universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Comparison of Neighborhood-Encoding Trees for Graph Kernels

verfasst von | submitted by

Ariana Christine Fox

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 645
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Data Science
Betreut von | Supervisor:	Assoz. Prof. Dipl.-Inf. Dr. Nils Morten Kriege

Acknowledgments

Thank you very much to Dr. Nils Kriege and Franka Bause for the countless hours and meetings that went into completing this thesis.

Abstract

Graph kernels are an established method in the field of graph learning. They provide a continuous similarity measure between graphs that can be more useful in tasks like classification than more traditional graph learning methods. In this work, we attempt to improve the classification accuracy and runtime of three key graph kernels based on Weisfeiler-Leman color refinement: Weisfeiler-Leman Subtree, Weisfeiler-Leman Optimal Assignment, and Generalized Weisfeiler-Leman. We modify these kernels by replacing the Weisfeiler-Leman color refinement with the canonicalization of two novel neighborhood-encoding trees: the k-Redundant Neighborhood Tree and the Truncated ePath Tree. Through experimental evaluation on simple molecular graph datasets and network graph datasets, we found that our new kernels had similar classification accuracies and similar or significantly slower runtimes.

Kurzfassung

Graphkerne erfreuen sich im Bereich des Graphenlernens wachsender Beliebtheit. Sie liefern ein kontinuierliches Ähnlichkeitsmaß zwischen Graphen, das bei Aufgaben wie der Klassifizierung nützlicher sein kann als traditionellere Graphenlernmethoden. In dieser Arbeit versuchen wir, die Klassifizierungsgenauigkeit und Laufzeit von drei wichtigen Graphkernen basierend auf der Weisfeiler-Leman-Farbverfeinerung zu verbessern: Weisfeiler-Leman Subtree, Weisfeiler-Leman Optimal Assignment und Generalized Weisfeiler-Leman. Wir modifizieren diese Kerne, indem wir die Weisfeiler-Leman-Farbverfeinerung durch die Kanonisierung zweier neuartiger Nachbarschaftskodierungsbäume ersetzen: des k-Redundant Neighborhood Trees und des Truncated ePath Trees. Durch experimentelle Auswertung einfacher molekularer Graphendatensätze und Netzwerkgraphendatensätze stellten wir fest, dass unsere neuen Kerne ähnliche Klassifizierungsgenauigkeiten und ähnliche oder deutlich langsamere Laufzeiten aufwiesen.

Contents

List of Figures

1. Introduction

Across a wide range of real-world scenarios, graphs provide the best structure to preserve important relationships within data. This can appear in many forms, such as social networks with individuals serving as vertices and friendships being encoded by edges, or in microbiology where graphs can be used to represent molecular structures. However, graph-structured data is frequently impossible to visualize for large data, and it can be even more difficult to interpret. Graph learning is a process of gathering insight from graphs by leveraging machine learning techniques for tasks like classification, clustering, or regression.

A common use case of graph learning is identifying similar graphs. Oftentimes, a graph has known characteristics (e.g., a molecule structure that is known to be toxic) and researchers want to investigate whether these traits are likely for other graphs. One popular example of this is the MUTAG dataset from which researchers feed a model nitroaromatic compounds structured as graphs with a label describing the compound's mutagenicity on Salmonella typhimurium. The model is then used on unseen nitroaromatic compounds to predict the mutagenicity label (Debnath et al., 1992). This dataset is a model case in graph classification research, as it provides a clear example of how graph structure relates to real-world labels. Advancing graph learning methods for datasets like MUTAG has implications for every field with data that can be modeled by graphs.

An intuitive approach to graph comparison is to check if the graphs are isomorphic, meaning they are structurally identical. This process, called isomorphism testing, is a key component of graph learning, however, it is limited by its binary nature: graphs are either isomorphic or they are not. It is also computationally complex with no known polynomial-time algorithm, making it infeasible for large-scale applications. Instead, to allow for more flexible comparisons in graph learning with a polynomial-time similarity measure, we turn to graph kernels. Graph kernels are functions with graphs as input that provide a continuous similarity measure, comparing the graphs' structures, such as subtrees or neighborhoods. They project these similarities into a high-dimensional feature space which can then be used in machine learning tools like Support Vector Machines for tasks like classification.

There are many types of graphs that behave differently with no one-size-fits-all solution, so many types of graph kernels already exist. Still, as is the case with all kernels, there is still room for improvement in terms of accuracy, time and computational power. Three kernels which we would like to improve upon are the Weisfeiler-Leman[1] (WL) Subtree

[1]The spelling of the last name for the co-creator of Weisfeiler-Leman color refinement A. Leman varies across the graph research community. For consistency, throughout this work we use the spelling Leman as he stated to be his preference: https://www.iti.zcu.cz/wl2018/pdf/leman.pdf.

Kernel (Shervashidze et al., 2011)., Generalized WL Kernel (Schulz et al., 2022), and WL Optimal Assignment Kernel (Kriege et al., 2016). These kernels all use WL color refinement (Weisfeiler and Lehman, 1968) for isomorphism testing as a building block. However, this isomorphism test has been found to fail to distinguish non-isomorphic graphs in several cases, including strongly regular graphs. Therefore, we propose a set of new graph kernels based on the three kernels above, replacing the color refinement labels with two novel structures that can capture complex neighborhood relationships: Truncated e-Path Trees (TPT) (Chen et al., 2022) and k-Redundant Neighborhood Trees (kNT) (Bause et al., 2024).

The TPT and kNT are neighborhood-encoding rooted trees built from a vertex within a graph, similar to the WL Unfolding Tree that captures the relationships of WL color refinement. However, the main difference lies in that the WL Unfolding Tree has no truncation metrics and will reproduce the relationships infinitely, while the TPT and kNT each have termination metrics (TPT prevents paths from revisiting vertices except in special cases and kNT restricts vertices from reappearing after k levels), attempting to capture the most important relationships while reducing redundancy. To compare trees, we first use a canonicalization algorithm to get a set of string representations for trees using each vertex in the graphs as the root. We then compare the set of strings with other graphs, and if any string is different across a pair of graphs, then they are found to be not isomorphic. In previous experimentation, we generated TPTs and kNTs for a wide range of graph types to see if the canonicalizations of these trees could accurately distinguish non-isomorphic graphs. We found that a TPT encoding did not fail to distinguish graphs of any type tested on, while a kNT encoding with $k = 1$ only failed on strongly regular graphs (notably, a graph type the WL color refinement algorithm has been shown to fail on). Further, as both are subtrees of the WL Unfolding Tree (though in some cases may coincide with the full WL Unfolding Tree), they can require less storage while still preserving important structural information.

Based on our findings, we believe there is potential for the new graph kernels to outperform the original Weisfeiler-Leman kernels in terms of accuracy or required resources. The most promising cases are indicated in the table below by (*). The WL Subtree kernel is foundational to all WL kernels, but has been improved upon by both the Generalized WL kernel and the Optimal Assignment WL kernel, so we don't expect exceptional results from the modified Subtree kernels. However, we still wish to see if the canonicalization of the neighborhood-encoding trees with reduced redundancy will lead to improvement over WL color refinement in the simplest of cases, so we do this testing. The WL Optimal Assignment kernel uses color refinement labels to compute optimal vertex matchings between graphs, so we hypothesize that the TPT variant of this kernel may lead to improved accuracies, as the TPT is more expressive than the Unfolding Tree implicitly generated by color refinement. The Generalized WL kernel is the most state-of-the-art among the three kernels we are hoping to improve upon, so we expect the TPT version of this kernel could show higher accuracy than the original WL version, as the TPT delivers the best results of the three neighborhood-encoding tree types in isomorphism testing. We also would like to see if the runtime of the Generalized WL kernel can be improved

with the implementation of the canonicalized kNTs because this kernel requires implicit computation of the tree edit distance and kNTs are usually smaller (and often much smaller) than the WL Unfolding Trees of the original kernel.

Kernel Type	Tree Type		
	WL Unfolding Tree	kNT	TPT
Subtree (Dot Product)	WL Subtree Kernel [Shervashidze et al., 2011]		
Optimal Assignment	WL Optimal Assignment Kernel [Kriege et al., 2016]		*
Generalized	Generalized WL Kernel [Schulz et al., 2022]	*	*

Table 1.1.: Combinations of Kernel and Tree Types. Gray highlighted cells indicate that the combination of Tree Type and Kernel Type already exists with the name of the paper where the combination was first proposed. Yellow highlighted cells indicate we use the combination of Tree Type and Kernel Type in our experimentation. Asterisks (*) indicate that for the combination, we hypothesize improved results over the existing WL kernel in terms of accuracy or runtime.

1.1. Objectives and Scope

The measures of success in our experimentation are the classification accuracy results of the kernel and its runtime.

We measure the accuracy of each kernel by using it for classification of graph datasets with a Support Vector Machine (SVM). SVMs are well-suited to this task, as they use the similarity measures output by the kernel to create a hyperplane which separates the data into classes. As the WL color refinement, kNT, and TPT are all able to distinguish most types of graphs, we expect there to be many cases where the accuracy results are the same or very similar. As such, we also compare the runtime for each kernel.

In order to get direct comparisons to the current state-of-the-art results, we use datasets already utilized by the authors of the WL based kernels, and we run experiments on these existing kernels as well as our new kernels.

2. Preliminaries

Unless denoted otherwise, all definitions in this chapter are sourced from Deo, 2016.

2.1. Graphs

A graph is a data structure composed of vertices and edges. We refer to the set of vertices as V, the set of edges as E, and the graph is then denoted $G = (V, E)$.

Definition 2.1.1 (Graph)
A graph $G = (V, E)$ consists of a set of objects $V = \{v_1, v_2, ...\}$ called vertices, and another set $E = \{e_1, e_2, ...\}$, whose elements are called edges, such that each edge e_k is identified with an ordered pair (v_i, v_j) of vertices if the graph is directed or an unordered pair of vertices $\{v_i, v_j\}$ if the graph is undirected.

Vertices are the units of the graph, and they may also be referred to as nodes. They can be labeled or unlabeled. Edges form a connection between two vertices, and they can be one of two types: directed or undirected. A directed edge means that there is a directed connection from one vertex to the other, but not necessarily in the opposite direction. An undirected edge creates a multidirectional connection between the two vertices, and it can be described by either the unordered vertex pair $\{v_i, v_j\}$ or by combination of two ordered vertex pairs (v_i, v_j) and (v_j, v_i) in the edge set E.

A path is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it. No edge appears (is covered or traversed) more than once in a path; the form of movement allowing repeated visitation of nodes and edges is called a walk. It is possible for a path to begin and end at the same vertex. Such a path is called a closed path. A closed path in which no vertex except the initial and the final vertex appears more than once is called a cycle. If every pair of vertices in a graph has at least one path between them, the graph is called connected.

2.2. Trees

One key graph type is called a tree, which is a connected graph without any cycles. A rooted tree is a special type of tree in which the vertices have a hierarchical relationship, with the topmost vertex being called the root of the tree, and each vertex stemming from it is its child. The children may also have children of their own, and any vertex with a child is called a parent of that child vertex. A vertex with no children is called a leaf. A

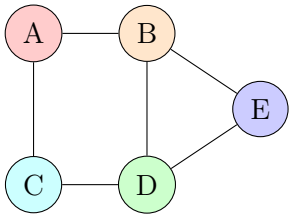


Figure 2.1.: Example Undirected Graph with 5 Vertices and 6 Undirected Edges.

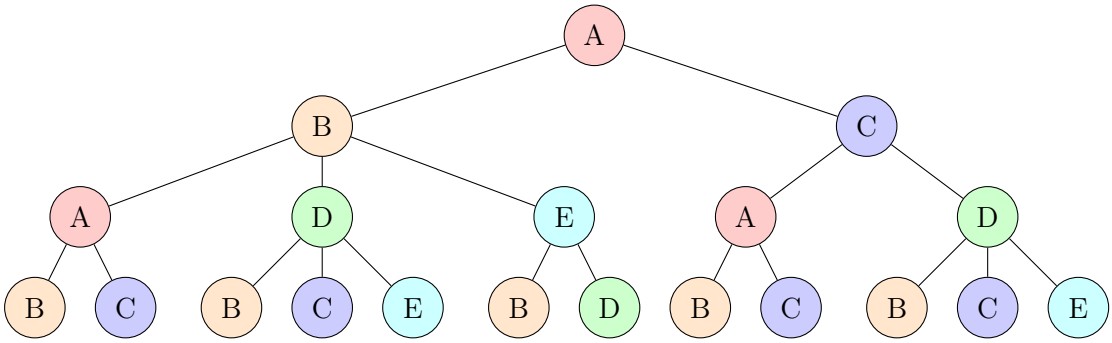


Figure 2.2.: Example Unfolding Tree with Maximum Height 4 for Vertex A in Figure 2.1.

subtree is a smaller tree contained entirely within another tree, and it is created from a vertex and all of that vertex's descendants in the original tree.

Definition 2.2.1 (Rooted Tree)

A tree is a connected graph without any cycles. A tree in which one vertex (called the root) is distinguished from all the others is called a rooted tree.

In a rooted tree, the depth of a vertex within the tree is the length of the shortest path from the vertex to the root, and the root itself has a depth of 0. The height of a tree is the maximum depth of any vertex within the tree.

Rooted trees can be constructed from a graph, encoding neighborhood relationships by organizing vertices in a hierarchical, cycle-free structure that captures the relationships between the vertices. Each vertex's neighbors in the original graph can be represented as children or descendants in the tree, simplifying the graph structure by removing cycles, which makes identifying and navigating relationships easier and more efficient.

An important instance of the rooted tree is the unfolding tree. An unfolding tree creates children for each vertex based on the vertex's neighbors in the original graph. If the original graph contains cycles, the unfolding tree can grow infinitely; therefore, its height is often bounded to prevent infinite expansion. These unfolding trees are significant because they capture all possible paths from the root vertex, effectively encoding the neighborhood structure of the graph from the perspective of the root.

2.3. Methods of Graph Comparison

Often as part of graph learning, we must compare graphs. The simplest method is a binary comparison: the graphs are either structurally identical or they are not. Graphs, however, may be presented in many formats, so to prove whether two graphs are structurally equal, also called isomorphic, can be difficult, and we must show there exists a mapping between them which preserves all relationships. If this mapping does not exist, the graphs are considered not isomorphic.

Definition 2.3.1 (Isomorphic)
Two graphs G and G' are said to be isomorphic (to each other) if there is a one-to-one correspondence between their vertices and between their edges such that the incidence relationship is preserved. In other words, suppose that edge e is incident on vertices v_1 and v_2 in G; then the corresponding edge e' in G' must be incident on the vertices v_1' and v_2' that correspond to v_1 and v_2, respectively.

There is no one best test for isomorphism, and no polynomial-time solution is known. As such, there are many methods which have been created to check for isomorphism with varying degrees of success.

2.3.1. Color Refinement

A common process of graph isomorphism testing is to generate labels for each vertex in the graph based on its neighbors and then compare these label sets. One of the most popular methods is the Weisfeiler-Leman (WL) color refinement algorithm, created in 1968 by B. Weisfeiler and A. Leman. In this algorithm, vertices are assigned colors, which are a hashing of the vertex's label and its neighbor's labels, with an iteration count that describes the distance from the vertex to its furthest neighbors. If the two graphs have different colorings at any iteration, then the algorithm finds that the graphs are not isomorphic. Though this isomorphism test has been proven to fail in a small number of cases, e.g., in distinguishing strongly regular graphs, it works in nearly all instances and has thus been used widely.

Algorithm 1 WL Color Refinement

Input : Graph $G = (V, E)$ with labeled or unlabeled vertices, *max_iterations*
Output : Refined vertex coloring after *max_iterations*

```
1 Assign each vertex an initial color based on its label
2 if all vertices are unlabeled then
3 |  Assign the same initial color to all vertices
4 for iteration ← 1 to max_iterations do
5 |  foreach vertex v ∈ V do
6 |  |  Assign new color to v based on its current label and multiset of colors of its
  |  |  neighbors
```

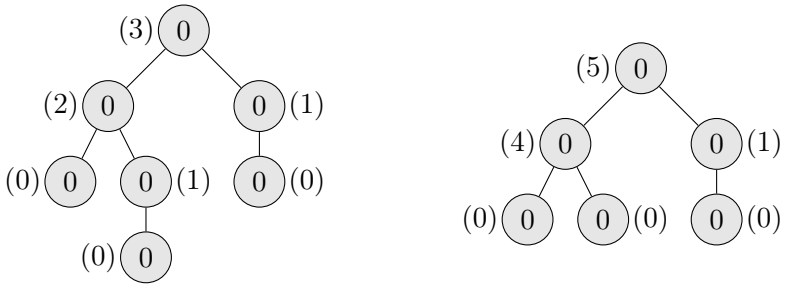


Figure 2.3.: Example Two Trees with Unique Canonical Labels. The two trees share identical subtrees that have the same canonical label, but the overall tree is different so the final label for each tree is unique. The canonical label for each subtree is indicated in () next to the root node of the subtree.

2.3.2. Canonicalization

Trees can also be generated from graphs to capture relationships around vertices in a non-cyclical manner that may be easier to compare. Still, comparing data structured as a tree can be complicated, but there exists a process called canonicalization by which a tree can be converted into a string representing its entire structure. A canonical label is generated which is unique to the structure and, if the process is repeated would result in the same label. The process follows a bottom-up approach where each leaf is assigned a canonical label based on its vertex label, then the parents of the leaves are assigned a label based on their children's canonical labels, which are sorted in order to assure reproducibility. Pseudo-code for this algorithm can be found in Appendix A.1.1.

When the tree used in canonicalization is the Unfolding Tree (see Section 3.1.4) then the results of the canonicalization will be identical to WL color refinement where the maximum height of the tree equals the number of iterations in WL color refinement - 1.

2.4. Machine Learning on Graphs

Many machine learning techniques can also be used on graphs, including kernel methods. This is essential to graph learning, as in many cases, binary distinction is too strict for our graph comparisons. One method in such scenarios is to use graph kernels, which instead provide a continuous similarity measure.

Definition 2.4.1 (Graph Kernel) *(Nikolentzos et al., 2021)*
A graph kernel is a symmetric, positive semi-definite function defined on the space of graphs $\mathfrak{G}$. This function can be expressed as an inner product in some Hilbert space. Specifically, given a kernel k, there exists a map $\phi : \mathfrak{G} \to \mathfrak{H}$ into a Hilbert space $\mathfrak{H}$ such that $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$ for all $G_1, G_2 \in G$. Roughly speaking, a graph kernel is a measure of similarity between graphs.

The function at the core of a graph kernel can be based on many different attributes of the graph. Some such kernels are random walk kernels (the similarity measure comes

from the number of common paths two graphs share) (Gärtner et al., 2003), cyclic pattern kernels (similarity is based on count of common patterns across two graphs) (Horváth et al., 2004), and shortest-path kernels (similarity is measured by common traits in shortest paths of both graphs) (Borgwardt and Kriegel, 2005). However, in this work we focus on subtree and assignment kernels. Subtree kernels measure graph similarity by comparing all pairs of vertices from two graphs through the iterative comparison of their neighborhoods (Nikolentzos et al., 2021), and we examine several methods of neighborhood exploration. Assignment kernels measure similarity by computing a matching between substructures of two graphs to maximize the overall similarity of the graphs (Nikolentzos et al., 2021) and we explore the effect of utilizing different substructures in the form of different neighborhood-encoding trees.

We can also take these graph kernels and use them with existing machine learning tools for tasks like classification. The tool we utilize in this work is the Support Vector Machine (SVM), used for both binary and multi-class classification. Using a train-test splitting method, part of the kernelized graph data is used to train the model while the rest of the graphs are used for testing. In our experimentation we used the Scikit-learn implementation of the Support Vector Classification (SVC) which does a "one-versus-one" approach to multiclass classification. In this multiclass classification task, each pair of classes is compared with a binary classification. The result of each classification awards a point to that class for that graph, and the graph is assigned to the class with the most points after all comparisons have been completed.

3. Essential Background

To validate the need for better graph learning tools, it is important to recognize and evaluate the existing methods. Our experimentation builds on the three graph kernels described here, which we extend through the integration of the novel trees.

3.1. Weisfeiler-Leman Graph Kernels

As discussed previously, the three state-of-the-art graph kernels that form the basis of our experimentation are derived from the Weisfeiler-Leman color refinement algorithm. All kernels using this same foundation fall under the Weisfeiler-Leman framework and follow a set of rules that dictate how to iteratively compute the kernel.

Definition 3.1.1 (Weisfeiler-Leman Framework) *(Nikolentzos et al., 2021)*
Let k be any kernel for graphs, that we will call the base kernel. Then the Weisfeiler-Leman kernel with h iterations with the base kernel k between two graphs G and G' is defined as

$$k_{WL}(G, G') = k(G_0, G'_0) + k(G_1, G'_1) + \cdots + k(G_h, G'_h)$$

where h is the number of Weisfeiler-Leman iterations. The Weisfeiler-Leman sequences $\{G_0, G_1, \ldots G_h\}$ and $\{G'_0, G'_1, \ldots G'_h\}$ of G and G_0, respectively, denote the successive relabeled graphs obtained by applying h iterations of WL color refinement to the specified graph.

In this work, we examine several kernels utilizing the Weisfeiler-Leman framework with different base kernels.

3.1.1. WL Subtree Kernel

The first series of kernels based around the color refinement algorithm, aiming to reduce the runtime of traditional graph kernels based on subgraph isomorphism and graph edit distances, were created by Shervashidze et al. The Weisfeiler-Leman Subtree (WL-ST) kernel provides the foundation for many newer kernels, including our own. This kernel utilizes color refinement to generate labels for each vertex up to the set iteration count. At each iteration, the kernel forms a histogram of the count of each unique vertex label. The feature vector of the kernel is created by concatenating these histograms.

Definition 3.1.2 (Weisfeiler-Leman Subtree Kernel) *(Shervashidze et al., 2011)*
Let G and G' be graphs. Define $\sum_i \subseteq \sum$ as the set of letters that occur as vertex labels at least once in G or G' at the end of the i-th iteration of the Weisfeiler-Leman algorithm. Let $\sum_0$ be the set of original vertex labels of G and G'. Assume all $\sum_i$ are pairwise disjoint.

Without loss of generality, assume that every $\sum_i = \sigma_{i1}, ..., \sigma_{i|\sum_i|}$ *is ordered. With* $\mathcal{G}$ *as the set of graphs under consideration, define a map* $c_i : \mathcal{G} \times \sum_i \to \mathbb{N}$ *such that* $c_i(G, \sigma_{ij})$ *is the number of occurrences of the letter* σ_{ij} *in the graph* G*. The Weisfeiler-Leman subtree kernel on two graphs* G *and* G' *with* h *iterations is defined as:*

$$K^{(h)}_{WL-S}(G, G') = \langle \phi^{(h)}_{WL-S}(G), \phi^{(h)}_{WL-S}(G') \rangle,$$

where

$$\phi^{(h)}_{WL-S}(G) = (c_0(G, \sigma_{01}), ..., c_0(G, \sigma_{0|\sum_0|}), ..., c_h(G, \sigma_{h1}), ..., c_h(G, \sigma_{h|\sum_h|})),$$

and

$$\phi^{(h)}_{WL-S}(G') = (c_0(G', \sigma_{01}), ..., c_0(G', \sigma_{0|\sum_0|}), ..., c_h(G', \sigma_{h1}), ..., c_h(G', \sigma_{h|\sum_h|})).$$

When it was first introduced, experimentation on the WL Subtree, revealed that, compared to other state-of-the-art kernels of the time, the WL Subtree would outperform in most cases in terms of runtime, but resulted in lower accuracy than that of several other kernel types (Shervashidze et al., 2011). Though no longer as novel as it once was, this kernel provided essential groundwork for future WL kernels which have improved in terms of classification accuracy and runtime.

3.1.2. WL Optimal Assignment Kernel

Another key WL graph kernel is the Weisfeiler-Leman Optimal Assignment (WL-OA) kernel in which color refinement labels are used to create a bijection between graphs as a measure of similarity (Kriege et al., 2016).

Definition 3.1.3 (Weisfeiler-Leman Optimal Assignment Kernel) *(Kriege et al., 2016) The vertex optimal assignment kernel (V-OA) is defined as* $K(G, G') = K^{k_\delta}_{\mathfrak{B}}(V(G), V(G'))$*, where* k_δ *returns 1 if vertex labels are equal and 0 otherwise, and* $\mathfrak{B}$ *is the set of all bijections between* $V(G), V(G')$*. The Weisfeiler-Leman optimal assignment kernel (WL-OA) is defined on the vertices like V-OA, but employs the non-trivial base kernel*

$$k^{(h)}_{WL-OA}(v, v') = \sum_{i=0}^{h} k_\delta(\tau_i(v), \tau_i(v'))$$

where $\tau_i(v)$ *and* $\tau_i(v')$ *are the labels of vertices* v *and* v' *at the* i*-th Weisfeiler-Leman iteration. Instead of comparing only initial labels like V-OA, the WL-OA allows for the comparison of vertex labels across iterations of color refinement.*

In general, Optimal Assignment (OA) kernels are not usable in kernel methods because optimal assignments often yield indefinite functions. However, utilizing the Weisfeiler-Leman color refinement as the base kernel produces a hierarchy that organizes vertex labels into nested subsets based on their refinement over iterations. This ensures the

base kernel is positive semi-definite, and, thus, has a valid inner product in a feature space. This, then, allows for the linear time computation of the kernel by the use of histogram intersection. In testing, Kriege et al. found that the WL-OA kernel significantly outperformed the WL-ST kernel in terms of accuracy on most datasets.

3.1.3. Generalized WL Kernel

The final kernel which we wish to modify is the Generalized Weisfeiler-Leman (GenWL) kernel (Schulz et al., 2022). Extending the kernels created by Shervashidze et al., the GenWL kernel was designed to relax the rigidity of the original kernels by eliminating exact vertex label matching in favor of similar vertex label grouping. The WL Unfolding Trees at each iteration are compared using the structure and depth preserving tree edit distance, where finding the minimum cost matchings is treated as a Wasserstein distance problem. When the labels for a vertex are in the same cluster, as determined by Wasserstein k-means clustering, then they are treated as identical in the kernel computation. When the permissible difference in tree edit distance within a cluster is 0, then the kernel returns the same result as the traditional WL-Subtree kernel.

Definition 3.1.4 (Generalized Weisfeiler-Leman Graph Kernel)

For a set $\mathcal{G}$ of graphs, let Θ_i be a set of hard clustering functions (i.e., partitionings) of the set of depth-h Unfolding Trees $\mathcal{T}^{(h)}$ appearing in the graphs in $\mathcal{G}$. We regard each element of h as a function $\rho : \mathcal{T}^{(h)} \to [k]$, where k is the number of clusters defined by ρ. Then, for any graphs $G, G' \in \mathcal{G}$ and depth parameter h, the relaxed WL-Subtree kernel is defined by

$$K^{(h)}_{R-WL}(G, G') = \sum_{i=0,\ldots,h} \sum_{\rho \in \Theta_h} \sum_{v \in V} \sum_{v \in V'} \delta(\rho(T^h(G, v)), \rho(T^h(G', v'))),$$

where δ is the Kronecker delta. Notice that $K^{(h)}_{R-WL}$ is equivalent to the original WL-Subtree kernel $K^{(h)}_{WL-S}$ for the case that $\Theta_h = \{\rho_h\}$ with ρ_h defined as follows: For all $T, T' \in \mathcal{T}^{(h)}, \rho_h(T) = \rho_h(T')$ if T and T' are isomorphic (or equivalently, $SDTED(T, T') = 0$).

Relaxing this even further, there exists a faster variant of this kernel, the Relaxed Generalized Weisfeiler-Leman (RWL) kernel, to reduce the large amount of pairwise distance computations between every vertex's WL Unfolding Tree. Instead of requiring exact tree edit distances between pairs, an initial clustering is performed and unique WL Unfolding Trees are replaced by the cluster centers for distance computation. In experimentation on a variety of Weisfeiler-Leman graph kernels, the GenWL kernel had the highest classification accuracy on several datasets, and at least comparable accuracy on the rest. However, the GenWL had much higher runtimes than other kernels, particularly the WL-Subtree kernel (Schulz et al., 2022).

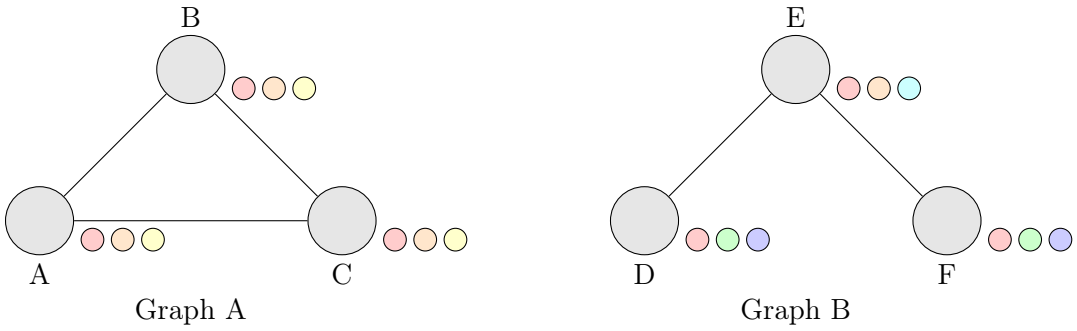


Figure 3.1.: Example Weisfeiler-Leman Color Refinement for 3 Iterations

	Vertices A, B, and C	Vertex E	Vertices D and F
Height 1 Tree	0	0	0
Height 2 Tree	1	1	2
Height 3 Tree	3	4	5

Table 3.1.: Unfolding trees for Graphs A and B of varying heights. A mapping between canonical labels and colors using {0:red, 1:orange, 2:green, 3:yellow, 4:cyan, 5:blue} reveals equivalency between Figure 3.1 and the contents of this table.

3.1.4. Relationship to the Unfolding Tree

As described in the previous chapter, the color refinement algorithm leveraged by these WL kernels uses an iterative approach to view neighbors of a vertex one depth at a time. Without a bounding parameter, the process will continue forever, finding unique colors at each iteration. This process is logically equivalent to generating the Unfolding Tree (UT) for each vertex. Figure 3.1 and Table 3.1 provide an example of this relationship.

Definition 3.1.5 (Unfolding Tree) *(Scarselli et al., 2009),(Bause et al., 2024)*
The neighbors of a vertex $u \in V$ are denoted by $N(u) = \{v | \{u, v\} \in E\}$. Let $\phi : V(T) \to V(G)$ be a mapping such that $\phi(n) = v$ if the vertex n in $V(T)$ represents the vertex v in $V(G)$. The Unfolding Tree $UT^h_{(G,v)}$ with height h of the vertex $v \in V(G)$ is defined recursively as the tree with a root n_v and child subtrees $UT^{h-1}_{(G,w)}$ for all $w \in N(v)$, and $UT^0_v = (n_v, \emptyset)$.

The Breadth-First Search (BFS) algorithm begins at a set vertex v by scanning its

neighbors, then scanning the neighbors of those neighbors, and repeating the process until every edge in the graph is scanned (Deo, 2016). The Unfolding Tree can be generated quite similarly, the only difference being that edges can be scanned repeatedly, until the reached height h is reached, if this maximum height is set. This process will capture all neighbor relationships, but it exhibits high redundancy, causing the Unfolding Trees to require large amounts of storage and have long runtimes for generation and in canonicalization.

3.2. Novel Trees for Graph Comparison

The job of a graph kernel is to provide a non-trivial similarity measure between graphs. In Weisfeiler-Leman graph kernels, this is achieved through color refinement, where labels encode neighborhood relationships to provide the basis for comparison. By employing recently developed neighborhood-encoding trees, we leverage new methods of capturing neighborhood relationships, providing objects for comparison beyond the traditional Unfolding Tree.

3.2.1. Truncated ePath Tree

One tool that can be employed to capture cycles within a graph while reducing redundancy is the extended path (epath) which is a special case of a path that does not allow for repetition of vertices except to indicate there is a cycle in the graph.

Definition 3.2.1 (Extended Path, epath) *(Chen et al., 2022)*
An epath is a path that has no repeated vertex along the path except that the starting vertex is allowed to be the ending vertex when the length of the path is larger than 2.

One such neighborhood-encoding tree is the Truncated ePath Tree (TPT), which is a modified BFS tree where paths are truncated when there is a repeated vertex, with an exception to preserve ring structures (Chen et al., 2022). Figure 3.2 gives an example of the TPT with no bounds on height for the vertex A in graph in Figure 2.1 and pseudocode to generate the TPT can be found in Appendix A.1.2.

Definition 3.2.2 (Truncated ePath Tree (TPT)) *(Chen et al., 2022)*
Given a graph $G = (V, E)$ and a vertex $v \in V$, the height h TPT, $TPT^h_{(G,v)}$, is an epath search tree obtained by running a BFS from v, where all epaths of length up to k are accessed.

3.2.2. k-Redundant Neighborhood Tree

Another type of neighborhood-encoding tree that is used to measure similarities between graphs is the k-Redundant Neighborhood Tree (kNT). This tree balances accuracy and runtime compared to the Unfolding Tree and TPT by deleting vertices and their subtrees based on the vertices present in the levels above them. An example of the 0NT and 1NT can be found in Figure 3.3 and pseudocode can be found in Appendix A.1.3.

Definition 3.2.3 (k-Redundant Neighborhood Tree (kNT)) *(Bause et al., 2024)*

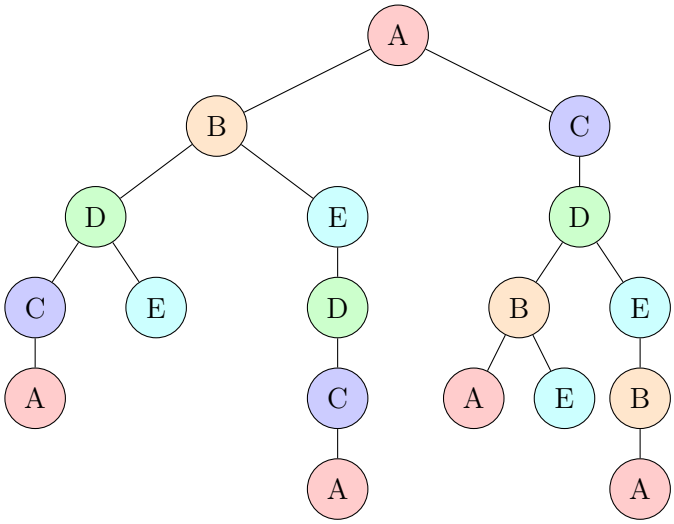


Figure 3.2.: Example TPT: TPT with Unbounded Height for Vertex A in Figure 2.1

For $k \geq 0$, the k-redundant neighborhood tree of a vertex $v \in V(G)$ with height h, denoted by $kNT^h_{(G,v,k)}$, is defined as the subtree of the Unfolding Tree $UT^h_{(G,v)}$ consisting of the vertices $u \in V(UT^h_{(G,v)})$ satisfying

$$\forall w \in V(UT^h_{(G,v)}) : \phi(u) = \phi(w) \implies depth(u) \leq depth(w) + k$$

Where depth is the length of the path from v to the root and $\phi(v)$ is the original vertex in $V(G)$. Note that for $k \geq h$ the k-redundant neighborhood tree is equivalent to the Weisfeiler-Leman Unfolding Tree with height h.

(a) k=0-NT (b) k=1-NT

Figure 3.3.: Example kNTs: kNTs for Vertex A in Graph in Figure 2.1

There exist vertices that the Unfolding Tree cannot distinguish but kNTs can (though the opposite is true, as well), meaning that kNTs can be more expressive on the graph level (Bause et al., 2024). Throughout our experimentation we use only the value $k = 1$

(called 1NT) when generating our kNTS, as our previous experimentation (see Appendix A.2) showed us that $k = 1$ has a high distinguishing capability while maintaining low redundancy.

Each graph kernel and neighborhood-encoding tree examined here has contributed significantly to graph learning research. All three Weisfeiler-Leman kernels have been considered state-of-the-art, yet have shortcomings. We hope that by combining elements of these graph learning methods, we are able to capitalize on each of their strengths and create new graph kernels that show improved performance. By replacing the Weisfeiler-Leman coloring with innovative neighborhood-encoding trees, we aim to improve upon these Weisfeiler-Leman kernels in terms of accuracy and/or runtime.

4. Combining Neighborhood-Encoding Trees and Graph Kernels

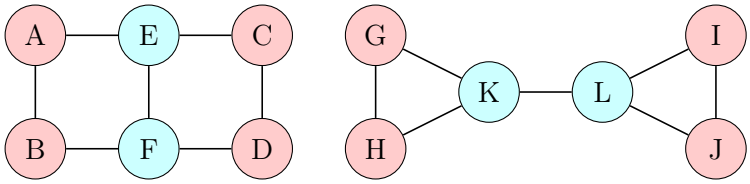
The main contribution of this work is the integration of the three previously introduced Weisfeiler-Leman graph kernels with the two neighborhood-encoding trees. In this section, we describe the methodology behind the modified kernels.

4.1. Design of the Modified Kernels

In the WL Subtree kernel, vertices are given labels during each iteration of color refinement ($i = 1, 2, ..., max\_iterations$). These label counts are gathered into histograms for each iteration, which are then combined to form a feature vector. In our implementation of the modified Subtree kernel, instead of these color refinement labels, we generate a tree per iteration (where the maximum tree height is equal to the number of iterations - 1) for the three tested tree types (UT, 1NT, and TPT), canonicalized the trees for that height, and created a histogram vector, as well. We then compute the kernel dot products the same as in the WL Subtree kernel. When the UT is used in the modified Subtree kernel, it yields a histogram vector which is the same as the WL Subtree kernel up to permutation of label IDs. As dot products are invariant to such permutations, the modified WL Subtree kernel using UTs produces the same kernel matrix as the WL Subtree kernel.

The WL Optimal Assignment kernel is computed similarly to the WL Subtree kernel, but instead of using the dot product of histogram vectors, the kernel is computed via the intersection of the histograms to find the optimal assignment matching. Much like the WL Subtree kernel, the WL Optimal Assignment kernel uses color refinement to generate labels for each vertex at each iteration which are then used to create the vector histograms for the histogram intersection. In our modified Optimal Assignment kernel, we replace the color refinement with a neighborhood-encoding tree with a maximum height 1 less than the count of iterations. We use the UT, kNT, and TPT as the different neighborhood-encoding trees. When using the UT in our modified Optimal Assignment kernel, the feature vectors are equal to those of the WL-OA kernel, up to permutation. Optimal assignments are invariant to such permutations, and, thus, our modified Optimal Assignment kernel and the WL-OA kernel produce the same kernel matrix.

The modified implementation of the Generalized WL kernel follows similarly to the previous two kernels, though there are more steps around the vertex labeling. In the Generalized WL kernel, each vertex is not given a single relabel to capture it and its neighbors' structure-encoding labels. Instead, each vertex is described by its original label and a tuple of its neighbors' labels, which have been calculated up to the maximum



Graph 1 Graph 2

Figure 4.1.: Illustrative Graph Pair for WL vs. Neighborhood-Encoding Trees. Vertex labels are indicated by vertex colors and the letters inside provide unique labels for identification purposes only.

tree height - 1. With the maximum tree height equal to 0, each vertex's neighbors are considered by an empty tuple. In our modified Generalized kernel, the equivalent descriptor is the vertex's original vertex label and a tuple of its neighbors' labels. We find the neighbors' labels by constructing a neighborhood-encoding tree up to the maximum height for the vertex we are relabeling and then canonicalizing the root's immediate children. When working with maximum tree height 0, our implementation takes the neighbor labels as an empty tuple, like the GenWL. Our implementation allows for the use of the UT, kNT, or TPT as the neighborhood-encoding tree which can be used to capture the relationships of each neighbor of the given vertex. After each vertex has been relabeled using this descriptor (called the concatenated label), the set of all concatenated labels is sorted and used as the feature vector for the kernel. These feature vectors are used in conjunction with a ground cost matrix to generate a pairwise distance matrix for each tree height. Once all matrices are created, we go back through the tree heights using Wasserstein k-means clustering to assign each node to a cluster and then assign the cluster label to each node. The new node labels are then used to make the feature vector similar to the Subtree kernels with per height histograms concatenated across heights. After the generation of the concatenated label set, the GenWL and modified Generalized kernels are processed the same. The feature vectors for the kernels are the same up to permutation, so when the cost matrices are computed, the GenWL and UT variant of the Generalized kernel will produce the same results.

4.2. Example of the Modified Subtree Kernel

To show plainly the motivation for modifying the kernels, we show here the difference between the results of the WL Subtree kernel and the modified Subtree kernel on two simple graphs, which can be found in Figure 4.1. To demonstrate the Weisfeiler-Leman kernels, we generate the feature vectors for the WL and modified Subtree kernels for both of these graphs. These Subtree kernels generate a histogram of vertex labels and append it to the feature vector from the previous iteration. Here, instead of performing color refinement, we examine the UTs that are implicitly created from the process, and we use the canonical labels (denoted in parentheses to the left of each node, capturing

the structure of each subtree) representing the tree structures in lieu of color labels. We generate each kernel up to a maximum tree height of 3, showing the intermediate trees and feature vector generation at each step.

Due to the symmetry of the graphs, many vertices will always have identical neighborhood structures and, thus, will produce the same WL colors and neighborhood-encoding trees at each iteration. The vertex sets {A, B, C, D} and {E, F} from Graph 1 will each have only one distinct label. The same is true for the vertex sets {G, H, I, J} and {K, L} from Graph 2. For visualization purposes, rather than repeating identical plots, we only plot one representative tree per set.

First, we consider the WL Subtree kernel by examining the UTs, shown in Figures 4.2 and 4.3. Starting with a maximum tree height of 0, the histogram of canonical labels for Graph 1 is {(0): 4, (1): 2} and the histogram for Graph 2 is {(0): 4, (1): 2}. Converting these into vector form gives the feature vector [4,2] for Graph 1 and [4,2] for Graph 2 for the initial iteration of the WL Subtree kernel. These feature vectors are identical, meaning that the WL Subtree kernel has not identified these graphs as being different.

Continuing to maximum tree height=1, the histogram for Graph 1 is {(2):4, (3): 2} and the histogram for Graph 2 is {(2): 4, (3): 2}. Converting these into vector form gives [4,2] for Graph 1 and [4,2] for Graph 2, which we concatenate with the feature vector from the previous iteration to get [4,2,4,2] for Graph 1 and [4,2,4,2] for Graph 2. The feature vectors are again identical meaning that the WL Subtree kernel has still not distinguished the two graphs. At maximum tree height=2, Graph 1 has histogram: 4: 4, 5:2 and Graph 2 has the histogram: 4: 4, 5:2, so both graphs receive the feature vector [4,2,4,2,4,2]. Even at maximum tree height=3, the UTs of the two graphs are identical. The histogram for Graph 1 is {(6):4, (7):2} and for Graph 2 it is {(6):4, (7):2}, and they both end with the feature vector [4,2,4,2,4,2,4,2], meaning that the WL Subtree kernel (and the color refinement algorithm) does not distinguish the two graphs.

When we use the 1NT to demonstrate the modified Subtree kernel (see Figure 4.4), and we initially find a similar pattern. For maximum tree heights 0 and 1, all 1NTs are identical to the UTs of the WL Subtree kernel at heights 0 and 1. The feature vectors for both graphs are [4,2] and [4,2,4,2] at maximum tree height 0 and 1, respectively. At maximum tree height=2, we see the first deviations from the UTs and encounter new canonical labels. The histogram for Graph 1 at this height is {(10):4, (13):2} and the histogram for Graph 2 is {(10):4, (13):2}, but because these histograms are again identical, both graphs have the feature vector [4,2,4,2,4,2].

Maximum tree height 3, however, demonstrates why we chose to explore the replacement of WL color refinement with canonicalized 1NTs. At previous heights of 1NTs and in all UTs, vertices A, B, C, D from Graph 1 and vertices G, H, I, J from Graph 2 were labeled the same, and vertices E, F from Graph 1 and K, L from Graph 2 were labeled the same. When we reach maximum tree height=3 using 1NTs, the vertices from Graph 1 and Graph 2 do not exhibit the same 1NT structure; Graph 1 has 4 vertices with the canonical label (17) and 2 vertices with the canonical label (13) while Graph 2 has 4 vertices with the canonical label (21) and 2 vertices with the canonical label (23). The histograms generated are now {(13): 0, (17): 4, (21): 2, (23): 0} for Graph 1 and {(13):

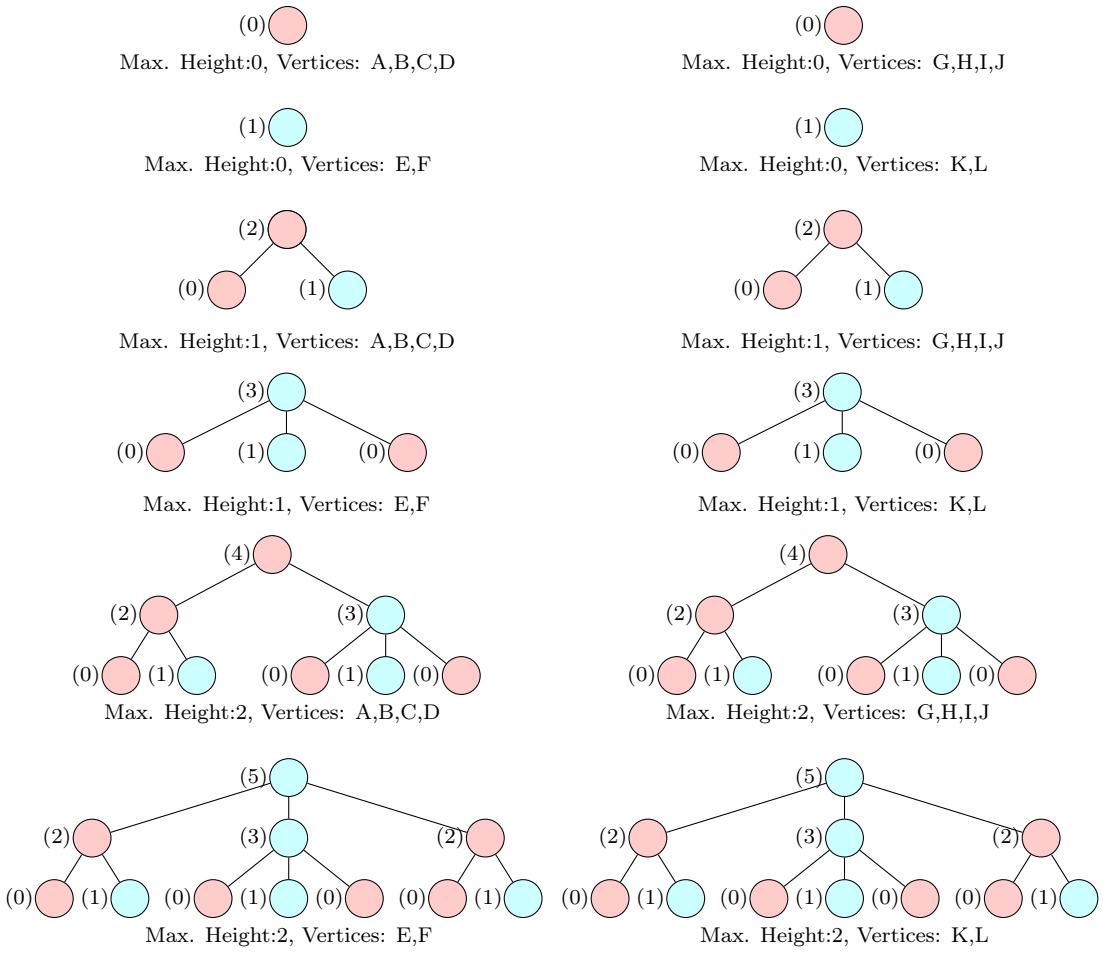


Figure 4.2.: Max. Height 0, 1, and 2 UTs for Graphs in Figure 4.1

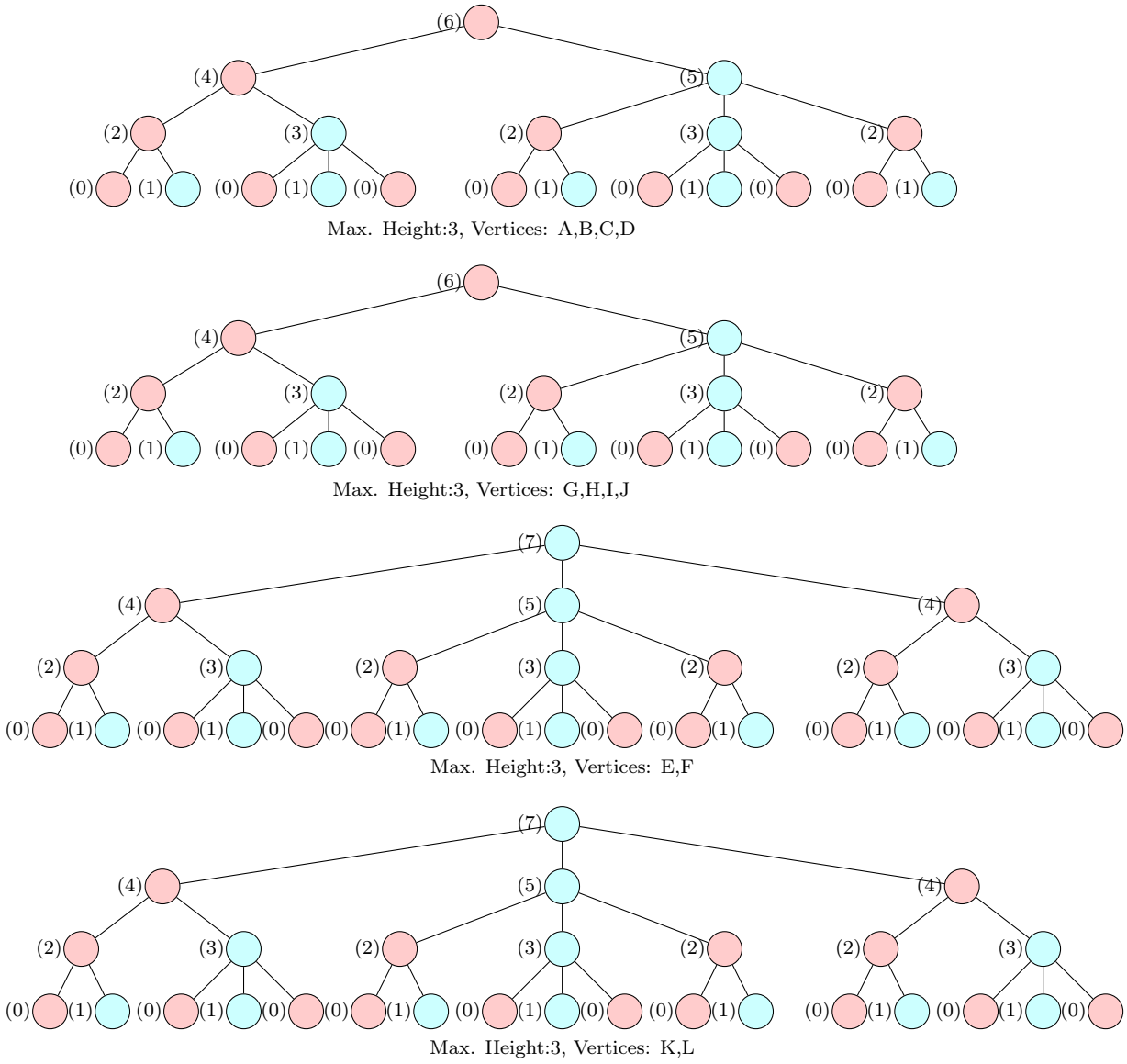


Figure 4.3.: Max. Height 3 UTs for Graphs in Figure 4.1

4, (17): 0, (21): 0, (23): 2} for Graph 2. When we use these histograms to generate the feature vectors, we find, for the first time, that the two graphs are not the same: Graph 1 has feature vector [4,2,4,2,4,2,0,4,2,0] and Graph 2 has feature vector [4,2,4,2,4,2,4,0,0,2].

The TPTs, shown in Figure 4.5, tell a very similar story to the 1NTs. Again, for both graphs, the maximum height 0 and 1 trees are identical to the UTs and 1NTs of the same height, leading to the feature vectors [4,2] and [4,2,4,2], respectively, for these heights. At maximum height 2, the TPTs are identical to the height 2 1NTs, resulting in the same histograms and feature vector (which is [4,2,4,2,4,2]) for both graphs. Similar to the 1NTs, the set of maximum height 3 TPTs is not the same for Graph 1 and Graph 2. Although the vertices E, F from Graph 1 and K, L from Graph 2 do have identical height 3 TPTs, the other vertices from the two graphs do not. The histogram of maximum height 3 TPTs for Graph 1 is {(26): 4, (27): 0, (30): 2} and the histogram for Graph 2 is {(26): 0, (27): 4, (30): 2}. This leads to the feature vectors [4,2,4,2,4,2,4,0,2] and [4,2,4,2,4,2,0,4,2] for Graph 1 and Graph 2, respectively.

Examining the pairs of feature vectors for the different variants of the maximum height 3 Subtree kernels, we find that the dot product for the WL Subtree kernel is 80, the dot product for the modified Subtree kernel employing the 1NT is 60, and the dot product for the modified Subtree kernel employing the TPT is 64.

These results provide strong encouragement for combining the neighborhood-encoding trees with the WL graph kernels. In cases where color refinement fails to distinguish graphs, the 1NT or TPT may be capable of making these distinctions. And, like with this example, the 1NT and TPT may produce different measures of similarity, further motivating us to examine the classification accuracies of the kernels when utilizing these two neighborhood-encoding trees.

4.3. Code for the Modified Kernels

The code we modified to create our variants of the Subtree and Optimal Assignment kernels comes from the GraKeL library (Siglidis et al., 2020) and the code we modified to create the variants of the Generalized WL kernel comes from the MLAI-Bonn GitHub repository GenWL (mlai-bonn, 2021).

In previous experimentation (see Appendix A.2), we developed code to generate UTs, kNTs, and TPTs and for the canonicalization of trees. In this thesis, we adapted our existing code to work seamlessly with the code for the Weisfeiler-Leman kernels. From the WL kernels, we removed all instances of WL color refinement and replaced them with canonical forms of neighborhood-encoding trees, as described in Section 4.1. After adapting the code to produce our modified kernels, we performed experiments testing runtime and classification accuracies for each kernel on different datasets.

All code used to generate our results can be found at: https://github.com/acf-unive/NeighborhoodTreeKernels. Each file contains a description explaining if the code is original to our work or adapted from another source. If the code is adapted, we note which sections of the code have been changed from the original source.

Figure 4.4.: Max. Height 0, 1, 2, and 3 1NTs for Graphs in Figure 4.1

Figure 4.5.: Max. Height 0, 1, 2, and 3 TPTs for Graphs in Figure 4.1

5. Experimentation and Results

5.1. Experimental Setup and Data Used

For all three kernels, we replaced the structures captured by WL color refinement with canonicalized neighborhood-encoding trees. We tested with the Unfolding Tree structure to confirm that the accuracy results were equal to the original kernel results, as the two kernels are theoretically identical when the number of iterations is equal to the height of the Unfolding Tree - 1. After this, we proceeded testing using the canonicalized 1NTs and TPTs as vertex labels.

The two types of datasets we used to test our kernels are simple molecular benchmark datasets and large network benchmark datasets which were experimentally shown to be best classified by Weisfeiler-Leman graph kernels (Kriege et al., 2020). The used simple molecular graph datasets are MUTAG (Debnath et al., 1991), NCI1 (Wale and Karypis, 2006), and PTC-MR (Helma et al., 2001), which all are sourced from TU Dortmund's Datasets library, available at graphlearning.io (Morris et al., 2020). The benchmark graph datasets used are IMDB-BINARY (Yanardag and Vishwanathan, 2015), EGONETS-1, and EGONETS-2, which are random EGONETS network graphs sampled from Buzznet, Digg, Flikr, and Livejournal by Schulz et al. (2022). These datasets were accessed via the MLAI-Bonn GitHub for the GenWL kernel (mlai-bonn, 2021) and statistics about the datasets can be found in the table 5.1.

5.2. Experimental Method

To get classification accuracy results, we trained an SVM on the precomputed kernels matrices from the modified Subtree, OA, and Generalized kernels and evaluated with 10-fold cross validation. The Generalized kernel utilizes barycenters for clustering, and their initialization has a significant amount randomness, so we did 3 runs of 10-fold cross validation for this kernel. As we were interested in seeing how the original WL kernels compare to our new variants, we ran the same experiments on the WL kernels and the new variants.

For all three kernel types, we used maximum tree height values ranging from 0 to 5 for the molecular datasets and values ranging 0 to 3 for the network datasets. For each maximum tree height, we canonicalized UTs, 1NTs, and TPTs to generate vertex labels which were passed through the kernel in place of the WL color refinement labels. Apart from the generation of labels, we kept any other hyperparameters of the kernel the same so we could evaluate only the effect of these varying neighborhood-encoding structures.

Dataset	$\lvert D\rvert$	$\lvert C\rvert$	$\overline{\lvert V\rvert}$	$\overline{\lvert E\rvert}$	$\frac{\overline{\lvert E\rvert}}{\overline{\lvert V\rvert}}$	Directed or Undirected	Σ_0^{AT}	Σ_1^{AT}	Σ_2^{AT}
MUTAG	188	2	17.9	19.8	1.1	Undirected	7	33	174
NCI1	4110	2	29.9	32.3	1.1	Undirected	37	292	4058
PTC-MR	344	2	14.29	14.69	1.0	Undirected	18	130	780
EGONETS-1	200	4	139.0	594.6	4.3	Directed	1	112	21k
EGONETS-2	200	4	178.6	1445.0	8.1	Directed	1	140	33k
IMDB	1000	2	19.8	96.5	4.9	Undirected	1	65	2931

(a) Dataset statistics.

Dataset	Σ_3^{UT}	Σ_4^{UT}	Σ_5^{UT}	Σ_3^{1NT}	Σ_4^{1NT}	Σ_5^{1NT}	Σ_3^{TPT}	Σ_4^{TPT}	Σ_5^{TPT}
MUTAG	572	1197	1766	572	1206	1760	572	1197	1779
NCI1	23k	$45k$	$59k$	$23k$	$45k$	$59k$	$23k$	$45k$	$59k$
PTC-MR	1987	2803	3168	1988	2805	3166	1990	2812	3204
EGONETS-1	$25k$	-	-	$25k$	-	-	$25k$	-	-
EGONETS-2	35k	-	-	$35k$	-	-	$35k$	-	-
IMDB	3595	-	-	3595	-	-	$15k$	-	-

(b) Unique vertex label counts for given tree and height.

Table 5.1.: Dataset properties and label statistics. $\lvert D\rvert$ denotes the number of graphs in the dataset and $\lvert C\rvert$ is the number of distinct graph labels; $\overline{\lvert V\rvert}$ and $\overline{\lvert E\rvert}$ denote average number of vertices and edges; Σ_h^T refers to the count of unique vertex labels for tree T of height h; AT refers to all tested tree types (UT, 1NT, TPT).

5.3. Evaluation Metrics

We measure accuracy as the average and standard deviation of accuracy scores resulting from 10-fold cross validation for a classification task. The runtime for each kernel is measured as the time it takes to run the entire process of kernel transformation, classification, and evaluation, and it does not include the time it takes to read in the data.

5.4. Challenges Faced

The network benchmark datasets were very large and, as such, we were unable to use them for very high trees on certain kernels. We were only able to use the EGONETS-2 and IMDB datasets on trees with a maximum height of 3 for all kernel variants. The NCI1 and EGONETS-1 datasets also proved difficult for use in the Generalized kernel variants, so we only used a highest maximum tree height of 3 for this kernel. In all these cases, we abandoned the pursuit of results for further heights when it took over 6 hours to get the results for a single kernel variant.

We were also hindered by the inability to replicate the logic of the approximation of the

Generalized WL kernel, so we could only modify the slower, non-approximated GenWL kernel for use with the neighborhood-encoding trees. Beyond the GenWL kernel, its authors, Schulz et al., proposed an approximated version which would reduce runtimes by clustering the WL color labels at each iteration in order to have fewer labels to cluster at the following iteration. However, in our kernel variants, we must use unique trees (not dependent on previous results) at each iteration, so we cannot retain iteration-to-iteration results like this. Though WL color refinement and the UT are able to reference previous iteration labels as the neighbor labels for a vertex in the current iteration, we cannot necessarily do the same for the neighborhood-encoding trees. When working with 1NTs and TPTs, there are stopping conditions which alter subtree construction, so the label of a subtree rooted at a child vertex (a neighbor label) is not guaranteed to match that vertex's label from the previous iteration.

For 1NTs, vertices can only reappear up to k levels higher in the tree. If a 1NT is added as a child subtree of another vertex, this may result in a tree which is not also a 1NT. In our experimentation we are only working with the case $k = 1$, so if the new root vertex was anywhere else in the 1NT which was added as a child, then the stopping condition would be violated, and the tree would not be a valid 1NT.

In the case of TPTs, the only vertex that can appear more than once in any path is the root, which can also be a leaf in the tree, given the length of the path is 2 or greater. However, if a TPT is added as a child of a root, this may cause the stopping condition to be violated (ex. the vertex that was a root in a previous iteration appeared at the start and end of a path, but now that vertex is a child of the new root, and the vertex appears twice in the middle of a path, which is not allowed), resulting in a tree that is not a valid TPT.

5.5. Results and Analysis

5.5.1. Subtree Kernel

Figure 5.1 shows the runtimes and accuracies on each dataset for each of the variants of the Subtree kernel.

For the Subtree kernel variants, the accuracies were nearly identical for all variants. The datasets used were real-world datasets, so it makes sense that the kernel variants could all classify the graphs with similar accuracy. The high consistency was also to be expected for this kernel, as there is no randomness throughout the kernelization process, and we did 10 fold cross validation with no shuffling.

The runtime for the WL Subtree kernel was always lower than any of our new neighborhood-encoding variants. The UT and TPT kernel variants had the longest runtimes of all kernels, especially as the number of iterations increased. This makes sense because the they are the larger than kNTs and, as such, should take longer to canonicalize. The two cases where the TPT variants were consistently slower than the UT variants were the EGONETS-1 and EGONETS-2 network datasets. When generating both UTs and TPTs, we use a form of the BFS approach, but the TPT has to do additional checks

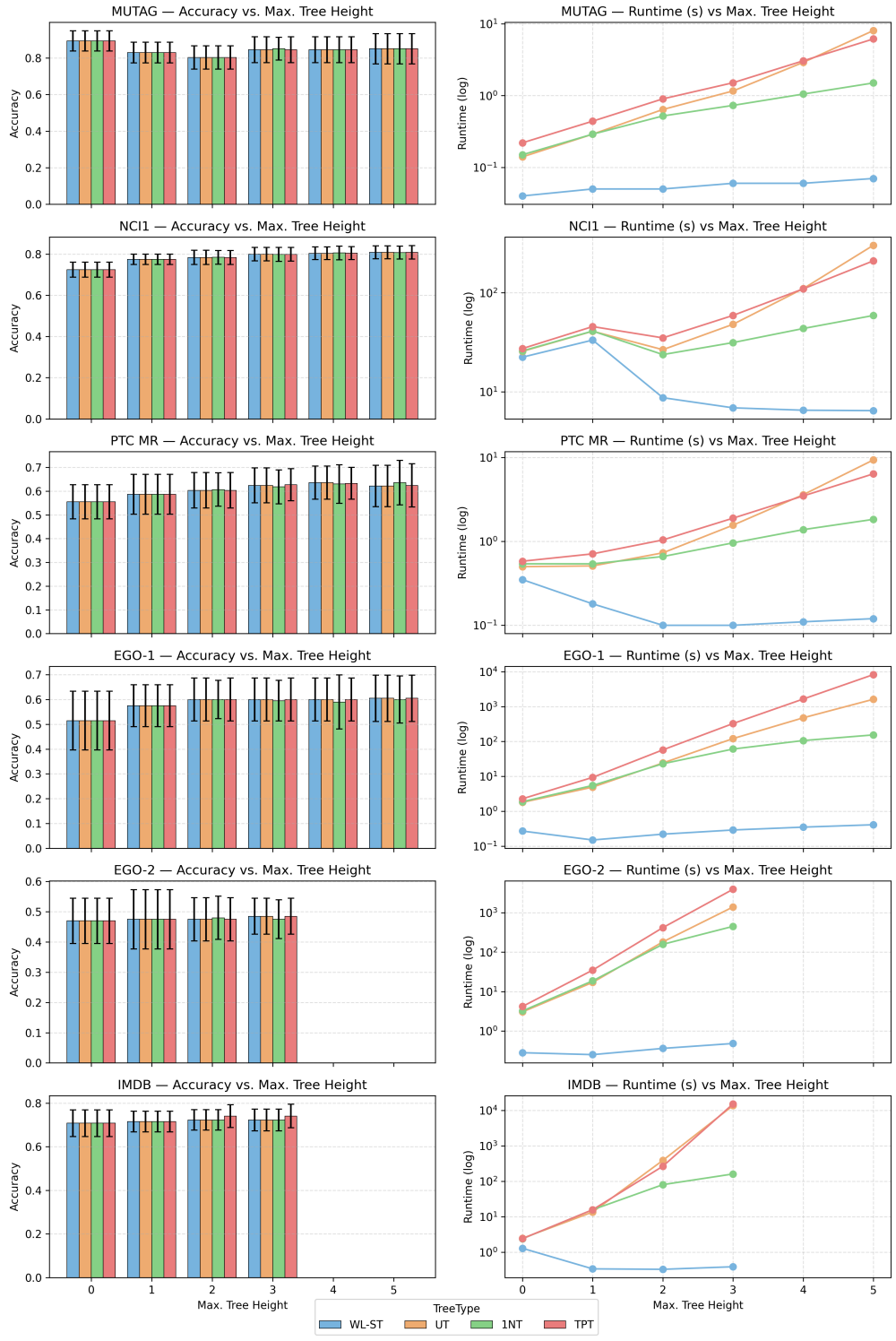


Figure 5.1.: Subtree Kernel Variants Accuracy and Runtime Results

to make sure that the vertex to be added does not already exist in the path. When there are graphs with many edges, like in these two datasets, the number of checks becomes very high and can prove to be quite costly.

An interesting result that can be seen in both the WL Subtree kernel and our modified Subtree kernel is that the runtimes appear to show improvement as the maximum tree height is increased, which is particularly apparent in the jumpt from 1 to 2 on the NCI1 dataset. The result seems counterintuitive, as larger trees seem like they should lead to longer runtimes. However, our calculation of runtime is based on the time to build the kernel and train the SVMs, and the time to train the SVMs dominates the overall runtime. For all Subtree kernel variants, the runtime for building the kernel does increase as the maximum tree height increases, but the SVMs take longer to train on the smaller trees, leading to longer runtimes where they may seem unexpected.

5.5.2. Optimal Assignment Kernel

The experiments performed on the Optimal Assignment kernel variants, shown in Figure 5.2, yielded similar results to the Subtree kernel variants.

The Optimal Assignment kernel variants also had little variation in the accuracy results across all kernels. This is, again, to be expected, as the datasets do not contain strongly regular graphs. Much like the modified Subtree kernel, the modified Optimal Assignment kernel accuracies generally increased as the maximum tree height increased.

The differences in runtimes across the kernel variants for the modified Optimal Assignment kernel was very similar to that of the modified Subtree kernel, as well. The original WL Optimal Assignment kernel always had the fastest runtime, and the canonicalized UT kernel variant generally had the longest runtime. Again, on the EGONETS-1 and EGONETS-2 datasets, the TPT had the longest runtimes consistently, likely because of the dense graph structure. Different than the Subtree kernel, the Optimal Assignment kernel variants all show increasing runtimes as the maximum tree height is increased.

The modified Optimal Assignment kernel also produced some results that initially appear to be unexpected. The MUTAG and PTC-MR datasets appear to show the UT kernel variant having a much higher runtime than the other kernels, even the TPT variant. However, the difference in actual runtime for the UT and TPT kernels with maximum tree height = 5 is only $\sim$7 seconds for the MUTAG dataset and $\sim$8 seconds for the PTC-MR dataset. We also would expect that there would be a difference between the runtimes of these kernel variants, as the TPT has smaller, truncated trees, which should be faster to canonicalize. As the maximum tree height increases, the TPT will see more truncation (especially on small graphs with few edges, like in the MUTAG and PTC-MR datasets) and the UT will continue to grow exponentially.

5.5.3. Generalized WL Kernel

Results for the variants of Generalized Weisfeiler-Leman kernel can be found in Figure 5.3.

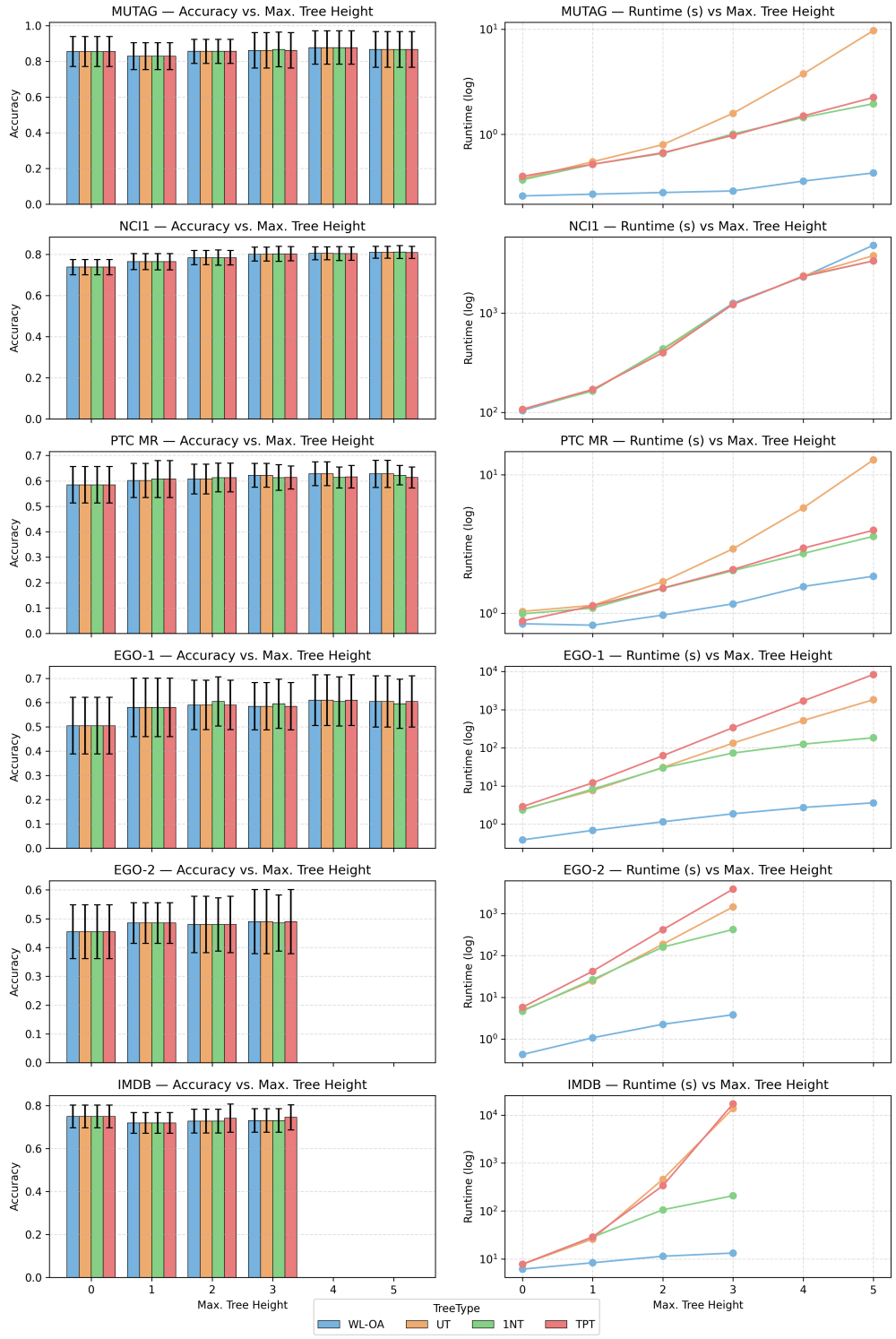


Figure 5.2.: Optimal Assignment Kernel Variants Accuracy and Runtime Results

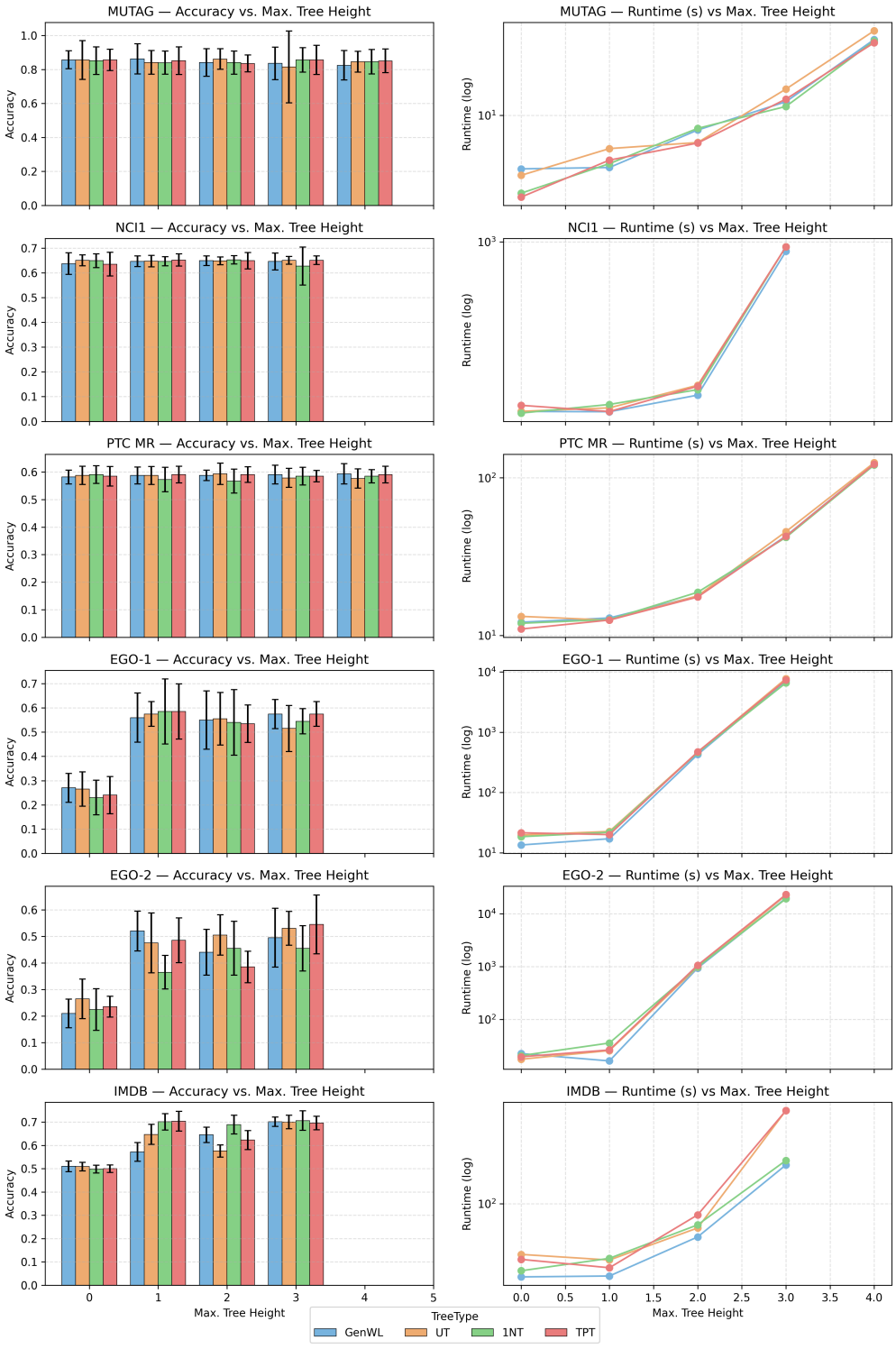


Figure 5.3.: Generalized Kernel Variants Accuracy and Runtime Results

The Generalized kernel exhibited more variable accuracy results than the Subtree or Optimal Assignment kernels, due to the random selection of initial barycenters. Though we did 3 runs of 10 fold cross validation to stabilize the results, the accuracy results still showed high variability.

On the simple molecular graphs, the accuracy results were quite consistent across all kernel variants and with increasing maximum tree height. For the network datasets, increasing the maximum tree height generally lead to higher accuracies. Within a set maximum tree height, there is not much difference between accuracies across the different Generalized kernel variants.

Very different from the Subtree and Optimal Assignment kernels, the modified Generalized kernel variants had much more similar runtimes to the GenWL kernel at each maximum tree heights. However, in the three network datasets, for several maximum tree heights, the GenWL kernel still had faster runtimes than any of our kernel variants.

6. Conclusion

6.1. Summary of Findings

Overall, the modifications we made to the three Weisfeiler-Leman graph kernels did not make significant changes to the classifying capabilities of the kernels. Our accuracy rates were marginally better, at best, and most often were nearly identical to the accuracy rates of the original kernels. A likely explanation for the lack of improvement is that the WL color refinement algorithm already captures the most important neighborhood relationships, so our alternative methods of neighborhood encoding did not offer any benefit for these classification tasks.

In the case of the Subtree and Optimal Assignment kernels, our variants also came at the price of significantly higher runtimes. For these two kernels, this result was expected, as our kernels require the generation and canonicalization of a tree while the WL Subtree and OA kernels are able to reference previous iterations and save on computing efforts. For the Generalized kernels, our runtimes were very similar to the GenWL kernel because the time to generate and canonicalize the trees was outweighed by the clustering runtimes.

6.2. Ideas for Further Research

The most novel of the kernels that we worked with in this thesis was the Generalized WL kernel, but due to incompatibilities in logic (see Section 5.4) we were unable to implement the approximated GenWL kernel which its authors used for their experimentation. Instead, we had to use the much slower non-approximated GenWL kernel and could test on far fewer datasets. As an alternative to the non-feasible approximated GenWL, it would be interesting to investigate other clustering methods for the trees.

One idea that could yield interesting results would be to cluster the canonical labels of some subtrees used to create the vertex label. As the vertices further from the root indicate further away relationships in the original graph, clustering the subtrees further from the root could potentially lead to better classification results. In the case of the approximation of the Generalized WL kernel, one major benefit was the improved runtime, as the labels are combined via clustering, leading to fewer unique labels as iterations continue. This proposal of clustering subtrees of neighborhood-encoding trees would not be able to benefit from these time saving measures, but if the clustering is done in a way that merges many unique subtree labels into fewer labels, there could possibly be an improved runtime over the non-approximated Generalized kernel we used here.

Another consideration that could potentially lead to more interesting results would be to test the kernels on artificial datasets with more graph structures that WL color

refinement cannot distinguish. If the graph datasets contained many strongly regular structures (which WL color refinement cannot distinguish), then the TPT variants of our kernels may have more success in classification tasks. This, however, may not prove to have any value in real-world graph learning. Our previous research also found the TPTs generation for strongly regular graphs had extremely long runtimes, so it may prove very difficult to test.

6.3. Concluding Remarks

Throughout this thesis we examined the effect of new neighborhood-encoding structures on three existing graph kernels built around Weisfeiler-Leman color refinement. Our experimentation revealed there was very little change in the classification accuracy on simple molecular and complex network datasets and that there was no improvement in runtimes. However, this process does inspire the idea to go forth using these novel neighborhood-encoding trees in further deviated variants of the Generalized Weisfeiler-Leman graph kernel. Although the impact on performance shown by our experimentation is limited, we demonstrated the feasibility of integrating kNTs and TPTs as alternatives to WL color refinement within graph kernels, which could provide a framework for further exploration in this direction.

Bibliography

Bause, F., Permann, C., & Kriege, N. M. (2024). Approximating the graph edit distance with compact neighborhood representations. In A. Bifet, J. Davis, T. Krilavičius, M. Kull, E. Ntoutsi & I. Žliobaitė (Eds.), *Machine learning and knowledge discovery in databases. research track* (pp. 300–318). Springer Nature Switzerland.

Borgwardt, K., & Kriegel, H. (2005). Shortest-path kernels on graphs. *Fifth IEEE International Conference on Data Mining (ICDM'05)*, 8 pp.-. https://doi.org/10.1109/ICDM.2005.132

Chen, R., Zhang, S., U, L. H., & Li, Y. (2022). Redundancy-free message passing for graph neural networks. In A. H. Oh, A. Agarwal, D. Belgrave & K. Cho (Eds.), *Advances in neural information processing systems*. https://openreview.net/forum?id=jwVZZzzNKkW

Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., & Hansch, C. (1991). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *J Med Chem*, *34*(2), 786–797. https://doi.org/10.1021/jm00106a046

Debnath, A. K., Lopez de Compadre, R. L., & Hansch, C. (1992). Mutagenicity of quinolines in salmonella typhimurium ta100. a qsar study based on hydrophobicity and molecular orbital determinants. *Mutation Research/Genetic Toxicology*, *280*(1), 55–65.

Deo, N. (2016). *Graph theory: With applications to engineering & computer science*. Dover Publications.

Gärtner, T., Flach, P., & Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In B. Schölkopf & M. K. Warmuth (Eds.), *Learning theory and kernel machines* (pp. 129–143). Springer Berlin Heidelberg.

Helma, C., King, R. D., Kramer, S., & Srinivasan, A. (2001). The predictive toxicology challenge 2000–2001. *Bioinformatics*, *17*(1), 107–108. https://doi.org/10.1093/bioinformatics/17.1.107

Horváth, T., Gärtner, T., & Wrobel, S. (2004). Cyclic pattern kernels for predictive graph mining. *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 158–167. https://doi.org/10.1145/1014052.1014072

Kriege, N. M., Giscard, P.-L., & Wilson, R. C. (2016). On valid optimal assignment kernels and applications to graph classification. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 1623–1631.

Kriege, N. M., Johansson, F. D., & Morris, C. (2020). A survey on graph kernels. *Applied Network Science*, *5*(1), 6. https://doi.org/10.1007/s41109-019-0195-3

mlai-bonn. (2021). GenWL: A generalized Weisfeiler-Lehman graph kernel [Accessed: 2025-09-02].

Morris, C., Kriege, N. M., Bause, F., Kersting, K., Mutzel, P., & Neumann, M. (2020). TUDataset: A collection of benchmark datasets for learning with graphs. *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*. www.graphlearning.io

Nikolentzos, G., Siglidis, G., & Vazirgiannis, M. (2021). Graph kernels: A survey. *Journal of Artificial Intelligence Research*, *72*, 943–1027. https://doi.org/10.1613/jair.1.13225

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). Computational capabilities of graph neural networks. *IEEE Trans Neural Netw*, *20*(1), 81–102. https://doi.org/10.1109/TNN.2008.2005141

Schulz, T. H., Horváth, T., Welke, P., & Wrobel, S. (2022). A generalized Weisfeiler-Lehman Graph Kernel. *Machine Learning*, *111*(7), 2601–2629. https://doi.org/10.1007/s10994-022-06131-w

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., & Borgwardt, K. M. (2011). Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.*, *12*(null), 2539–2561.

Siglidis, G., Nikolentzos, G., Limnios, S., Giatsidis, C., Skianis, K., & Vazirgiannis, M. (2020). Grakel: A graph kernel library in Python. *Journal of Machine Learning Research*, *21*(54), 1–5.

Wale, N., & Karypis, G. (2006). Comparison of descriptor spaces for chemical compound retrieval and classification. *Sixth International Conference on Data Mining (ICDM'06)*, 678–689. https://doi.org/10.1109/ICDM.2006.39

Weisfeiler, B., & Lehman, A. A. (1968). A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia, Ser. 2*(N9), 12–16.

Yanardag, P., & Vishwanathan, S. (2015). Deep graph kernels. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1365–1374. https://doi.org/10.1145/2783258.2783417

Acronyms

BFS Breadth-First Search. 14

GenWL Generalized Weisfeiler-Leman. 13

kNT k-Redundant Neighborhood Tree. 15

OA Optimal Assignment. 12

RWL Relaxed Generalized Weisfeiler-Leman. 13

TPT Truncated ePath Tree. 15

UT Unfolding Tree. 14

WL Weisfeiler-Leman. 7

WL-OA Weisfeiler-Leman Optimal Assignment. 12

WL-ST Weisfeiler-Leman Subtree. 11

A. Appendix

A.1. Additional Pseudo-code for Algorithms

A.1.1. Tree Canonicalization Algorithm

Tree canonicalization is done through a bottom-up approach. Leaves are first canonicalized by hashing their vertex labels. Then, for the parent of each leaf all children are sorted based on their canonical labels and the parent is assigned a new canonical label based on its children and its vertex label. This process is continued until the root of the tree is reached and there is one label which canonicalizes the root, providing a single label which captures the full structure of the tree.

Algorithm 2 Tree Canonicalization

Input : *tree*, *node*, *subtree_dict*, *current_label*
Returns : *canonical_label*

```
7  Function getCanonicalLabel(tree, node, subtree_dict, current_label)
8      if node is leaf then
9          subtree_form ← (node['name'],)
10     else
11         child_labels ← [ ]
12         foreach child in node.children do
13             child_label = getCanonicalLabel(tree, child, subtree_dict, current_label)
14             child_labels.append(child_label)
15         sort child_labels
16         subtree_form ← (node['name'], tuple(child_labels))
17     if subtree_form ∉ subtree_dict then
18         subtree_dict[subtree_form] ← current_label +1
19         current_label ← current_label +1
20     canonical_form ← subtree_dict[subtree_form]
21     return canonical_form
```

A.1.2. TPT Algorithm

Generating a TPT requires following a breadth-first-search style approach. For a given vertex in a graph, we start by creating a tree and adding the vertex as the tree's root. We then add the neighbors of the vertex as children of the root. Then, we add the neighbors of those vertices as children in the tree, and we repeat this process. However, when adding

a vertex to the tree, we must each time check that neither of these conditions are violated: 1. There are no repeating vertices along the path, and 2. Unless the path is longer than 2 and the repeated vertex is the root.

Algorithm 3 Generate Truncated ePath Tree

```
OutputOutput
Input : Graph, StartNode, MaxDepth
Truncated ePath Tree rooted at StartNode
Function buildTPT(Graph, StartNode, MaxDepth)
    Initialize Queue with (StartNode, [StartNode])  IterationCount ← 0
    while Queue not empty do
        Dequeue first element → (CurrentNode, Path)
        Depth ← len(Path) −1
        NodeID ← concatenate(Path, '-')
        NodeName ← last element of NodeID split by '-'
        if Depth = 0 then
            Add NodeID to tree as root
        else
            ParentID ← concatenate(all but last element of Path)
            Add NodeID to tree with CurrentNode as parent
            Add edge (ParentID, NodeID)
        if Depth ≥ MaxDepth then
            continue // skip to next iteration
        Neighbors ← Graph.neighbors(CurrentNode)
        foreach Neighbor in Neighbors do
            Neighbor ← str(Neighbor)
            if Neighbor = StartNode and len(Path) > 2 then
                NewNodeID ← NodeID + Neighbor
                NewNodeName ← last element of NewNodeID split by '-'
                Add NewNodeID to tree with CurrentNode as parent
                Add edge (NodeID, NewNodeID)
                break // stop further exploration
            if Neighbor ∉ Path then
                Append (Neighbor, Path + [Neighbor]) to Queue
```

A.1.3. kNT Algorithm

The k-Redundant Neighborhood Tree is used to eliminate redundancy for any nodes k levels or above in a tree. In practice, this means that when we generate the tree (using a method similar to the BFS) we begin with a node in the graph defined as the root and add the neighbors from the graph to the tree, then the neighbors of those neighbors,

and so on until we reach the maximum height or stop encountering neighbors. There is however, a stopping condition in which we would not add a node to the tree or continue adding the neighbors of that node, and that condition is if the node already appears more than k levels higher in the tree.

Algorithm 4 Generate k-Redundant Neighborhood Tree

Input : Graph, StartNode, k, MaxHeight
Output : k-redundant neighborhood tree rooted at StartNode

```
49 Function buildkNT(Graph, StartNode, k, MaxHeight)
50     Initialize Depths dictionary D with D[StartNode] ← 0
51     Queue ← [StartNode]
52     for i ← 1 to MaxHeight do
53         NextQueue ← [ ]
54         foreach node v in Queue do
55             OriginalNode ← base node from v (split by '_')
56             foreach neighbor u of OriginalNode in Graph do
57                 u ← str(u)
58                 DepthU ← i
59                 if u ∈ D and DepthU > D[u] + k then
60                     continue // skip this neighbor
61                 D[u] ← min(D[u], DepthU) (initialize if not set)
62                 UniqueChild ← u_i_OriginalNode_v  Add UniqueChild as a vertex in the
                    tree
63                 Add edge (v, UniqueChild)
64                 Append UniqueChild to NextQueue
65         Queue ← NextQueue
```

A.2. Previous Research

In this work we make references to our previous work, which was completed in the Summer 2024 semester at the University of Vienna as part of the "Data Analysis Project" course. Here, in a paper entitled "Comparison of Trees Expressiveness", we, the same authors as this thesis, did an experimental evaluation of the expressiveness of 0NTs, 1NTs, and TPTs.

Through this research, we created code to generate k-NTs and TPTs, as well as a function to canonicalize trees. We tested on a variety of different graph types, including Eulerian, perfect, planar, highly irregular, and strongly regular graphs.

Our key findings from this experimentation were that the TPT did not fail to distinguish pairs of graphs of any type and that the 1NT could distinguish graphs which were not strongly regular, but the 0NT failed to distinguish strongly regular graphs and more.

The graph types that the 0NT failed to distinguish that the 1NT succeeded in distin-

guishing included pairs of planar graphs with 8 vertices, Euler graphs with 9 vertices, and connected Euler graphs with 9 vertices, totaling 8 cases. Based on these results, we concluded that the 0NT was less expressive than the 1NT. These results directly led to our decision to experiment using the 1NT instead of the 0NT throughout this thesis.