combinations reported in Michel et al. [26], and the best result for our different combine methods. Due to the high standard deviation across all methods, we present a statistical box plot for the accuracy based on three runs on the test set of 10-fold cross-validation in Appendix G. We group the methods by their time complexity. Note that, while PathNN performs well on ENZYMES and PROTEINS, the time complexity of this method is exponential. Therefore, we also highlight the best method with polynomial time complexity. For IMDB-B and IMDB-M, which have small diameters, we see that k -NTs

Table 3: Average accuracy for DAG-MLP using 4-fold cross-validation on EXP-Class [1], evaluated with varying number of layers.

k -NTs	1 layer	2 layers	3 layers	4 layers	5 layers	6 layers
0-NTs	51.1 \pm 1.6	57.5 \pm 6.6	91.7 \pm 11.6	99.7 \pm 0.3	100.0 \pm 0.0	100.0 \pm 0.0
1-NTs	50.1 \pm 0.2	58.9 \pm 4.6	59.4 \pm 5.7	99.6 \pm 0.5	99.9 \pm 0.2	100.0 \pm 0.0
2-NTs	-	52.6 \pm 3.4	54.9 \pm 5.3	52.4 \pm 3.8	97.6 \pm 1.9	100.0 \pm 0.0
3-NTs	-	-	56.2 \pm 5.7	51.1 \pm 1.9	52.4 \pm 4.1	87.1 \pm 21.4
4-NTs	-	-	-	50.1 \pm 0.2	50.6 \pm 1.0	50.4 \pm 0.7
5-NTs	-	-	-	-	50.4 \pm 0.7	50.0 \pm 0.0
6-NTs	-	-	-	-	-	53.2 \pm 5.2

Table 4: Classification accuracy (\pm standard deviation) over 10-fold cross-validation on the datasets from TUDataset, taken from Michel et al. [26]. Best performance is highlighted in **gray**, best results from methods with polynomial time complexity are highlighted in **bold**. “-” denotes not applicable and “NA” means not available.

	Model	IMDB-B	IMDB-M	ENZYMES	PROTEINS
Linear	GIN [37]	71.2 \pm 3.9	48.5 \pm 3.3	59.6 \pm 4.5	73.3 \pm 4.0
	GAT [35]	69.2 \pm 4.8	48.2 \pm 4.9	49.5 \pm 8.9	70.9 \pm 2.7
	SPN ($l = 1$) [2]	NA	NA	67.5 \pm 5.5	71.0 \pm 3.7
	SPN ($l = 5$) [2]	NA	NA	69.4 \pm 6.2	74.2 \pm 2.7
Exp	PathNet [33]	70.4 \pm 3.8	49.1 \pm 3.6	69.3 \pm 5.4	70.5 \pm 3.9
	PathNN- \mathcal{P} [26]	72.6 \pm 3.3	50.8 \pm 4.5	73.0 \pm 5.2	75.2 \pm 3.9
	PathNN- \mathcal{SP}^+ [26]	-	-	70.4 \pm 3.1	73.2 \pm 3.3
Ours	DAG-MLP (0-NTs)	72.9 \pm 5.0	50.2 \pm 3.2	67.9 \pm 5.3	70.1 \pm 1.7
	DAG-MLP (1-NTs)	72.4 \pm 3.8	51.3 \pm 4.4	70.6 \pm 5.5	70.2 \pm 3.4

outperform all other methods. For ENZYMES a variant of our approach achieves the best result among the approaches with non-exponential time complexity, and k -NTs lead to a significant improvement over GIN.

Node classification. We investigate the performance of our approach on node classification datasets. These datasets differ regarding their homophily ratio, i.e., the fraction of edges in a graph that connect vertices with the same class label [40]. Heterophily tasks are particularly challenging for standard GNNs [40] as they require capturing the structure of neighborhoods instead of “averaging” over the neighboring features. In Table 5 we present results from [14] including the state-of-the-art graph rewiring technique SJLR combined with SGC and GCN, which performs best in the evaluation. We also performed experiments with GIN and DAG-MLP using the same data splits as [14] to ensure a fair comparison. We report the best results for l layers with $l \in \{2, 3, 4\}$ and four different combine methods for GIN and DAG-MLP.

As observed, DAG-MLP outperforms GIN on the heterophily datasets (those with low homophily ratio), while GIN performs better on homophily ones. These results indicate that

Table 5: Accuracy and standard deviation on node classification tasks (GCN, SJLR-GCN, SGC and SJLR-GCN taken from [14]).

Model	Texas	Wisconsin	Cornell	Cora	CiteSeer	PubMed
Homophily ratio	0.11	0.21	0.3	0.8	0.74	0.8
GCN	58.05 ± 0.9	52.10 ± 0.9	67.34 ± 1.5	81.81 ± 0.2	68.35 ± 0.3	78.25 ± 0.3
SJLR-GCN	60.13 ± 0.8	55.16 ± 0.9	71.75 ± 1.5	81.95 ± 0.2	69.50 ± 0.3	78.60 ± 0.3
SGC	56.69 ± 1.7	47.90 ± 1.7	53.40 ± 2.1	76.90 ± 1.3	67.45 ± 0.8	71.79 ± 2.1
SJLR-SGC	58.40 ± 1.4	55.42 ± 0.9	67.37 ± 1.6	81.24 ± 0.7	68.39 ± 0.6	76.28 ± 0.9
GIN	73.78 ± 6.0	71.76 ± 5.1	60.81 ± 8.5	76.76 ± 1.4	64.49 ± 1.5	76.46 ± 1.1
DAG-MLP (0-NTs)	85.68 ± 4.8	81.35 ± 4.1	79.02 ± 6.8	74.01 ± 2.0	60.55 ± 3.6	75.33 ± 1.1
DAG-MLP (1-NTs)	80.54 ± 6.0	81.62 ± 3.4	79.41 ± 4.6	74.54 ± 1.4	61.09 ± 1.5	75.53 ± 1.1

neighborhood trees can capture the relevant neighborhood structure more accurately than unfolding trees used by GIN. Additionally, our method outperforms SJLR on the two heterophily datasets Texas and Wisconsin by a large margin.

6. Conclusion

We introduce a neural tree canonization technique and combine it with neighborhood trees, which are pruned versions of unfolding trees used by standard MPNNs. By merging trees in a DAG, we create compact representations that serve as the foundation for our neural architecture termed DAG-MLP. It inherits the advantageous properties of the GIN architecture, while being more expressive than 1-WL on many graphs. Notably, our method is only less expressive on node level for specific examples. Our work contributes general techniques for constructing compact computation DAGs for tree structures that encode node neighborhoods. This exploration reveals a complex interplay between information redundancy, computational redundancy, and expressivity. The delicate balance of these factors is an avenue for future work.

Acknowledgments

We would like to thank Christian Permann for his contribution to the conception of neighborhood trees and their efficient generation. This work was supported by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG19009]. The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

Author Contributions

NK devised the project and the main conceptual ideas. FB made significant contributions to the conception of k -redundant neighborhood trees and their efficient generation. NK and FB jointly developed the methods for redundancy removal presented in Sections 4.1, 4.2. FB developed Theorem 3, and implemented the k -redundant neighborhood trees, merge DAGs, and algorithmic components of the implementation. NK, SM, and FB collaborated on the development of the DAG-MLP architecture. SM implemented DAG-MLP, conducted the experimental evaluation, and wrote parts of the corresponding sections in the manuscript. SM extended support to directed graphs and graphs with edge attributes, implemented the learning pipeline, wrapped the DAG to be used within the learning pipeline, and configured the necessary environments to reproduce results. FB and NK jointly drafted the manuscript with input from all authors. FB revised the manuscript with feedback from reviewers. NK, WG, and JL supervised the project. All authors provided critical feedback, participated in discussions, contributed to the interpretation of the results, and approved the final manuscript.

References

- [1] R. Abboud, I. I. Ceylan, M. Grohe, and T. Lukasiewicz. The Surprising Power of Graph Neural Networks with Random Node Initialization. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, pages 2112–2118, 2021.
- [2] R. Abboud, R. Dimitrov, and İ. İ. Ceylan. Shortest path networks for graph property prediction. In *LoG 2022*, volume 198 of *PMLR*, 2022. URL <https://proceedings.mlr.press/v198/abboud22a.html>.
- [3] S. Abu-El-Haija, B. Perozzi, A. Kapoor, N. Alipourfard, K. Lerman, H. Harutyunyan, G. V. Steeg, and A. Galstyan. MixHop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *ICML 2019*, volume 97 of *PMLR*, 2019. URL <http://proceedings.mlr.press/v97/abu-el-haija19a.html>.
- [4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN 0-201-00029-6.
- [5] U. Alon and E. Yahav. On the bottleneck of graph neural networks and its practical implications. In *ICLR 2021*. OpenReview.net, 2021. URL <https://openreview.net/forum?id=i800Ph0CVH2>.
- [6] F. Bause, C. Permann, and N. Kriege. Approximating the graph edit distance with compact neighborhood representations. 2023.
- [7] M. Black, Z. Wan, A. Nayyeri, and Y. Wang. Understanding oversquashing in GNNs through the lens of effective resistance. In *International Conference on Machine Learning*, volume 202 of *PMLR*, pages 2528–2547. PMLR, 23–29 Jul 2023. URL <https://proceedings.mlr.press/v202/black23a.html>.

- [8] R. Chen, S. Zhang, L. H. U, and Y. Li. Redundancy-free message passing for graph neural networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 4316–4327. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/1bd6f17639876b4856026744932ec76f-Paper-Conference.pdf.
- [9] M. W. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. M. Mitchell, K. Nigam, and S. Slatery. Learning to extract symbolic knowledge from the world wide web. In *AAAI/IAAI*, 1998.
- [10] F. Errica, M. Podda, D. Bacciu, and A. Micheli. A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=HygDF6NFPB>.
- [11] C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: an automatic citation indexing system. In *Digital library*, 1998.
- [12] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.
- [13] F. D. Giovanni, L. Giusti, F. Barbero, G. Luise, P. Lio, and M. M. Bronstein. On over-squashing in message passing neural networks: The impact of width, depth, and topology. volume 202 of *PMLR*. PMLR, 2023.
- [14] J. H. Giraldo, K. Skianis, T. Bouwmans, and F. D. Malliaros. On the trade-off between over-smoothing and over-squashing in deep graph neural networks. In *CIKM*, 2023.
- [15] S. Jegelka. Theory of graph neural networks: Representation and learning. *CoRR*, abs/2204.07697, 2022.
- [16] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken. Redundancy-free computation for graph neural networks. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 997–1005. ACM, 2020. doi: 10.1145/3394486.3403142. URL <https://doi.org/10.1145/3394486.3403142>.
- [17] N. M. Kriege. Weisfeiler and leman go walking: Random walk kernels revisited. In *NeurIPS*, 2022.
- [18] N. M. Kriege, P.-L. Giscard, and R. C. Wilson. On valid optimal assignment kernels and applications to graph classification. In *International Conference on Neural Information Processing Systems*, NIPS, 2016.
- [19] P. Li, Y. Wang, H. Wang, and J. Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. In *NeurIPS*, 2020. URL <https://proceedings.neurips.cc/paper/2020/hash/2f73168bf3656f697507752ec592c437-Abstract.html>.

- [20] Q. Li, Z. Han, and X. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 2018. URL <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16098>.
- [21] M. Liu, H. Gao, and S. Ji. Towards deeper graph neural networks. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2020. doi: 10.1145/3394486.3403076.
- [22] P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert. Extensions of marginalized graph kernels. In *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004. ISBN 1-58113-838-5. URL <http://doi.acm.org/10.1145/1015330.1015446>.
- [23] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably Powerful Graph Networks. In *Advances in Neural Information Processing Systems*, 2019.
- [24] A. McCallum, K. Nigam, J. D. M. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3:127–163, 2000.
- [25] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. ISBN 978-3-540-77977-3. URL <https://doi.org/10.1007/978-3-540-77978-0>.
- [26] G. Michel, G. Nikolentzos, J. Lutzeyer, and M. Vazirgiannis. Path neural networks: Expressive and accurate graph neural networks. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.
- [27] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, 2020. URL <http://www.graphlearning.io>.
- [28] R. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. Relational Pooling for Graph Representations. In *Proceedings of the 36th International Conference on Machine Learning*, pages 4663–4673, 2019.
- [29] G. Nikolentzos, M. Chatzianastasis, and M. Vazirgiannis. Weisfeiler and leman go hyperbolic: Learning distance preserving node representations. In *International Conference on Artificial Intelligence and Statistics*, volume 206 of *PMLR*, 2023. URL <https://proceedings.mlr.press/v206/nikolentzos23a.html>.
- [30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20(1):81–102, 2009. doi: 10.1109/TNN.2008.2005141.
- [31] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. In *The AI Magazine*, 2008.
- [32] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

- [33] Y. Sun, H. Deng, Y. Yang, C. Wang, J. Xu, R. Huang, L. Cao, Y. Wang, and L. Chen. Beyond Homophily: Structure-aware Path Aggregation Graph Neural Network. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence*, pages 2233–2240, 2022.
- [34] J. Topping, F. D. Giovanni, B. P. Chamberlain, X. Dong, and M. M. Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. In *International Conference on Learning Representations, ICLR*. OpenReview.net, 2022. URL <https://openreview.net/forum?id=7UmjRGzp-A>.
- [35] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations*, 2018.
- [36] K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. volume 80 of *PMLR*. PMLR, 2018.
- [37] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR*, 2019.
- [38] Y. Yang, X. Wang, M. Song, J. Yuan, and D. Tao. SPAGAN: shortest path graph attention network. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. IJCAI, 2019. URL <https://doi.org/10.24963/ijcai.2019/569>.
- [39] J. You, J. M. G. Selman, R. Ying, and J. Leskovec. Identity-aware graph neural networks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, 2021. URL <https://ojs.aaai.org/index.php/AAAI/article/view/17283>.
- [40] J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, and D. Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. In *NeurIPS*, 2020.

A. Comparison to RFGNN and TPTs

In RFGNN [8], truncated epath trees (TPTs) are introduced to represent the information flow with the goal of reducing redundancy. While the motivation aligns with our approach, there are significant differences between TPTs and k -redundant neighborhood trees and the computational properties of the techniques. In RFGNN the focus is solely on reducing redundancy in information flow, not computation. Additionally, the definition of TPTs allows for much more redundancy than that of k -NTs. We first introduce the concepts used by Chen et al. [8], and then discuss differences and disadvantages in detail.

An *epath* is defined as a path with no repeated vertices, except the starting vertex, which is allowed to be the ending vertex if the length of the epath is larger than 2.

Definition 4 (Truncated ePath Tree [8]). Given graph G and $v \in V(G)$, the TPT $TPT_{G,v}^h$ with height h is an epath search tree obtained by running a BFS from v , where all epaths of length up to h are accessed.

Firstly, the definition of TPTs allows for vertices to redundantly appear multiple times in a tree. If a vertex appeared at depth 1, for example, it can still appear elsewhere in the TPT, but not as its own descendant. In TPTs, parts of paths that differ will be repeated, without the ability to compress them. In contrast, neighborhood trees allow for compressed representations, as demonstrated in Appendix D. Chen et al. [8], Lemma 7 claims that TPTs are more expressive than unfolding trees of the same height by providing two example graphs, which can also be distinguished by 1-NTs of the same height, cf. Theorem 3. Chen et al. [8], Lemma 6 states, that for any two nodes, if the unfolding trees of height k differ, the TPTs of height k differ as well, however, this is not true. Figure 7 shows two nodes that can be distinguished by their unfolding trees (of height ≥ 4), but not by their TPTs. The problem is the same as for NTs - they have a maximum height, whereas unfolding trees will grow indefinitely. Hence, TPTs are not strictly more expressive than unfolding trees on node level, for the same reason that k -NTs are not. Unlike NTs, TPTs have several other disadvantages.

The size and running time complexity of RFGNN are very restrictive. While the term BFS in the definition implies linear running time, the BFS has to be modified, leading to exponential running time. The compression of TPTs (or even the forest of TPTs for the vertices of a graph) is not discussed in the publication, making preprocessing and computation much more time-consuming. In the experimental evaluation, only TPTs up to height 3 are used due to resource-intensity, indicating that the full expressivity of TPTs cannot be utilized in practice. This limitation is also reflected in the experimental results in [8], where the outcomes using RFGNN are only marginally better.

With our proposed approach, we address not only redundancy in information flow using k -NTs, but also remove redundancy in computation by incorporating merge DAGs. This strategy enables our approach to reach its full expressive potential in practice while maintaining a reasonable running time.

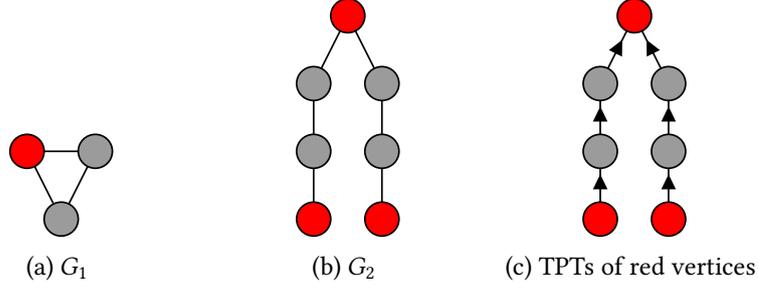


Figure 7: Two graphs (a), (b) in which the red vertices can be distinguished by unfolding trees, but not by TPTs. Figure (c) shows the TPTs of the red vertices (the red vertex in the middle of G_2), which are the same.

B. Theoretical Analysis of Oversquashing in Comparison with RFGNN

Several authors [8, 36, 34, 13, 7] developed and refined techniques to measure the influence of a vertex v with an initial vertex feature \mathbf{x}_v on the output $\mathbf{h}_u^{(k)}$ of a vertex u after layer k by the Jacobian $\partial \mathbf{h}_u^{(k)} / \partial \mathbf{x}_v$. Following Chen et al. [8, Lemma 3], the relative influence of a node v on a node u in an MPNN is

$$I(v, u) = \mathbb{E} \left(\frac{\partial \mathbf{h}_u^{(k)} / \partial \mathbf{x}_v}{\sum_{w \in V} \partial \mathbf{h}_u^{(k)} / \partial \mathbf{x}_w} \right) = \frac{[\hat{A}^k]_{u,v}}{\sum_{w \in V} [\hat{A}^k]_{u,w}},$$

where $\hat{A} = A + I$ is the adjacency matrix of the graph G with added self-loops. Note that $[\hat{A}^k]_{u,v}$ is the number of walks of length k from u to v (and vice versa) in G with added self-loops. Oversquashing occurs when $I(v, u)$ becomes small, indicating that only a small fraction of walks of length up to k ending at u start at v .

This idea can easily be linked to the concept of unfolding trees underlying our work. Consider the unfolding tree F_k^u of vertex u with height k . It follows from its construction that there is a bijection between walks of length at most k ending at u in G and paths in F_k^u from some node to the root (see [17] for details on unfolding trees and walks). Therefore, pruning the unfolding tree has an effect on walk counts and, thus, on the relative influence. Consider the example in Figure 1 and let u be the red vertex (upper left) and v the yellow vertex (lower right). We obtain a relative influence of $I_{\text{MPNN}}(v, u) = \frac{1}{8}$ for unfolding trees, and $I_{0\text{NT}}(v, u) = \frac{1}{4}$ for 0-NTs, showing that NTs have the potential to reduce oversquashing.

We formally show that our method is less susceptible to oversquashing than MPNNs and RFGNN [8]. Consider a vertex v and a vertex u with shortest-path distance of k . To pass information from v to u , at least k layers are required. Comparing the unfolding tree (MPNN), the 0- and 1-NT (our approach) and the TPT (RFGNN), all of height k , reveals that the vertex v occurs in the last level only, i.e., as a leaf of the tree, and the number of occurrences of v is equal in all trees, since all walks and simple paths of length k reaching v are shortest paths.

Hence, the numerator of the relative influence is equal for all methods. However, since 0- and 1-NTs are subtrees of unfolding trees, and 0-NTs/1-NTs are subtrees of TPTs (they contain only shortest paths/some simple paths, instead of all simple paths), the total number of nodes in the trees, i.e., walks contributing to the denominator of the relevant information can be compared, obtaining

$$I_{\text{MPNN}}(v, u) \leq I_{\text{TPT}}(v, u) \leq I_{\text{1NT}}(v, u) \leq I_{\text{0NT}}(v, u).$$

This theoretical analysis shows that our proposed method offers advantages in mitigating oversquashing, leveraging the formalization developed in recent papers. Additionally, it establishes a theoretical connection between the proposed approach and RFGNN, highlighting that our method more effectively addresses the oversquashing problem.

C. The AHU Algorithm

Aho, Hopcroft and Ullman describe a linear-time algorithm for deciding whether two rooted unordered trees are isomorphic [4, Section 3.2]. The algorithm forms the basis for our neural tree canonization technique, as discussed in Section 4.1, and serves as a fundamental subroutine for combining trees into a single merge DAG, as detailed in Appendix E. Here, we give a complete description of the original algorithm, its extension to trees with node labels or features, and the required modification for tree canonization.

In its original version, the algorithm solves the subtree isomorphism problem for two rooted unordered unlabeled trees T_1 and T_2 . Algorithm 1 shows the pseudocode of the algorithm.⁴ First, the nodes in the disjoint union of the input trees $T_1 \cup T_2$ are partitioned into levels according to their depth distinguishing leaves and internal nodes, see Figure 8. Note that the levels are numbered in reverse order of depth, i.e., for a node v on level i the equality $\text{depth}(v) = \text{hgt}(T) - i$ holds. The lists \mathcal{L}_i^* and \mathcal{L}_i contain all leaves and internal nodes, respectively, on level i . The labels c_{ahu} of the leaves are set to 0 and the tree is processed in bottom-up-fashion. In iteration i of the for-loop, the labels of all nodes \mathcal{L}_k for all $k < i$ have been computed and \mathcal{L}_k is sorted according to them. Note that \mathcal{L}_i^* contains only nodes v with $c_{\text{ahu}}(v) = 0$ for all $0 \leq i \leq \text{hgt}(T)$. Tuples $\hat{c}_{\text{ahu}}(v)$ are generated for the nodes v in \mathcal{L}_i by iterating over \mathcal{L}_{i-1}^* and then \mathcal{L}_{i-1} appending the label of the current node to the tuple of its parent. Each tuple contains an integer label for each child and is in ascending order. In the next step, the nodes in \mathcal{L}_i are sorted according to the tuples using radix sort. Then, the RELABEL function assigns new integers $c_{\text{ahu}}(v)$ to all nodes v in \mathcal{L}_i based on their tuples. Since \mathcal{L}_i is sorted, all nodes with the same label form a contiguous sub-list. New integers are computed by scanning the list assigning 1 to the first entry and increasing the integer whenever the current tuple differs from the previous one. Using this approach, the RELABEL function computes an injection between tuples and integers appearing for the nodes in \mathcal{L}_i . Two trees are isomorphic if and only if their nodes yield the

⁴For clarity of presentation, we adapted and simplified the textual description of the textbook [4]. In contrast to the original description, our algorithm operates on the disjoint union of both trees instead of applying the same operations to T_1 and T_2 individually.

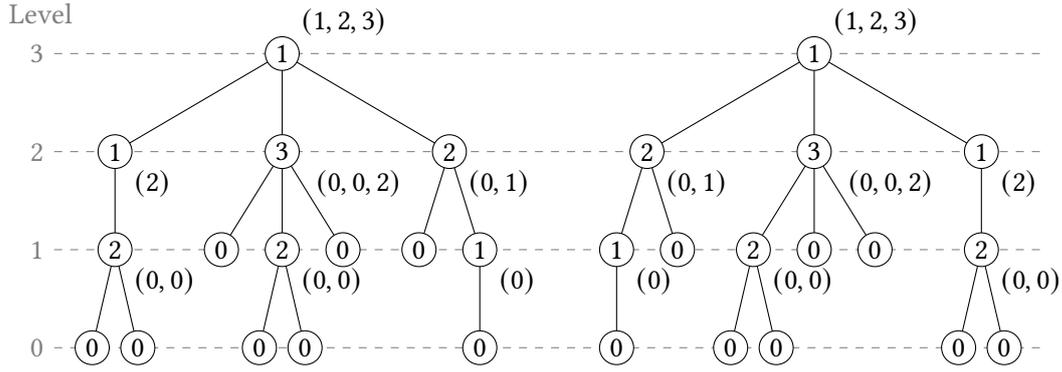


Figure 8: Two isomorphic trees T_1 (left) and T_2 (right) and the labels c_{ahu} (inside each node) and \hat{c}_{ahu} (right of each node) computed by the AHU algorithm.

same multiset of labels on all levels. Figure 8 shows an example of two trees that are identified as isomorphic. The algorithm can be implemented in linear time by applying RADIXSORT to sort tuples of labels from a bounded range.

As noted by Aho et al. [4], the algorithm can be extended to trees with initial integer labels in range 1 to n with $n = O(|V(T)|)$, by including the label of a node as the first element in its tuple. In this case, the RELABEL function assigns integers that were not used as initial labels and the leaves in \mathcal{L}_i^* have to be sorted according to their label after initialization. The overall running time remains linear.⁵ If the labels are not integers from a bounded range, e.g., continuous values, an initial mapping to integers is required, which can be realized by comparison-based sorting in $O(n \log n)$.

In order to generalize the method to tree canonization, it is no longer sufficient that the re-labeling function is injective for the tuples appearing on each level, but it has to be injective for all possible tuples that can occur in *any* tree. We discuss this situation in Section 4.1 and propose a learnable function with this property.

D. Building Compact Trees

Since unfolding trees can grow exponentially in size and our goal is to avoid redundant computation, we do not build unfolding trees and k -NTs explicitly. Rather, we build DAGs that represent them, corresponding to the merge DAG of only that tree using $L = \phi$. This way, the k -NTs can be generated by a simple, slightly modified BFS algorithm, and the size of k -NTs is in $O(|E(G)| \cdot (k + 1))$, which means it is linear in the size of the input graph G .

⁵A similar technique including level-wise processing, creation of tuples sorted by radix sort and relabeling has been proposed by Shervashidze et al. [32, Section 2.1] in the context of the Weisfeiler-Leman kernel to achieve a linear running time.

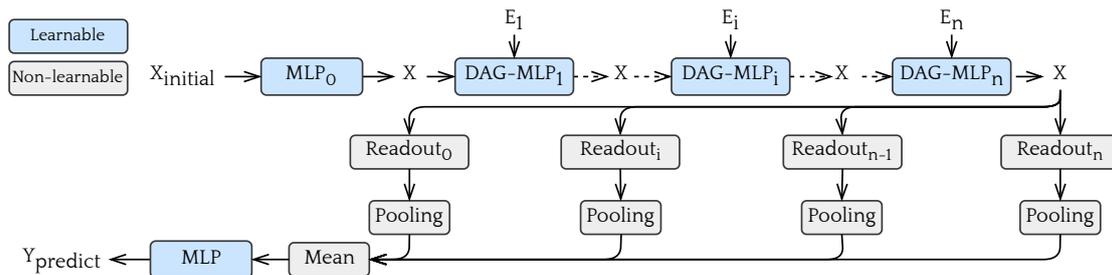


Figure 9: DAG-MLP architecture with n layers for graph-level prediction tasks.

E. Merging Trees – Algorithm

Algorithm 2 describes how to merge a set of trees $\{T_1, \dots, T_n\}$ into a DAG under a labeling function $L: \bigcup_{i \in \{1, \dots, n\}} V(T_i) \rightarrow \mathcal{O}$, where \mathcal{O} is some arbitrary labeling. All substructures that are isomorphic under L are merged. For that, the canonization of all vertices is computed first. Then each tree is merged to the DAG separately: Starting at the root $r(T)$ of the tree that is added, if a node with the same canonization as $r(T)$ exists in the DAG, nothing needs to be done. Otherwise, the subtrees rooted at the children of $r(T)$ are added first (using the same procedure as for $r(T)$), and then a new node for $r(T)$ is added along with edges to the nodes in the DAG that have the same canonization as the children of $r(T)$. Note that, if some children have the same canonization, in this step multiedges can occur. The algorithm can easily be extended to merge DAGs by iterating over all roots in MERGE and adding them to the DAG. The running time of the algorithm depends on the canonization, which can be done in time linear in the numbers of nodes (see Appendix C), and the time needed to add the trees to the DAG. Since we add each node at most once, and can check whether a canonization is already present in the DAG in constant time, this also only needs time linear in the number of tree nodes.

F. DAG-MLP Architecture for Graph Classification

Figure 9 shows an example of the architecture when using unfolding or neighborhood trees of height up to n for graph classification requiring n DAG-MLP layers. The vertex features are initially transformed into embedding with fixed dimension using MLP_0 . Messages are then propagated using the DAG from height 0 to 1 (\mathcal{E}_1), which corresponds to layer 1. This process is repeated for n layers, where the i th step computes embeddings for nodes of height i in the DAG. After n layers, all node embeddings in the DAG (X) have been updated. Using readouts, we extract the embeddings of each vertex from k -NTs of different heights within the DAG. These extracted embeddings correspond to the embeddings of different layers. A pooling operation is then applied to the output of each layer, and the pooled outputs are averaged. These averaged outputs are passed through a final MLP, transforming them into probabilities for class prediction.

Table 6: Average accuracy of DAG-MLP using 5-fold cross-validation on CSL [28], evaluated with varying parameters k and l .

k -NTs	1 layer	2 layers	3 layers	4 layers	5 layers	6 layers
0-NTs	10.0 \pm 0.0	20.0 \pm 0.0	40.0 \pm 0.0	70.0 \pm 0.0	84.0 \pm 4.9	100.0 \pm 0.0
1-NTs	10.0 \pm 0.0	10.0 \pm 0.0	30.0 \pm 0.0	48.0 \pm 4.0	78.0 \pm 9.8	100.0 \pm 0.0
2-NTs	-	10.0 \pm 0.0	16.0 \pm 4.9	20.0 \pm 12.6	50.0 \pm 8.9	80.0 \pm 12.6
3-NTs	-	-	10.0 \pm 0.0	10.0 \pm 0.0	20.0 \pm 12.6	38.0 \pm 14.7
4-NTs	-	-	-	10.0 \pm 0.0	16.0 \pm 8.0	34.0 \pm 12.0
5-NTs	-	-	-	-	10.0 \pm 0.0	10.0 \pm 0.0
6-NTs	-	-	-	-	-	10.0 \pm 0.0

Table 7: Classification accuracy for 10-folds (\pm standard deviation) on MUTAG comparing DAG-MLP that combines layers of different heights to DAG-MLP that only uses layers at a fixed height.

k -NTs	Combine Heights			Fixed Single-Height		
	1 layer	2 layers	3 layers	1 layer	2 layers	3 layers
0-NTs	84.6 \pm 6.2	86.7 \pm 5.3	86.9 \pm 6.0	85.3 \pm 6.3	89.0 \pm 4.7	87.2 \pm 5.1
1-NTs	84.9 \pm 6.0	83.3 \pm 7.3	88.6 \pm 6.7	85.8 \pm 6.0	88.8 \pm 4.4	90.4 \pm 5.1

G. Additional Experiments

Following the same experimental setup as in Table 3, Table 6 shows the accuracy with varying parameters k and l . Since the expressive capabilities are the same as those of GIN when the number of layers l equals the redundancy parameter k , all results with $l = k$ are not better than guessing.

Table 7 shows a comparison of 0- or 1-NTs, with combined heights and fixed single-height, for 10-fold cross-validation on MUTAG. The results as well as those shown in Table 8 indicate that using multiple different tree heights does not improve the generalization capabilities of the model.

Figure 10 shows box plot charts for the accuracy obtained in Table 4. Due to the use of 10-fold cross-validation and the random initialization of the MLPs, the results tend to have high variance. For all datasets, the accuracy of DAG-MLP is statistically within the same boundaries as those of the best related methods reported in Table 4.

H. Datasets

We provide information about the datasets used in the experimental evaluation. Table 9 provides an overview of the datasets, along with their corresponding characteristics.

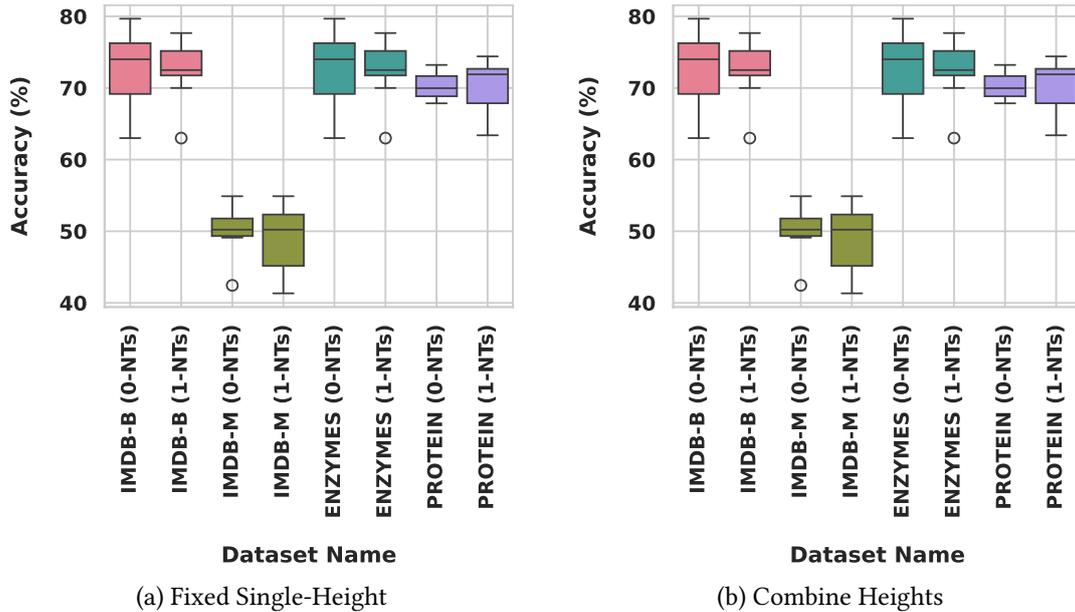


Figure 10: Graph classification test accuracy box plot over three runs of 10-fold cross-validation of the DAG-MLP on the datasets from the TUDataset.

Synthetic datasets. (1) EXP-Classification (EXP-Class) and EXP-Isomorphic (EXP-Iso) evaluate GNN expressivity, featuring graph pairs with varying SAT outcomes and 1-WL distinguishability [1]. EXP-Class extends EXP-Iso by including 50% “corrupted” data, making the learning task more challenging. (2) Circulant Skip Links (CSL) graphs [28] are highly symmetric, 4-regular graphs that consist of a cycle with additional ‘skip links.’ Despite their symmetry, these graphs present a challenge for the WL test and GNNs based on WL, as these methods fail to distinguish between non-isomorphic instances of such graphs.

Real-world datasets. We examine Texas, Wisconsin, Cornell, Cora, CiteSeer, PubMed, MUTAG, IMDB-B, IMDB-M, ENZYMES, and PROTEINS from [9, 24, 11, 31, 27]. Texas, Wisconsin, and Cornell are web page datasets, each representing a different university’s web domain. Cora dataset comprises scientific publications classified into seven categories, making it a standard benchmark for citation network studies. CiteSeer is another citation network dataset, including academic papers for document classification. The PubMed dataset, derived from biomedical literature, leveraging its rich metadata encompassing abstracts, citations, and other bibliometric information. IMDB-B and IMDB-M are movie network datasets for binary and multi-class classification, respectively. ENZYMES has six protein graph classes, while PROTEINS represents a binary classification task from bioinformatics.

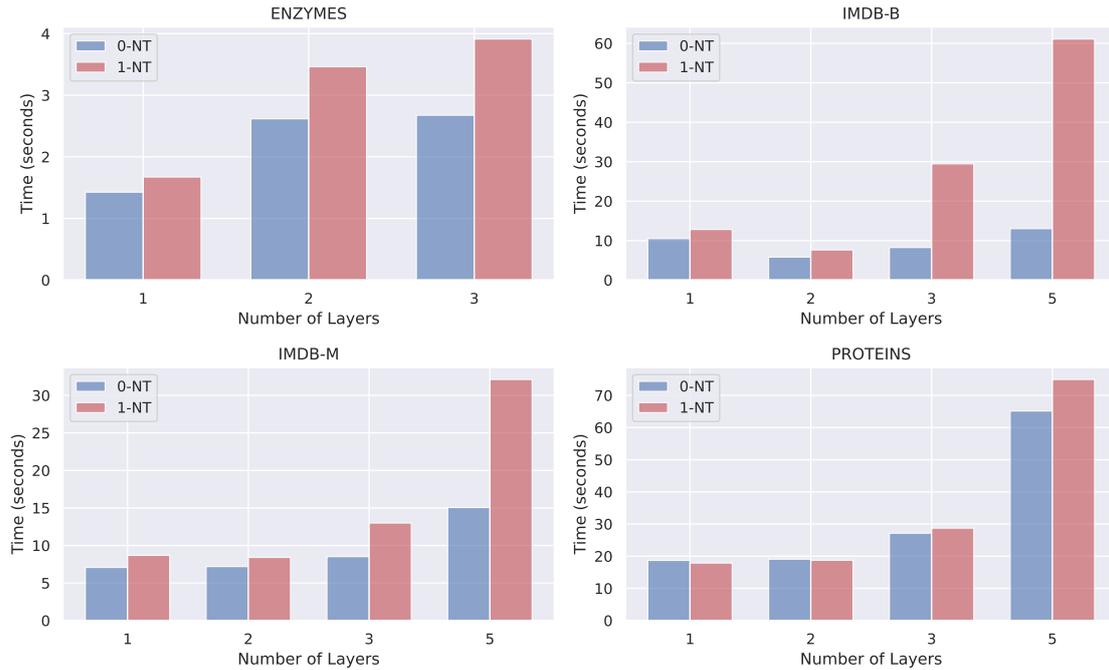


Figure 11: Running time for building the 0- and 1-redundant NTs.

I. Running Time

The running time for generating and merging 0- and 1-NTs with different layers on different datasets is presented in Figure 11. We employ a parallelized algorithm to construct the NTs, where each graph is also processed in parallel.

J. Hyper-Parameters

The hyper-parameters used for the synthetic datasets can be seen in Table 10. The hyper-parameters for the TUDataset experiments were chosen as follows: The batch size for training is set to 64. Learning rate (LR) is set to 0.001. The classifier trained for 500 epochs. The dimension of the embedding is set to 128. The optimizer used is Adam, and the scheduler is set to StepLR with a step size of 100 and a gamma value of 0.5. The aggregation method (pooling) is defined as mean and a dropout rate of 0.5 is specified. Early stopping is configured with a patience of 250 epochs and uses accuracy instead of loss. The number of layers for each dataset is set as in Table 11, and shuffling of the dataset is enabled.

K. Hardware

The hardware configuration consists of dual AMD 7252 CPUs, each with 8 cores, and two NVIDIA A40 GPUs. The system is supplemented with 256 GB of RAM. Each NVIDIA A40 GPU comes with 10,752 CUDA cores and a clock frequency of 1.305 GHz. The GPUs have 48 GB of memory and a bandwidth of 696 GB/s, operating at a Thermal Design Power (TDP) of 300 Watts. In terms of performance, the GPUs can deliver 37,400 GFLOPs in single-precision (FP32) and 1,169 GFLOPs in double-precision (FP64) computations.

Algorithm 1 AHU algorithm for tree isomorphism

function RELABEL(nodes \mathcal{L} , labels $\hat{c}_{\text{ahu}}, c_{\text{ahu}}$) \triangleright Replaces tuple labels $\hat{c}_{\text{ahu}}(v)$ by integer labels $c_{\text{ahu}}(v)$ for the nodes v in the list $\mathcal{L} = (l_1, l_2, \dots, l_N)$ sorted according to \hat{c}_{ahu} .

prev $\leftarrow \hat{c}_{\text{ahu}}(l_1)$

$k \leftarrow 1$

for $i \leftarrow 1$ **to** N **do**

if $\hat{c}_{\text{ahu}}(l_i) = \text{prev}$ **then**

$c_{\text{ahu}}(l_i) \leftarrow k$

else

$k \leftarrow k + 1$

\triangleright Next integer label

$c_{\text{ahu}}(l_i) \leftarrow k$

 prev $\leftarrow \hat{c}_{\text{ahu}}(l_i)$

return c_{ahu}

function TREEISMORPHISM(T_1, T_2)

$T \leftarrow T_1 \cup T_2$

$H \leftarrow \text{hgt}(T)$

for $i \leftarrow 0$ **to** H **do**

$\mathcal{L}_i^* \leftarrow \{v \in V(T) \mid \text{depth}(v) = H - i \wedge \text{chi}(v) = \emptyset\}$ \triangleright Leaves with the same depth

$\mathcal{L}_i \leftarrow \{v \in V(T) \mid \text{depth}(v) = H - i \wedge \text{chi}(v) \neq \emptyset\}$ \triangleright Non-leaves with the same depth

for each leaf $v \in V(T)$ **do**

$c_{\text{ahu}}(v) \leftarrow 0$

\triangleright Initialize labels for leaves

for $i \leftarrow 1$ **to** H **do**

for each l in ordered list $\mathcal{L}_{i-1}^* + \mathcal{L}_{i-1}$ **do** \triangleright Iterate over concatenated ordered list

 Append $c_{\text{ahu}}(l)$ to the tuple $\hat{c}_{\text{ahu}}(p(l))$ \triangleright Pass label of previous level upwards

$\mathcal{L}_i \leftarrow \text{RADIXSORT}(\mathcal{L}_i, \hat{c}_{\text{ahu}})$

\triangleright Sort list according to their tuples

$c_{\text{ahu}} \leftarrow \text{RELABEL}(\mathcal{L}_i, c_{\text{ahu}}, \hat{c}_{\text{ahu}})$

if $\{\{c_{\text{ahu}}(v) \mid (\mathcal{L}_i^* \cup \mathcal{L}_i) \cap V(T_1)\}\} \neq \{\{c_{\text{ahu}}(v) \mid (\mathcal{L}_i^* \cup \mathcal{L}_i) \cap V(T_2)\}\}$ **then**

return false

return true

Algorithm 2 Merging trees

function MERGE(set of trees \mathcal{T} , labeling L) ▷ merges \mathcal{T} into a DAG D
 $D \leftarrow$ empty DAG ▷ start with empty DAG
 initialize $D.can_map$ as an empty map ▷ maps canonization to node in DAG
 for each $T \in \mathcal{T}$ **do**
 compute canonization $can(v)$ for $v \in V(T)$ under L
 ADD($D, T, r(T), L$) ▷ add tree, starting at root
 return D

function ADD(DAG D , tree T , vertex v , labeling L) ▷ adds substructure rooted at $v \in V(T)$ to D
 if $can(v) \in D.can_map$ **then** ▷ node (and substructure) already present in D
 return
 for each $c \in \text{chi}(v)$ **do** ▷ add all children first (if necessary)
 ADD(D, T, c, L)
 add new node v_2 with $L(v_2) = L(v)$ to D ▷ add new node
 set $can(v_2) = can(v)$ and $D.can_map(can(v_2)) = v_2$
 for each $c \in \text{chi}(v)$ **do**
 if edge exists from $D.can_map(can(c))$ to v_2 **then**
 increase multiplicity of edge by 1 ▷ a sibling had the same canonization
 else
 add edge from $D.can_map(can(c))$ to v_2 ▷ add edges from children to new node

Table 8: Performance comparison of GIN and DAG-MLP architectures with different combination strategies between layers across datasets. The best result for each method on each dataset is marked in **bold**.

Model	Texas	Wisconsin	Cornell
GIN (l=2) - Without Combine	61.89 ± 7.0	57.65 ± 5.6	46.22 ± 6.0
GIN (l=2) - Sum Combine	69.46 ± 7.3	68.04 ± 6.0	58.11 ± 5.8
GIN (l=2) - Mean Combine	70.81 ± 7.1	68.82 ± 5.8	54.86 ± 8.2
GIN (l=2) - Concat Combine	73.78 ± 6.0	69.22 ± 6.7	60.81 ± 8.5
GIN (l=3) - Without Combine	67.03 ± 7.2	58.82 ± 6.5	43.51 ± 7.3
GIN (l=3) - Sum Combine	67.03 ± 5.0	66.08 ± 5.8	50.81 ± 7.8
GIN (l=3) - Mean Combine	64.86 ± 6.6	63.92 ± 6.5	52.97 ± 9.5
GIN (l=3) - Concat Combine	72.70 ± 4.6	71.76 ± 5.1	59.73 ± 11.3
GIN (l=4) - Without Combine	67.57 ± 5.5	59.41 ± 3.8	42.70 ± 5.0
GIN (l=4) - Sum Combine	66.49 ± 8.4	63.14 ± 6.2	52.16 ± 10.1
GIN (l=4) - Mean Combine	70.54 ± 4.6	63.73 ± 8.9	49.73 ± 9.8
GIN (l=4) - Concat Combine	69.73 ± 6.1	68.63 ± 7.9	57.30 ± 7.7
GIN (l=5) - Without Combine	62.70 ± 6.8	52.16 ± 8.0	44.86 ± 7.0
GIN (l=5) - Sum Combine	67.57 ± 6.6	60.59 ± 6.4	47.03 ± 8.4
GIN (l=5) - Mean Combine	70.54 ± 6.7	57.84 ± 7.7	48.11 ± 10.9
GIN (l=5) - Concat Combine	73.24 ± 4.6	66.27 ± 3.7	51.89 ± 7.7
DAGMLP (l=2; 0-NTs) - Without Combine	74.59 ± 4.7	65.10 ± 6.6	60.81 ± 4.6
DAGMLP (l=2; 0-NTs) - Sum Combine	85.68 ± 4.8	77.84 ± 5.3	68.38 ± 4.5
DAGMLP (l=2; 0-NTs) - Mean Combine	77.03 ± 6.3	76.67 ± 3.7	65.95 ± 5.7
DAGMLP (l=2; 0-NTs) - Concat Combine	81.35 ± 7.1	79.41 ± 4.6	68.92 ± 5.4
DAGMLP (l=3; 0-NTs) - Without Combine	74.86 ± 8.8	66.08 ± 4.3	57.84 ± 2.5
DAGMLP (l=3; 0-NTs) - Sum Combine	78.11 ± 6.6	76.67 ± 4.5	64.59 ± 5.9
DAGMLP (l=3; 0-NTs) - MEAN Combine	77.57 ± 5.1	74.31 ± 5.1	63.24 ± 4.0
DAGMLP (l=3; 0-NTs) - Concat Combine	80.27 ± 8.1	78.82 ± 5.2	64.05 ± 5.8
DAGMLP (l=2; 1-NTs) - Without Combine	65.68 ± 5.4	64.51 ± 7.4	55.95 ± 6.7
DAGMLP (l=2; 1-NTs) - Sum Combine	80.54 ± 6.0	79.22 ± 6.3	67.03 ± 6.7
DAGMLP (l=2; 1-NTs) - Mean Combine	78.38 ± 5.7	79.61 ± 3.9	66.49 ± 5.0
DAGMLP (l=2; 1-NTs) - Concat Combine	79.73 ± 3.7	79.61 ± 5.1	69.19 ± 4.6
DAGMLP (l=3; 1-NTs) - Without Combine	65.95 ± 6.3	60.98 ± 7.9	54.59 ± 6.3
DAGMLP (l=3; 1-NTs) - Sum Combine	78.38 ± 6.2	79.61 ± 5.2	64.59 ± 5.0
DAGMLP (l=3; 1-NTs) - Mean Combine	73.51 ± 5.4	75.29 ± 4.9	66.49 ± 4.9
DAGMLP (l=3; 1-NTs) - Concat Combine	80.27 ± 6.0	78.63 ± 4.8	67.03 ± 2.6

Table 9: Summary of characteristics for the synthetic datasets [28, 1] and TUDatasets [27]. The table provides information on the dataset name, number of graphs ($|\mathbf{G}|$), average number of nodes ($\overline{|V|}$), average number of edges ($\overline{|E|}$), and average diameter (\overline{D}) for each dataset.

Dataset	$ \mathbf{G} $	$\overline{ V }$	$\overline{ E }$	\overline{D}
CSL	150	41.0	164.0	6.0
EXP-Class	1200	55.8	139.6	12.6
EXP-Iso	1200	44.4	110.2	8.5
MUTAG	188	17.93	39.59	8.22
IMDB-B	1000	19.8	193.1	1.9
IMDB-M	1500	13.0	131.9	1.5
ENZYMES	600	32.6	124.3	10.9
PROTEINS	1113	39.1	145.6	11.6
Texas	1	183	325	8
Wisconsin	1	251	515	8
Cornell	1	183	298	8
Cora	1	2708	10556	19
CiteSeer	1	3327	9104	28
PubMed	1	19717	88648	18

Table 10: Synthetic dataset hyper-parameter configuration details.

Dataset	Task	Embedding	Target	Layers	Batch Size	Epochs	LR
EXP-Class	Classification	64	10	15	32	200	10^{-3}
EXP-Iso	Isomorphism Test	1	1	6	1	-	-
CSL	Classification	64	10	6	32	200	10^{-3}

Table 11: TUDatasets layer configuration details.

Dataset	IMDB-B	IMDB-M	ENZYMES	PROTEINS
Layers	5, 3, 2	5, 3, 2	3, 2	5, 3, 2