Learning to Aggregate on Structured Data

Clemens Damke

April 6, 2020

PADERBORN UNIVERSITY
The University for the Information Society
Department of Electrical Engineering,
Computer Science and Mathematics
Warburger Straße 100
33098 Paderborn

INTELLIGENT SYSTEMS
Intelligent Systems and
Machine Learning Group (ISG)

Master Thesis

Learning to Aggregate on Structured Data

Clemens Damke

1. Reviewer Prof. Dr. Eyke Hüllermeier
Intelligent Systems and Machine Learning Group (ISG)
Paderborn University

2. Reviewer Prof. Dr. Axel-Cyrille Ngonga Ngomo
Data Science Group (DICE)
Paderborn University

April 6, 2020

Clemens Damke
Learning to Aggregate on Structured Data
Master Thesis, April 6, 2020
Reviewers: Prof. Dr. Eyke Hüllermeier and Prof. Dr. Axel-Cyrille Ngonga Ngomo
Supervisor: Vitalik Melnikov

Paderborn University
Intelligent Systems and Machine Learning Group (ISG)
Heinz Nixdorf Institute
Department of Electrical Engineering, Computer Science and Mathematics
Warburger Straße 100
33098 Paderborn

Abstract

Contents

1 Introduction

1.1 Motivation

The field of *machine learning* (ML) on graph-structured data has applications in many domains due to the general expressive power of graphs. The three most common types of graph ML problems are

1. **Link prediction:** A graph with an incomplete edge set is given and the missing edges have to be predicted. The suggestion of potential friends in a social network is a typical example for this.

2. **Vertex classification & regression:** Here a class or a score has to be predicted for each vertex of a graph. In social graphs this corresponds to the prediction of properties of individuals, e.g. personal preferences or gender. Another example is the prediction of the amount of traffic at the intersections of a street network.

3. **Graph classification & regression:** In this final problem type a single global class or continuous value has to be predicted for an input graph. The canonical example for this is the prediction of properties of molecule graphs, e.g. the toxicity or solubility of a chemical.

In this thesis we will focus on the last problem type, *graph classification and regression* (GC/GR). An ML method for this problem has to accept graphs of varying size and should be permutation invariant wrt. the order in which graph vertices are provided. Those requirements are not met by the commonly used learners that only accept fixed-size feature vectors as their input, e.g. *logistic regression model*s (LRMs), *support vector machine*s (SVMs) or *multilayer perceptron*s (MLPs).

A GC/GR method has to account for two central aspects of the problem: 1. Local structural analysis and 2. global aggregation. The first aspect is about the extraction of relevant features of substructures of the input graph. The latter is about the way in which the local features are combined into a final class or regression value. The existing GC/GR methods are mostly motivated by local structural graph analysis. The aspect of global aggregation on the other hand is less emphasized by those methods.

There is however a separate branch of research that specifically looks at the problem of learning aggregation functions, called *learning to aggregate* (LTA). Current LTA

approaches explicitly learn an aggregation functions for sets which can be interpreted as graphs without edges. The motivation for this thesis is to generalize LTA from sets to arbitrary graphs. The overall goal is to combine the aggregation learning perspective with existing GC/GR methods.

1.2 Goals

To extend LTA to graphs, three goals have to be achieved:

1. **Formalization of LTA:** Before LTA can be extended, its essential characteristics have to be defined. Those characteristics should provide the terminology to formally capture the differences and similarities between LTA and existing GC/GR methods.

2. **Give an LTA interpretation of GC/GR methods:** Using the LTA formalization, representative GC/GR approaches should be restated as LTA instances. Currently there is no comprehensive formulation of the relation between both fields of research; this is addressed by the the second goal.

3. **Define an LTA method for graphs:** Using the LTA perspective on GC/GR, hidden assumptions of the existing approaches should become clear and in which way they share the assumptions of LTA. The last goal is to use those insights to formulate an LTA-GC/GR method that combines ideas from the existing approaches with the LTA assumptions.

1.3 Structure

Chapter 2: Related Work

Chapter 3: Learning to Aggregate on Graphs

Chapter 4: Learning to Decompose Graphs

Chapter 5: Evaluation

Chapter 6: Conclusion

1.4 Formal Notation

2 Related Work

Before combining LTA and GC/GR as described in section 1.2, we first give an overview of the state-of-the-art in both fields of research. This is done in three steps:

1. We begin with an overview of the existing LTA methods for unstructured inputs.

2. Then we look at the domain of structured inputs. To solve the GC/GR problem, relevant graphs characteristics have to be defined in order to determine the similarity and dissimilarity of graphs. We will look at three approaches for graph characterization: The graph isomorphism test, the so-called Weisfeiler-Lehman coloring and lastly the notion of graph spectra.W

3. Based on the described graph characterization approaches, an overview of current GC/GR methods will then be given.

2.1 Learning to Aggregate

The class of LTA problems was first described by Melnikov and Hüllermeier [MH16]. There an input instance is understood as a composition $C = \{\!\{c_1, \ldots, c_n\}\!\}$ of so-called constituents, i.e. as a variable-size multiset (denoted as $\{\!\{ \cdot \}\!\}$). The assumption in LTA problems is that for all constituents $c_i \in C$ a local score $y_i \in \mathcal{Y}$ is either given or computable. The set of those local scores should be indicative of the overall score $y \in \mathcal{Y}$ of the composition C. LTA problems typically require two subproblems to be solved:

1. **Aggregation:** A variadic aggregation function $\mathcal{A} : \mathcal{Y}^* \to \mathcal{Y}$ that estimates composite scores has to be learned, i.e. $y \approx \hat{y} = \mathcal{A}(y_1, \ldots, y_n)$. Typically the aggregation function $\mathcal{A}$ should be associative and commutative to fit with the multiset-structure of compositions.

2. **Disaggregation:** In case the constituent scores y_i are not given, they have to be derived from a constituent representation, e.g. a vector $x_i \in \mathcal{X}$. To learn this derivation function $f : \mathcal{X} \to \mathcal{Y}$, only the constituent vectors $\{\!\{x_i\}\!\}_{i=1}^n$ and the composite score y is given. Thus the constituent scores y_i need to be *disaggregated* from y in order to learn f.

Overall LTA can be understood as the joint problem of learning the aggregation function $\mathcal{A}$ and the local score derivation function f. Two main approaches to

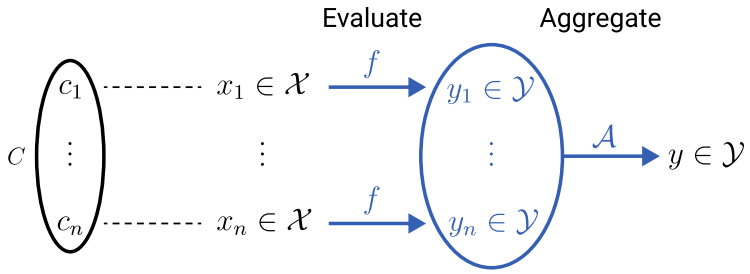


Figure 2.1. Overview of the structure of LTA for multiset compositions.

represent the aggregation function in LTA problems have been explored.

2.1.1 Uninorm-Aggregation

The first approach uses *uninorms* [MH16] to do so. There the basic idea is to express composite scores as fuzzy truth assignments $y \in [0,1]$. Such a composite assignment y is modeled as the result of a parameterized logical expression of constituent assignments $y_i \in [0,1]$. As the logical expression that thus effectively aggregates the constituents, a uninorm U_λ is used. Depending on the parameter $\lambda \in [0,1]$, U_λ combines a t-norm T and a t-conorm S which are continuous generalizations of logical conjunction and disjunction respectively. One popular choice of norms are the so-called Łukasiewicz norms:

$$\text{t-norm } T(a,b) \coloneqq \max\{0, a+b-1\}, \quad \text{t-conorm } S(a,b) \coloneqq \min\{a+b, 1\},$$

$$\text{uninorm } U_\lambda(a,b) \coloneqq \begin{cases} \lambda T\left(\frac{a}{\lambda}, \frac{b}{\lambda}\right) & \text{if } a,b \in [0,\lambda] \\ \lambda + (1-\lambda) S\left(\frac{a-\lambda}{1-\lambda}, \frac{b-\lambda}{1-\lambda}\right) & \text{if } a,b \in [\lambda,1] \\ \lambda \min\{a,b\} & \text{else} \end{cases} \tag{2.1}$$

At the extreme points $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$, T and S coincide with the Boolean operators $\wedge$ and $\vee$; the values at all other points are interpolated as shown in fig. 2.2. The uninorm U_λ uses the conjunctive t-norm T for values below the threshold λ and the disjunctive t-conorm S for values above the threshold. U_λ therefore smoothly interpolates between a conjunctive and disjunctive operator with the extreme points $U_1 = T$ and $U_0 = S$.

Since t-norms and t-conorms are commutative and associative they can also be applied to non-empty sets of arbitrary size, i.e. $T(\{\!\{y_1, \ldots, y_n\}\!\}) = T(y_1, T(\{\!\{y_2, \ldots, y_n\}\!\}))$ with fixpoint $T(\{\!\{y\}\!\}) = y$. Using this extension, a uninorm U_λ can be applied to sets which turns it into a parameterized aggregation function $\mathcal{A}_\lambda : [0,1]^* \to [0,1]$. In this simple model the LTA aggregation problem boils down to the optimization of λ. The LTA disaggregation problem is solved by jointly optimizing a logistic regression model, i.e. the constituent scores $\{\!\{y_i \in [0,1]\}\!\}_{c_i \in C}$ are described by $y_i = \left(1 + \exp(-\theta^\top x_i)\right)^{-1}$. Overall an LTA model is therefore described by the uninorm

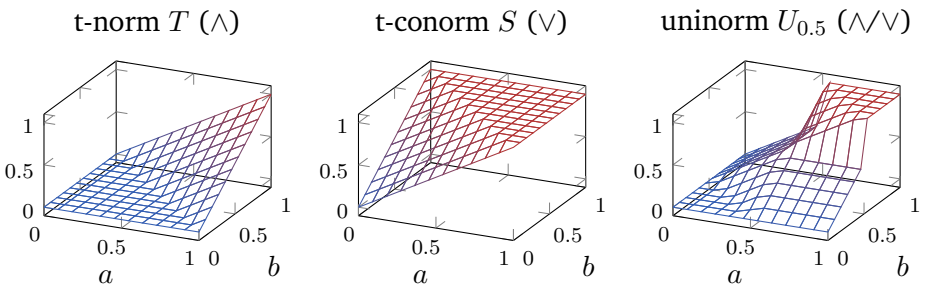


Figure 2.2. The Łukasiewicz norms and the corresponding uninorm for $\lambda = 0.5$.

parameter λ and the regression coefficients θ.

2.1.2 OWA-Aggregation

Recently Melnikov and Hüllermeier [MH19] have looked at an alternative class of aggregation functions. Instead of using fuzzy logic to describe score aggregation, *ordered weighted average* (OWA) operators were used. OWA aggregators work by sorting the input scores and then weighting them based on their sort position, i.e.

$$\mathcal{A}_\lambda(y_1, \ldots, y_n) \coloneqq \sum_{i=1}^{n} \lambda_i y_{\pi(i)}, \tag{2.2}$$

where $\lambda \in \mathbb{R}^n$ is a weight vector with $\|\lambda\|_1 = 1$ and $\pi : [n] \to [n]$ is a sorting permutation of the input scores with $[n] \coloneqq \{1, \ldots, n\}$ and $y_i < y_j \Rightarrow \pi(i) < \pi(j)$. Depending on the choice of the vector λ, the OWA function $\mathcal{A}_\lambda$ can express common aggregation functions like $\min$ (if $\lambda = (1, 0, \ldots, 0)$), $\max$ (if $\lambda = (0, \ldots, 0, 1)$) or the arithmetic mean (if $\lambda = \left(\frac{1}{n}, \ldots, \frac{1}{n}\right)$).

To deal with varying composition sizes n, the weights $\lambda_1, \ldots, \lambda_n$ can however not be statically assigned. Instead they are interpolated using a so-called *basic unit interval monotone* (BUM) function $q : [0, 1] \to [0, 1]$. It takes constituent positions that are normalized to the unit interval, i.e. $\frac{i}{n} \in [0, 1]$. The BUM function q is then used to interpolate a weight for any normalized sort position via $\lambda_i \coloneqq q\left(\frac{i}{n}\right) - q\left(\frac{i-1}{n}\right)$. Because q is monotone with $q(0) = 0$ and $q(1) = 1$, it always holds that $\|\lambda\|_1 = q(1) - q(0) = 1$. Using this model, the aggregation problem boils down to optimizing the shape of q.

In the OWA approach the BUM function q is modeled as a piecewise linear spline. This spline is described by $m + 1$ points $\left\{\left(\frac{j}{m}, a_j\right)\right\}_{j=0}^{m}$, the so-called knots of the spline. The curve of q is obtained by linearly interpolating between neighboring knots as shown in fig. 2.3. If $0 = a_0 \leq a_1 \leq \cdots \leq a_m = 1$, q is a BUM function. The LTA aggregation problem is therefore solved by optimizing $a \in \mathbb{R}^{m+1}$ under this constraint. The disaggregation problem is tackled by adding the scores $y_1, \ldots, y_M \in \mathbb{R}$ to the learnable parameters of the model where M is assumed to be the finite number of

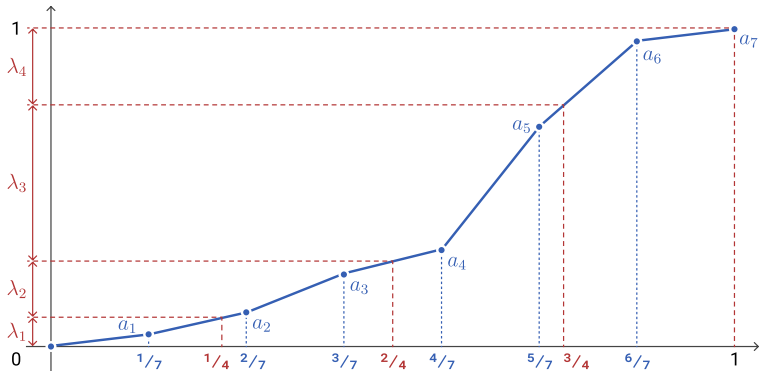


Figure 2.3. Illustration of how a describes q and its relation to λ ($n = 4$, $m = 7$).

constituents. Currently the OWA approach requires all possible constituents to be part of the training dataset since it does not consider constituent features $x_i \in \mathcal{X}$ to predict the scores of previously unseen constituents.

2.2 Graph Characterization

Definition 2.1. A *graph* $G := (\mathcal{V}_G, \mathcal{E}_G)$ consists of a finite set of vertices $v_i \in \mathcal{V}_G$ and a set of edges $e_{ij} = (v_i, v_j) \in \mathcal{E}_G \subseteq \mathcal{V}_G^2$. Optionally discrete vertex labels $l_G[v_i] \in L_\mathcal{V}$ or edge labels $l_G[e_{ij}] \in L_\mathcal{E}$ may be associated with all vertices $v_i \in \mathcal{V}_G$ and edges $e_{ij} \in \mathcal{E}_G$ respectively. Also continuous feature vectors $x_G[v_i] \in \mathcal{X}_\mathcal{V}$, $x_G[e_{ij}] \in \mathcal{X}_\mathcal{E}$ may be given. If $\mathcal{X}_\mathcal{E} = \mathbb{R}$, $x_G[e_{ij}]$ can be interpreted as an edge weight of e_{ij}.

In this thesis all graphs G are assumed to be undirected if not explicitly stated otherwise, i.e. $e_{ij} \in \mathcal{E}_G \leftrightarrow e_{ji} \in \mathcal{E}_G \wedge l_G[e_{ij}] = l_G[e_{ji}] \wedge x_G[e_{ij}] = x_G[e_{ji}]$. We denote the set of all undirected graphs as $\mathcal{G}$. Additionally we denote $G[S] := (S, \mathcal{E}_G \cap S^2)$ as the subgraph of G induced by $S \subseteq \mathcal{V}_G$.

To classify or score a graph, it first needs to be characterized by a set of relevant properties. The most strict characterization of a graph is its so-called *isomorphism class*. It uniquely identifies a graph but lacks any notion of similarity between non-identical graphs. We will begin with a brief definition of this strict isomorphism-based graph characterization. Then two less strict characterization approaches are described; the so called Weisfeiler-Lehman coloring and the notion of graph spectra. They are the theoretical foundation of many current GC/GR methods.

2.2.1 The Graph Isomorphism Problem

In order to process a graph G of size $n = |\mathcal{V}_G|$, one is forced to choose some encoding, e.g. an adjacency matrix $A_G \in \{0,1\}^{n \times n}$. Such an encoding introduces a vertex ordering $v_1, \ldots, v_n$ that does not carry any semantic meaning. Consequently there are $n!$ equivalent encodings of G. To represent those encodings we introduce the

notion of ordered induced subgraphs.

Definition 2.2. For all vertex k-tuples $v = (v_1, \ldots, v_k) \in \mathcal{V}_G^k$ let $\hat{v} = \text{set}(v) \coloneqq \{v_i \mid i \in [k]\}$. Then $G[v] \coloneqq (\hat{v}, \mathcal{E}_G \cap \hat{v}^2, v)$ is called the *ordered subgraph* of G induced by v. We denote the set of all ordered subgraphs with $\mathcal{G}_{ord}$. If $\hat{v} = \mathcal{V}_G$ and $k = |\mathcal{V}_G|$, we call v an *ordering* of G and $G[v]$ an *encoding* of G.

Definition 2.3. Two ordered induced subgraphs $G[v]$ and $H[w]$ with $(v_1, \ldots, v_k) \in \mathcal{V}_G^k$ and $(w_1, \ldots, w_k) \in \mathcal{V}_H^k$ are *equivalent* ($G[v] \equiv H[w]$) iff.

$$
\begin{aligned}
&\forall i, j \in [k] : (v_i, v_j) \in \mathcal{E}_G \leftrightarrow (w_i, w_j) \in \mathcal{E}_H \\
\wedge\ &\forall i, j \in [k] : l_G[v_i, v_j] = l_H[w_i, w_j] \quad \wedge \quad \forall i \in [k] : l_G[v_i] = l_H[w_i] \\
\wedge\ &\forall i, j \in [k] : x_G[v_i, v_j] = x_H[w_i, w_j] \quad \wedge \quad \forall i \in [k] : x_G[v_i] = x_H[w_i].
\end{aligned}
$$

Using this notion of equivalence, we call $[G[v]] \coloneqq \{G[w] \mid G[w] \equiv G[v] \wedge w \in \mathcal{V}_G^*\}$ the *ordered subgraph equivalence class* of $G[v]$.

Definition 2.4. Two graphs G and H are *isomorphic* ($G \simeq H$) iff. there are equivalent encodings $G[v] \equiv H[w]$ of them. Consequently $[G] \coloneqq \{H \mid H \simeq G\}$ is called the *isomorphism class* of G.

The goal of the *graph isomorphism* (GI) problem is to check whether $G \simeq H$ for two arbitrary graphs. Even though there is no known universal polynomial algorithm that solves GI, Babai et al. [Bab+80] showed that almost all graphs can be trivially distinguished in linear time. More recently Babai [Bab15] also presented a quasipolynomial upper time bound for the remaining hard GI instances. For all practical purposes, GI can therefore be solved efficiently; e.g. via the `nauty` program [MP13][🔗NT]. One important subroutine in most GI checkers is the Weisfeiler-Lehman algorithm which will be described next.

2.2.2 Weisfeiler-Lehman Graph Colorings

The *Weisfeiler-Lehman* (WL) algorithm [WL68][Cai+92] characterizes a graph G by assigning discrete labels $c \in \mathcal{C}$, called *colors*, to vertex k-tuples $(v_1, \ldots, v_k) \in \mathcal{V}_G^k$, where $k \in \mathbb{N}_0$ is the freely choosable *WL-dimension*. A mapping $\chi_{G,k} : \mathcal{V}_G^k \to \mathcal{C}$ is called a *k-coloring* of G.

Definition 2.5. A coloring χ' *refines* χ ($\chi' \preceq \chi$) iff. $\forall a, b \in \mathcal{V}_G^k : \chi(a) \neq \chi(b) \rightarrow \chi'(a) \neq \chi'(b)$, i.e. χ' distinguishes at least those tuples that are distinguished by χ.

Definition 2.6. Two colorings χ and χ' are *equivalent* ($\chi \equiv \chi'$) iff. $\chi \preceq \chi' \wedge \chi \succeq \chi'$, i.e. χ is identical to χ' up to color substitutions.

The k-dimensional WL algorithm (k-WL) works by iteratively refining k-colorings

$\chi^{(0)}_{G,k} \succeq \chi^{(1)}_{G,k} \succeq \dots$ of a given graph G until the convergence criterion $\chi^{(i)}_{G,k} \equiv \chi^{(i+1)}_{G,k}$ is satisfied. We denote the final, maximally refined k-WL coloring with $\chi^{*}_{G,k}$.

Definition 2.7. The color distribution $dist_{\chi_{G,k}} : \mathcal{C} \to \mathbb{N}_0$ of a k-coloring $\chi_{G,k}$ counts each color $c \in \mathcal{C}$ in the coloring, i.e. $dist_{\chi_{G,k}}(c) \coloneqq \left|\left\{v \in \mathcal{V}^k_G \,|\, \chi_{G,k}(v) = c\right\}\right|$.

Definition 2.8. Two graphs G and H are k-WL *distinguishable* ($G \not\simeq_k H$) iff. there exists a color $c \in \mathcal{C}$ s.t. $dist_{\chi^*_{G,k}}(c) \neq dist_{\chi^*_{H,k}}(c)$.

As we will see, the way in which WL colorings are refined is vertex order invariant; thus any difference in the final coloring of two graphs always implies the non-isomorphism of the colored graphs, i.e. $G \not\simeq_k H \implies G \not\simeq H$. The opposite does however not necessarily hold; two k-WL indistinguishable graphs are not always isomorphic, i.e. $\exists\, G, H : G \simeq_k H \wedge G \not\simeq H$.

In addition to the binary aspect of WL distinguishability and its relation to the GI problem, WL colorings are also useful for more fuzzy graph similarity comparisons as we will see in section 2.3.2 when we look at graph kernels. Before that however, the details of WL color refinement strategy have to be described. We begin with the color refinement algorithm for the most simple case of $k = 1$. Then the definitions and intuitions from the 1-dimensional case are extended to its higher-dimensional generalization. Lastly we will discuss the discriminative power of the WL algorithm and its relation to the WL-dimension k.

The 1-dimensional WL algorithm

In the 1-dimensional WL algorithm (1-WL), a color is assigned to each vertex of a graph. If the vertices $v \in \mathcal{V}_G$ of the input graph G are labeled, those labels $l_G[v] \in L_{\mathcal{V}} \subseteq \mathcal{C}$ can be used as the initial graph coloring $\chi^{(0)}_{G,1}(v) \coloneqq l_G[v]$. Since WL colors are inherently discrete, continuous vertex feature vectors $x_G[v]$ are not considered here. For unlabeled graphs a constant coloring is used, e.g. $\forall v \in \mathcal{V}_G : \chi^{(0)}_{G,1}(v) = \mathbf{A}$ for some initial color $\mathbf{A} \in \mathcal{C}$. In each iteration of the 1-WL color refinement algorithm, the following neighborhood aggregation scheme is used to compute a new color for all vertices:

Definition 2.9. $\chi^{(i+1)}_{G,1}(v) \coloneqq h\left(\chi^{(i)}_{G,1}(v), \{\!\{\chi^{(i)}_{G,1}(u) \,|\, u \in \Gamma_G(v)\}\!\}\right)$, with $\Gamma_G(v)$ denoting the set of adjacent vertices of $v \in \mathcal{V}_G$ and $h : \mathcal{C}^* \to \mathcal{C}$ denoting an injective hash function that assigns a unique color to each finite combination of colors.

In practice the hash function h is usually defined lazily by using $\mathcal{C} \subseteq \mathbb{N}$ and enumerating color combinations in whichever order they are hashed s.t. a new color is introduced every time a previously unseen color combination appears at runtime[1].

[1] $\mathcal{C}$ is countably infinite but we assume it only contains the finite no. of colors occurring in a dataset.

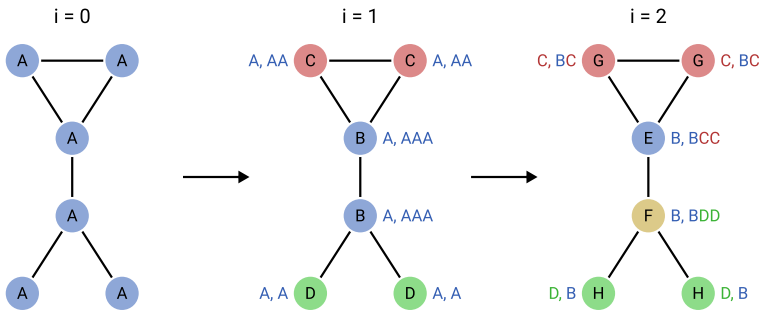


Figure 2.4. Example 1-WL color refinement steps. After two iterations the coloring stabilizes. Each vertex v is labeled with its current color and has its previous color and the colors of the hashed neighbors $\Gamma_G(v)$ written next to it (see definition 2.9).

The k-dimensional WL algorithm

As we just saw, the 1-WL algorithm iteratively refines colorings of single vertices. While the obtained colorings differ for most non-isomorphic graphs $G \not\simeq H$, 1-WL does not generally solve the GI problem as illustrated in fig. 2.5.

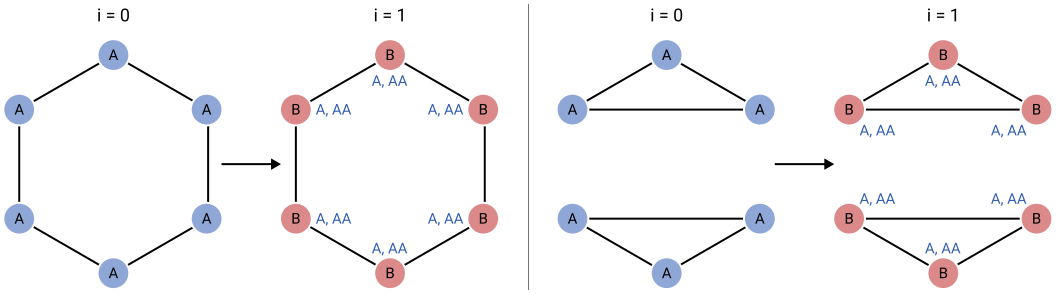


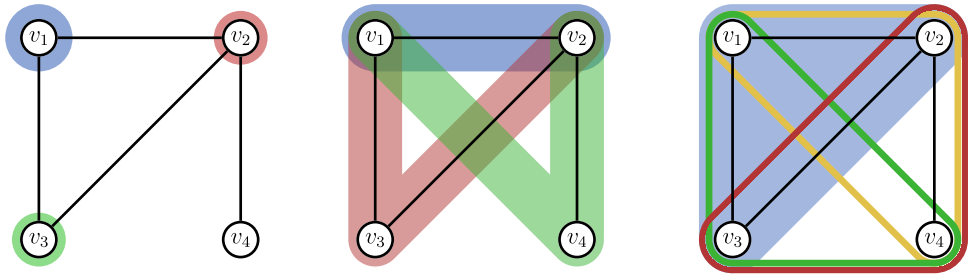
Figure 2.5. Two simple non-isomorphic graphs that are indistinguishable by 1-WL.

By extending WL to higher dimensions, such 1-WL indistinguishable cases can however be handled. Analogous to the 1-dimensional definition 2.9, the k-dimensional color refinement step is defined by

Definition 2.10. $\chi_{G,k}^{(i+1)}(s) := h\left(\chi_{G,k}^{(i)}(s), \{\!\{(\chi_{G,k}^{(i)}(s[u/1]), \ldots, \chi_{G,k}^{(i)}(s[u/k])) \mid u \in \mathcal{V}_G\}\!\}\right)$

with $s = (v_1, \ldots, v_k) \in \mathcal{V}_G^k, \quad s[u/j] := (v_1, \ldots, v_{j-1}, u, v_{j+1}, \ldots, v_k).$

In 1-WL a vertex color is refined by combining the colors of neighboring vertices. In k-WL the color of a k-tuple $s \in \mathcal{V}_G^k$ is refined by combining the colors of its neighborhood which is defined as the set of all k-tuples in which at most one vertex differs from s. Note that each vertex k-tuple has one neighbor for each $u \in \mathcal{V}_G$, each of which is a k-tuple of vertex k-tuples. This more abstract notion of neighborhood is illustrated in fig. 2.6. For $k = 2$ this means that each potential edge $(v, w) \in \mathcal{V}_G^2$ has all possible walks of length 2 from v to w as its neighbors (see fig. 2.6b). Also note that, even though k-WL refines k-tuple colors, lower-dimensional structures still get their own colors since a tuple does not have to consist of distinct vertices, i.e. in k-WL the color of a single vertex $v \in \mathcal{V}_G$ is described by $\chi_{G,k}^{*}(s)$ for $s = (v, \ldots, v) \in \mathcal{V}_G^k$.



(a) v_1's 1-WL neighbors **(b)** (v_1, v_2)'s 2-WL neighbors **(c)** (v_1, v_2, v_3)'s 3-WL neighbors

Figure 2.6. Tuple neighborhoods for different values of k. The vertices highlighted in blue form the root tuple $s \in \mathcal{V}_G^k$ whose neighbors are shown; for simplicity neighbors with $u \in s$ are left out (see definition 2.10). Each neighbor is highlighted with a different color, except for 3-WL where the red, green and yellow triples actually form the single neighbor for $u = v_4$.

Let us now look at how the tuple colors are initialized. For this we use the ordered subgraph equivalence classes $[G[s]]$ (see definition 2.3) which determine the initial color $\chi_{G,k}^{(0)}(v)$ of each k-tuple s. For $k = 1$ the equivalence class of a single vertex v directly corresponds to its label $l_G[v]$. More generally for $k > 1$ this means that

$$\chi_{G,k}^{(0)}(s) = \chi_{G,k}^{(0)}(t) \iff G[s] \equiv G[t]. \tag{2.3}$$

Note that there is a fundamental difference in how the adjacency information encoded in $\mathcal{E}_G$ is used in 1-WL vs. k-WL: In 1-WL a vertex coloring by itself cannot encode adjacency which is why this information is explicitly incorporated in each refinement step via Γ_G (see definition 2.9). In k-WL on the other hand, each pair of vertices $(v, u) \in \mathcal{V}_G^2$ appears in at least one k-tuple (assuming $k \geq 2$) and therefore has at least one color which can implicitly encode the adjacency information. Edges and non-edges are colored differently in the initial coloring since $G[(v, u)] \not\equiv G[(w, u)]$ if $(v, u) \in \mathcal{E}_G$ but $(w, u) \notin \mathcal{E}_G$; thus no explicit adjacency information is needed in the k-WL color refinement step in definition 2.10.

Discriminative Power of WL

Now we will look at the types of graphs that can be distinguished by WL in relation to the WL-dimension k. We begin by showing that the power of WL grows with k.

Lemma 2.11. $G \not\simeq_k H \implies G \not\simeq_{k+1} H$ *($(k+1)$-WL is at least as powerful as k-WL).*

Proof. All k-tuples can be mapped to $(k+1)$-tuples via $\varphi(v_1, \ldots, v_k) := (v_1, \ldots, v_k, v_k)$. For each neighbor $(s[u/1], \ldots, s[u/k])$ of $s \in \mathcal{V}_G^k$, there is a corresponding neighbor $(\varphi(s[u/1]), \ldots \varphi(s[u/k]), \varphi(s[u/k]))$ of $\varphi(s)$. Using eq. (2.3) (for $i = 0$) and definition 2.10 (for $i > 0$) it follows that $\forall s, t \in \mathcal{V}_G^k : \chi_{G,k}^{(i)}(s) \neq \chi_{G,k}^{(i)}(t) \to \chi_{G,k+1}^{(i)}(\varphi(s)) \neq \chi_{G,k+1}^{(i)}(\varphi(t))$. The lemma then follows by definition 2.8. □

Proposition 2.12 (see Immerman and Lander [IL90] for the proof)**.** *For all $k \in \mathbb{N}$ there are non-isomorphic graphs $G \not\simeq H$ of size $\mathcal{O}(k)$ with $G \simeq_k H$.*

Corollary 2.13. *The discriminative power of k-WL grows monotonously with k and never converges, i.e. for all $k \in \mathbb{N}$ there is an $l \in \mathbb{N}$ s.t. the set of k-WL distinguishable graphs is a proper subset of the $(k+l)$-WL distinguishable graphs.*

Proof. Note that k-WL trivially solves the GI problem for all graphs of size $\leq k$ via the initial coloring (see eq. (2.3)). Using lemma 2.11 and proposition 2.12 the corollary directly follows. □

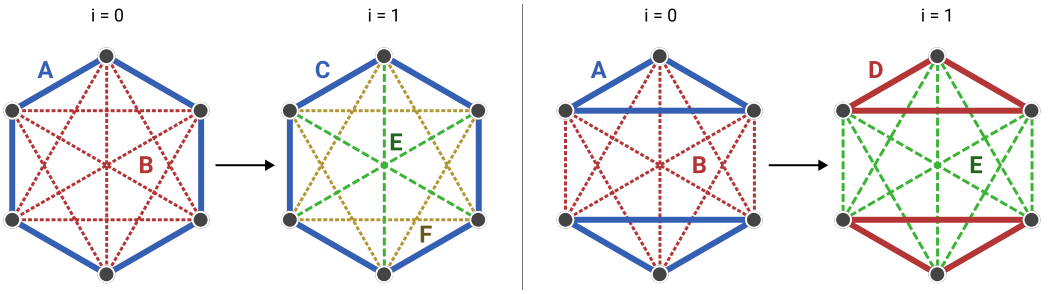


Figure 2.7. Two non-isomorphic graphs with $G \simeq_1 H$ and $G \not\simeq_2 H$.

Figure 2.7 illustrates corollary 2.13 by showing how 2-WL is able to distinguish the two 1-WL indistinguishable graphs from fig. 2.5. Since the time complexity of k-WL grows exponentially with k [IL90, cor. 1.9.7], it does not provide an efficient universal solution to GI. However it turns out that almost all graphs are WL distinguishable even for a small constant k.

For $k = 1$ Babai et al. [Bab+80] have shown that two randomly selected non-isomorphic graphs $G \not\simeq H$ of size n are 1-WL indistinguishable with probability $n^{-1/7}$. Thus 1-WL is already able to distinguish most graphs; it fails however to distinguish any pair of unlabeled d-regular graphs of the same size [IL90, cor. 1.8.5].

Definition 2.14. A graph G is called *d-regular* ($rg_d(G)$) iff. $\forall v \in \mathcal{V}_G : |\Gamma_G(v)| = d$.

We have already seen this in fig. 2.5 since the "six-cycle" and the "two-triangles" graphs are in fact both 2-regular and of size 6. This restriction alone is typically not an issue in the context of GC/GR though, as graphs in many real-world domains are rarely perfectly regular [Abe18]. A more relevant restriction of 1-WL is the fact that it is unable to detect cycles of length $m \geq 3$.

Definition 2.15. k-WL *computes* a function $f : \mathcal{G} \to Y$ iff. that function can be expressed as $f(G) = g(dist_{\chi^*_{G,k}})$ via some function $g : (\mathcal{C} \to \mathbb{N}) \to Y$.

Definition 2.16. k-WL *counts* a certain ordered subgraph $S \in \mathcal{G}_{ord}$ iff. it computes $count_S(G) \coloneqq |\{\text{set}(v) \,|\, v \in \mathcal{V}^*_G \wedge G[v] \equiv S\}|$ (see definition 2.2). Similarly k-WL *detects* an ordered subgraph S iff. it computes $contain_S(G) \coloneqq \mathbb{1}[count_S(G) > 0]$, with $\mathbb{1}$ denoting the indicator function.

To see why 1-WL is unable to detect m-cycles in graphs, note that fig. 2.5 already contradicts the positive statement for $m = 3$; this counterexample can be trivially generalized to all larger m by replacing the "six-cycle" graph with a "$(2m)$-cycle-graph" graph and the "two-triangles" graph with a "two-m-cycles" graph which preserves the 2-regularity and therefore the 1-WL indistinguishability.

This cycle-detection restriction of 1-WL is relevant in practice because cycle counts are used in many domains to analyze graphs, e.g. triangle counts are commonly used in social network analysis to find interaction clusters [Mil02][New03][Wel+07] and the detection of 4-, 5- or 6-cycles is required to determine important chemical properties like the aromaticity of a molecule [AB73][Kek66].

If we increase the WL-dimension to $k = 2$, the described 1-WL restrictions no longer apply. 2-WL is able to distinguish more than $1 - \mathcal{O}(1/n)$ of the regular n-vertex graphs [IL90, cor. 1.8.6] and can even count cycles:

Proposition 2.17 (see Fürer [Fü17] and Arvind et al. [Arv+19] for the full proof)**.** *2-WL is able to count m-cycles for all $m \leq 7$ but it cannot even detect them for $m > 7$.*

Proof Sketch. Only the idea behind triangle and 4-cycle counting is outlined here to give an intuition for why proposition 2.17 holds. Note that the 2-WL neighborhood of each edge $(v, w) \in \mathcal{E}_G$ includes all possible paths (v, u, w) of length 2, i.e. all possible triangles. By definition 2.10 there must thus be a color subset $C_j^{\triangle} \subseteq \mathcal{C}$ representing that an edge is involved in exactly j triangles after one refinement step. 2-WL then trivially counts triangles by setting $g(dist_\chi) = \frac{1}{3} \sum_j j \sum_{c \in C_j^{\triangle}} dist_\chi(c)$ to satisfy definition 2.15. Analogously for 4-cycles, let $C_j^{\square} \subseteq \mathcal{C}$ be the colors indicating a non-edge $(v, w) \notin \mathcal{E}_G$ with j common vertex neighbors $u \in \Gamma_G(v) \cap \Gamma_G(w)$. The number of 4-cycles is then determined by the colors of the diagonals through them via $g(dist_\chi) = \frac{1}{2} \sum_{j \geq 2} \binom{j}{2} \sum_{c \in C_j^{\square}} dist_\chi(c)$. Using a similar but more involved combinatorial argument requiring multiple color refinement steps, 5-, 6- & 7-cycle counting can be shown. □

As we just saw, 2-WL is significantly more powerful than 1-WL. Though, by proposition 2.12, there are of course still 2-WL indistinguishable graphs; among others those are the strongly regular graphs $srg_{n,d,\lambda,\mu}(G)$.

Definition 2.18. $srg_{n,d,\lambda,\mu}(G) \iff |\mathcal{V}_G| = n \quad \wedge \quad \forall v \in \mathcal{V}_G : |\Gamma_G(v)| = d$
$\wedge\, \forall (v, w) \in \mathcal{E}_G : |\Gamma_G(v) \cap \Gamma_G(w)| = \lambda$
$\wedge\, \forall (v, w) \in \mathcal{V}_G^2 \setminus \mathcal{E}_G : |\Gamma_G(v) \cap \Gamma_G(w)| = \mu$

Generally this restriction of 2-WL is not an issue since strongly regular graphs do not appear in typical GC/GR datasets. By going to $k = 3$, even some strongly regular graphs as well as all planar graphs can be distinguished though [Kie+17]. Apart from that there are currently few results regarding the classes of distinguishable graphs and computable functions for even higher WL-dimensions.

2.2.3 Spectral Graph Theory

Let us now look at an alternative perspective on graph characterization which is provided by *spectral graph theory*. While WL characterizes a graph via its color distribution $dist_{\chi^*_{G,k}}$, the spectral approach uses its so-called *spectrum* $\lambda_G = (\lambda_{G,1}, \ldots, \lambda_{G,n}) \in \mathbb{R}^n$ where $n = |\mathcal{V}_G|$. The key idea behind this is to interpret the adjacency matrix $A_G \in \mathbb{R}^{n \times n}$ of G not simply as an encoding of the edges ($A_{G,i,j} = \mathbb{1}[(v_i, v_j) \in \mathcal{E}_G]$) but as a linear operator $A_G : (\mathcal{V}_G \to \mathbb{R}) \to (\mathcal{V}_G \to \mathbb{R})$ acting on the vector space of real-valued functions with a vertex domain. This means that A_G transforms so-called *graph signals* $x : \mathcal{V}_G \to \mathbb{R}$ which are functions assigning *signal strengths* to vertices. By applying $A_G x$ the signal strength $x[v_i]$ of each vertex is added to the signal strengths of its neighbors $\Gamma_G(v_i)$. Using this functional perspective, one can characterize graphs via the *Fourier transform* (FT). We will now see how this is done; however, since a comprehensive description of spectral graph theory would exceed the scope of this thesis, only a brief overview will be given. For a more detailed introduction to the field we refer to Shuman et al. [Shu+13].

The classical Fourier transform Let us begin by describing the FT for the more common case of functions with real domains. All functions $f : \mathbb{R} \to \mathbb{R}$ can be interpreted as infinite-dimensional vectors $f = \int_{\mathbb{R}} f(t)\, b_t \,\mathrm{d}t$ with $b_t : \mathbb{R} \to \mathbb{R}$ being a standard basis vector/function defined as $b_t(s) \coloneqq \begin{cases} 1 & \text{if } s = t \\ 0 & \text{else} \end{cases}$. The values of f are then described as its b_t components $\langle b_t, f \rangle = f(t)$, where $\langle b_t, \cdot \rangle = \delta_t(\cdot)$ denotes the Dirac delta function translated by t. The Fourier transform $\hat{f} = \mathcal{F}(f)$ of f corresponds to a change of basis from the standard basis vectors b_t to the Fourier basis vectors $u_\xi(t) \coloneqq e^{2\pi i \xi t}$, i.e. $f = \int_{\mathbb{R}} \hat{f}(t)\, u_\xi \,\mathrm{d}\xi$. The Fourier basis is characterized by the fact that it is an eigenbasis of the so-called *Laplace operator* Δ, i.e. $\Delta u_\xi = \lambda_\xi u_\xi$ with λ_ξ being the eigenvalue corresponding to u_ξ. In the real domain this Laplace operator Δ is effectively the same as the second-derivative $\frac{\mathrm{d}^2}{\mathrm{d}t^2}$. Thus it is easy to see that indeed $\Delta u_\xi = -\frac{\mathrm{d}^2}{\mathrm{d}t^2} u_\xi = \lambda_\xi u_\xi$ for $\lambda_\xi = (2\pi\xi)^2$.

The graph Fourier transform To extend the notion of the FT from real functions $f : \mathbb{R} \to \mathbb{R}$ to graph signals $x : \mathcal{V}_G \to \mathbb{R}$ we need to define a graph variant of the Laplace operator Δ. It turns out that there are multiple possible ways to do so, the most simple being the so-called *combinatorial graph Laplacian*.

Definition 2.19. The combinatorial graph Laplacian $L_G \in \mathbb{R}^{n \times n}$ is defined as

$$L_G \coloneqq D_G - A_G \quad \text{with the degree matrix } D_G \coloneqq \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix}, d_i \coloneqq |\Gamma_G(v_i)|.$$

Using this definition, applying $L_G x$ is analogous to taking the second derivative $\frac{\mathrm{d}^2}{\mathrm{d}t^2} f$.

Putting both Laplacian variants, L_G and $\frac{\mathrm{d}^2}{\mathrm{d}t^2}$, side-by-side gives an intuition for why this is the case:

$$-\frac{\mathrm{d}^2}{\mathrm{d}t^2} f(t) = \lim_{h \to 0} \frac{1}{h^2} (\underbrace{f(t) - f(t-h)}_{\Delta_{t,t-h}} + \underbrace{f(t) - f(t+h)}_{\Delta_{t,t+h}}) \quad \Big| \quad L_G x[v] = \sum_{u \in \Gamma_G(v)} \underbrace{(x[v] - x[u])}_{\Delta_{v,u}}$$

The second derivative of a function f essentially averages the value differences in the neighborhood of a point t. For real-valued functions this neighborhood only consists of the two infinitesimally close points to the left and to the right of t, i.e. $t-h$ and $t+h$. The combinatorial graph Laplacian represents the same operation, where each point/vertex v might however have more than two neighbors $u \in \Gamma_G(v)$ that need to be averaged.

The FT of a graph signal x then is $\hat{x}[i] = \langle u_{G,i}, x \rangle$ with $u_G = \{u_{G,i}\}_{i=1}^n$ being the eigenvectors of L_G, s.t. $x = \sum_{i=1}^n \hat{x}[i] u_{G,i}$. Since the set of Laplacian eigenvectors is finite, we can express the graph FT $\mathcal{F}_G$ as a change of basis matrix $U_G = \begin{pmatrix} u_{G,1,1} & \cdots & u_{G,1,n} \\ \vdots & \ddots & \vdots \\ u_{G,n,1} & \cdots & u_{G,n,n} \end{pmatrix} \in \mathbb{R}^{n \times n}$. Similarly the inverse FT can be expressed as $\mathcal{F}^{-1} = U_G^{-1} = U_G^\top$ because U_G is a real orthogonal matrix. To see why this is the case, note that L_G is guaranteed to only have real eigenvectors u_G and eigenvalues λ_G due to the fact that we only consider undirected graphs with symmetric adjacency matrices A_G.

Interpretation of the spectrum By convention we assume that the eigenvalues are ordered ascendingly: $\lambda_{G,1} \leq \cdots \leq \lambda_{G,n}$. Those eigenvalues are called the *spectrum of* G and they are a vertex-permutation-invariant graph characterization. Intuitively those eigenvalues describe the connectivity between different parts of the graph. The first eigenvalues represent connectivity at a general, coarse level while the last eigenvalues represent the connectivity of finer substructures. Figure 2.8 illustrates this idea. A concrete example for the relation between a graph's structure and its spectrum is the fact that a graph with m connected components has exactly m zero eigenvalues, i.e. $0 = \lambda_{G,1} = \cdots = \lambda_{G,m} < \lambda_{G,m+1}$. For a more detailed discussion of this relation we refer to Das [Das04]. We will now instead look at the discriminative power of the spectrum.

Definition 2.20. Two graphs G and H are *cospectral* ($G \simeq_\lambda H$) iff. $\forall i : \lambda_{G,i} = \lambda_{H,i}$.

Spectral graph comparisons Alzaga et al. [Alz+10] have shown that, while the cospectrality test can be more powerful than 1-WL, it is always weaker than 2-WL, i.e. $G \simeq_2 H \implies G \simeq_\lambda H$. Despite this limit on the discriminative power of the graph spectrum, it is still useful for determining graph similarity, e.g. by defining a distance measure on the vector space of spectra (see Gu et al. [Gu+15]).

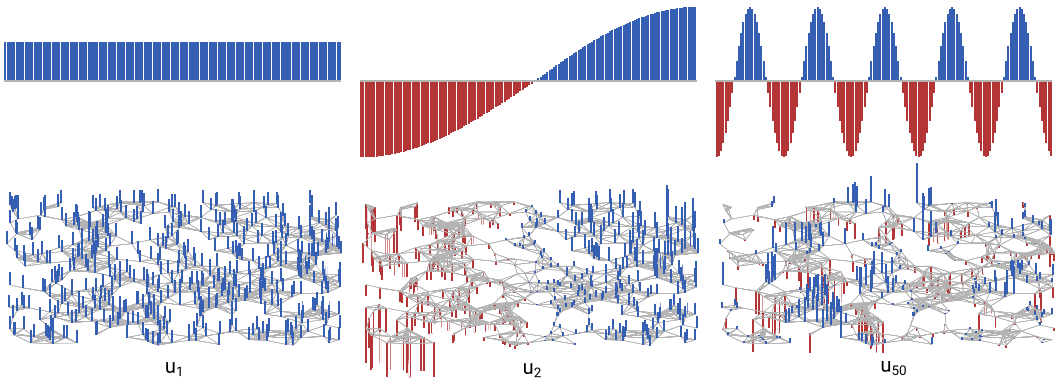


Figure 2.8. Comparison between the basis functions/vectors of the real domain FT and the graph FT. For the eigenfunctions on the upper half only the real cosine components of the complex exponentials are shown. BASED ON: [SHU+13]

As briefly mentioned before, the combinatorial graph Laplacian L_G is not the only possible choice of graph Laplacian. Note that the eigenvalues of L_G grow with the number of edges[2]. Consequently a small graph G will typically have a large spectral distance to a large graph H (assuming $|\mathcal{E}_G| \ll |\mathcal{E}_H|$) irrespective of their similarity when ignoring the scale difference. In domains where the absolute size of a graph should not influence its spectral characterization, it can therefore be useful to normalize the spectrum. One common way to do so is via the so-called *symmetric normalized Laplacian* L_G^{sym}.

Definition 2.21. $L_G^{\text{sym}} := D^{-\frac{1}{2}} L_G D^{-\frac{1}{2}}$ is the *symmetric normalized Laplacian* of G.

The eigenvalues $\lambda_{G,i}^{\text{sym}}$ of this Laplacian all lie in the range $[0, 2]$. Figure 2.9 illustrates how this makes it possible to compare the structure of graphs across varying vertex counts. Besides L_G^{sym} there are also other ways to normalize the spectrum, e.g. by using the random walk Laplacian $L_G^{\text{rw}} := D^{-1} L_G$, which will however not be covered here (see Shuman et al. [Shu+13]).

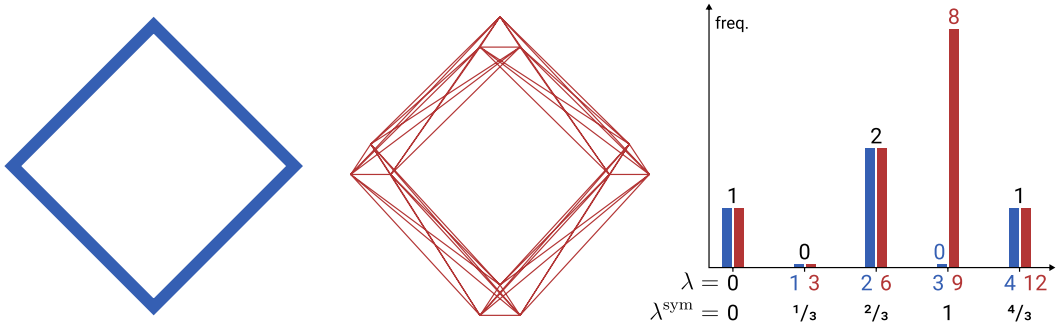


Figure 2.9. Comparison of the unnormalized L_G spectrum and the normalized L_G^{sym} spectrum. Two structurally similar graphs of different size are compared. The large graph is derived by replacing each vertex of the small 4-cycle graph by a triangle. While their unnormalized spectra have different absolute ranges ($[0, 4]$ vs. $[0, 12]$), the histogram on the right shows that their normalized eigenvalue distributions align.

[2] This is trivially shown by $\sum_{i=1}^{n} \lambda_{G,i} = \text{Tr}(L_G) = \text{Tr}(D_G) = |\mathcal{E}_G|$.

2.3 Graph Classification and Regression

Now that we have looked at the WL- and spectrum-based graph characterization approaches, we will see how those can be used to learn models that solve the *graph classification and regression* (GC/GR) problem. The existing approaches to tackle the GC/GR problem can be categorized into three main families: 1. Explicit graph embeddings, 2. graph kernels and 3. graph neural networks. In the following sections the characteristics of those families are described and a brief overview of specific methods is given.

2.3.1 Explicit Graph Embeddings

The basic idea of explicit graph embedding approaches is to map a graph $G \in \mathcal{G}$ to some vector in a finite vector space $\mathcal{X} = \mathbb{R}^d$. A function $\varphi : \mathcal{G} \to \mathcal{X}$ is called a *graph embedding function*. By embedding a graph into $\mathcal{X}$, any classification or regression algorithm that works with vectors can then be applied to solve the GC/GR problem.

The so-called vertex embedding problem is closely related to the graph embedding problem. As the name suggests, a *vertex embedding function* φ_G maps all vertices $v \in \mathcal{V}_G$ to $\mathcal{X}$. The embedding vector $\varphi_G(v)$ ideally encodes relevant information about a vertex and its structural position in G. It can be used to solve the vertex classification and regression problem via arbitrary ML methods for vectors. We will now look at two main families of explicit graph and vertex embedding approaches.

Fingerprint Embeddings

The first works on graph embeddings were motivated by the study of chemical structures [AB73][WW86]. There a molecule can be interpreted as a labeled graph for which the GC/GR problem corresponds to the prediction of some chemical property, e.g. toxicity or solubility. So-called *fingerprint embeddings* try to match a fixed set of subgraphs $S_1, \ldots, S_d$ to the input graph. The embedding of a graph G is a binary vector $\varphi_{\mathrm{FP}}(G) \in \{0,1\}^d$ with $\varphi_{\mathrm{FP}}(G)[i] \coloneqq contain_{S_i}(G)$ (see definition 2.16). Alternatively a fingerprint embedding can additionally encode multiplicities via $\varphi_{\mathrm{FP}}(G)[i] \coloneqq count_{S_i}(G) \in \mathbb{N}_0$.

This simple approach usually requires a careful choice of subgraphs but can still be competitive with the other more recent approaches we will look at in the following sections. Fingerprint embeddings are for example used in multiple state-of-the-art toxicity prediction tools like RASAR [Lue+18][🔗TT], the Toxicity Estimation Software Tools [🔗TET] or ProTox [Drw+14][Ban+18][🔗PT].

Skip-gram inspired Embeddings

Skip-gram embeddings were introduced by Mikolov et al. as part of the well-known `word2vec` [Mik+13] word embedding method from natural language processing. While a fingerprint embedding explicitly assigns an interpretation to each embedding dimension (i.e. to each standard basis vector), a skip-gram embedding only optimizes the distance between embedding vectors based on the similarity of the embedded instances without providing an interpretation of the embedding dimensions.

word2vec Let us first look at the `word2vec` skip-gram method. It gets a sequence of words $(w_0, \ldots, w_n)$ as input and outputs embedding vectors $\varphi(w_0), \ldots, \varphi(w_n) \in \mathbb{R}^d$. To do this the context $\Gamma_k(w_i) = \{w_{i-k}, \ldots, w_{i+k}\}$ is computed for all words where w_i is the so-called *context root*. The word contexts are then used to optimize the following log-likelihood objective:

$$\max_{\varphi,\varphi_\Gamma} \sum_{i=1}^{n} \log P(\Gamma_k(w_i)|w_i) \;=\; \max_{\varphi,\varphi_\Gamma} \sum_{i=1}^{n} \sum_{w_j \in \Gamma_k(w_i)} \left[\varphi(w_i)^\top \varphi_\Gamma(w_j) - \log Z_{w_i}\right] \tag{2.4}$$

$$\text{with } P(\Gamma_k(w_i)|w_i) := \prod_{w_j \in \Gamma_k(w_i)} \overbrace{\frac{\exp\left(\varphi(w_i)^\top \varphi_\Gamma(w_j)\right)}{Z_{w_i}}}^{P(w_j|w_i)} \text{ and } Z_{w_i} := \sum_{j=1}^{n} \exp\left(\varphi(w_i)^\top \varphi_\Gamma(w_j)\right)$$

`word2vec` essentially uses an expectation maximization scheme to maximize the probabilities $P(w_j|w_i)$ of observing the context words $w_j \in \Gamma_k(w_i)$ of all words w_i. Those probabilities are described by the overlap of the embeddings $\varphi(w_i)$ of words w_i and the embeddings $\varphi_\Gamma(w_j)$ of their context words w_j. Intuitively this means that words with similar contexts will be mapped close to each other in the embedding space. Note that `word2vec` actually finds two embeddings $\varphi(w)$ and $\varphi_\Gamma(w)$ for each word of which only the first is returned. The two embeddings represent two different perspectives on words: φ describes a word w_i as the root of a context $\Gamma_k(w_i)$, φ_Γ on the other hand describes a word w_j as part of a context $\Gamma_k(w_i) \ni w_j$.

Vertex Embeddings Skip-gram embeddings can be naïvely extended to graphs by realizing that `word2vec` effectively already is a vertex embedding method for linear graphs in which $\Gamma_k(v)$ is simply the k-neighborhood of the vertex/word v. The problem with this naïve extension is that the sizes of k-neighborhoods in arbitrary graphs can be much larger and often tend to grow exponentially with k. To deal with this computational problem the so-called DeepWalk [Per+14] and `node2vec` [GL16] methods perform random walks of fixed length to effectively take samples from the neighborhood of vertices. Both methods only differ in the transition matrix that is used for the random walk. Another difference of DeepWalk and `node2vec` compared to `word2vec` is the so-called *feature space symmetry* which states that the context root interpretation (φ) of a vertex should be symmetric to its context element

interpretation (φ_Γ), i.e. $\varphi = \varphi_\Gamma$. The combination of random walk context sampling and and the feature space symmetry assumption can be used to compute vertex embeddings even for very large graphs.

Graph Embeddings Skip-gram methods can not only be used for vertex embeddings but also to embed entire graphs. One way to do this is via the `graph2vec` [Nar+17] method. It is inspired by `doc2vec` [LM14] which in turn is based on `word2vec`. `graph2vec` gets a set of graphs $\mathcal{G}' = \{G_1, \dots G_N\}$ as input and outputs graph embeddings $\varphi(G_1), \dots, \varphi(G_N)$. While in `word2vec` every word can be a context root as well as a context element, `graph2vec` uses the graphs $\mathcal{G}'$ as context roots. The context $\Gamma_T(G_i)$ of a graph G_i is defined as

$$\Gamma_T(G_i) \coloneqq \bigcup_{v_j \in \mathcal{V}_{G_i}} \left\{ \chi^{(t)}_{G_i,1}(v_j) \right\}_{t=0}^{T} \quad \text{with } T \in \mathbb{N} \text{ and } \chi^{(t)}_{G_i,1} \text{ as in definition 2.9.} \tag{2.5}$$

Intuitively this context can be understood as the set of 1-WL-distinguishable subgraphs of G_i with diameter $\leq 2T$. Since WL is used to identify distinct subgraphs, `graph2vec` can only be applied to graphs with discrete vertex labels. Using the previous definitions, the context root embedding function has the signature $\varphi : \mathcal{G} \to \mathbb{R}^d$ while the context element embedding function is of type $\varphi_\Gamma : \bigcup_{G_i \in \mathcal{G}} \Gamma_T(G_i) \to \mathbb{R}^d$. To find those embeddings the `word2vec` objective from eq. (2.4) is reused. Analogous to `word2vec`, `graph2vec` therefore embeds graphs that share subgraphs close to each other, whereas graphs that do not share substructures tend to be embedded further away from each other.

2.3.2 Graph Kernels

Instead of mapping a graph into an explicitly defined vector space of fixed dimension, one can also do so implicitly by employing the kernel trick. There is a large variety of so-called *graph kernels* (GKs) to do this. GKs can be used in combination with any kernelized learner, typically SVMs, to solve the GC/GR problem. While there is a large variety of different GKs [Kri+20], we will focus on those that are based on the previously described family of WL algorithms.

WL subtree kernel One well-known GK is based directly on the 1-WL coloring algorithm, the so-called *WL subtree kernel* [She+11]. It is uses the mapping

$$\varphi_{\mathrm{ST}}(G) \coloneqq \bigoplus_{t=0}^{T} \left(\mathit{dist}_{\chi^{(t)}_{G,1}}(c) \right)_{c \in \mathcal{C}} \quad \text{with } \oplus \text{ denoting vector concatenation.} \tag{2.6}$$

$\varphi_{\mathrm{ST}}(G)$ encodes the color counts across a fixed number T of 1-WL refinement steps. Via this mapping, the WL subtree kernel function $k_{\mathrm{ST}} : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ can then be simply written as the standard inner vector product $k_{\mathrm{ST}}(G, H) \coloneqq \langle \varphi_{\mathrm{ST}}(G), \varphi_{\mathrm{ST}}(H) \rangle$.

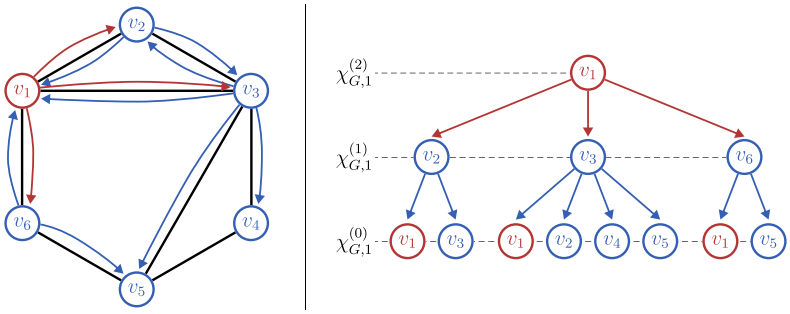


Figure 2.10. Illustration of the correspondence between the WL color $\chi^{(t)}_{G,1}(v)$ and the breadth-first subtree of depth t rooted at v (for $t = 2$ and $v = v_1$). An arrow $v \dashrightarrow u$ indicates that the refined color of v depends on the color of u. BASED ON: [SHE+11]

Figure 2.10 illustrates how this definition relates to subtrees. Since all vertex colors $c \in \mathcal{C}$ correspond to a subtree isomorphism class, $k_{\mathrm{ST}}(G, H)$ effectively computes the similarity of G and H by comparing their local subtree structures of depth at most T. Note that the underlying mapping φ_{ST} cannot be computed independently for each graph $G \in \mathcal{G}_{\mathcal{D}} \subseteq \mathcal{G}$ in a given training dataset $\mathcal{D}$ since the set of colors $c \in \mathcal{C}$ introduced by the hashing function h is conjointly determined by all graphs $\mathcal{G}_{\mathcal{D}}$ (see definition 2.9). Consequently the dimensionality d of $\varphi_{\mathrm{ST}}(G) \in \mathbb{N}^d$ varies, depending on the entire dataset $\mathcal{D}$.

WL shortest path kernel Extending the idea of the WL subtree kernel, the *WL shortest path kernel* [She+11][BK05] adds global structural information to the local subtree comparisons. The i-th component of a graph's shortest path vector $\varphi_{\mathrm{SP}}(G)$ corresponds to the 4-tuple $(t_i, a_i, b_i, d_i) \in \{0, \ldots, T\} \times \mathcal{C} \times \mathcal{C} \times \mathbb{N}_0$ and its value is described by

$$\varphi_{\mathrm{SP}}(G)[i] \coloneqq \left| \left\{ (v, u) \in \mathcal{V}_G^2 \,|\, \chi^{(t_i)}_{G,1}(v) = a_i \wedge \chi^{(t_i)}_{G,1}(u) = b_i \wedge d_{\mathrm{SP}}(v, u) = d_i \right\} \right|, \quad (2.7)$$

with $d_{\mathrm{SP}}(v, u)$ denoting the length of the shortest path from v to u in G. Analogous to k_{ST}, the shortest path kernel function is defined as $k_{\mathrm{SP}}(G, H) \coloneqq \langle \varphi_{\mathrm{SP}}(G), \varphi_{\mathrm{SP}}(H) \rangle$. Since $\varphi_{\mathrm{SP}}(G)$ has one component for each possible 4-tuple (t_i, a_i, b_i, d_i), its dimensionality grows quadratically with the total number of introduced 1-WL colors $\mathcal{C}$ and linearly with the length of the longest shortest path $\max_{G \in \mathcal{G}_{\mathcal{D}}, (v,u) \in \mathcal{V}_G^2} d_{\mathrm{SP}}(v, u)$. While this approach is computationally significantly more expensive than the subtree kernel, it also has a higher discriminative power and allows the shortest path kernel to distinguish even some 1-WL indistinguishable graphs. The subtree kernel only compares graphs by the presence of local subtrees; the shortest path kernel additionally checks how well the distances between those subtrees align. Looking back at the 1-WL indistinguishable "six-cycle"/"two-triangles" example from fig. 2.5, we see that this additional information distinguishes the two graphs due to the fact that the longest shortest path in a six-cycle has length 3 while the longest shortest path in a triangle is of length 1.

Higher dimensional WL kernels Instead of incorporating shortest path information to increase the power of the WL kernel, one can alternatively just increase the WL-dimension k. A naïve generalization of the 1-WL subtree kernel would simply use eq. (2.6) but with the k-WL colorings $\chi^{(t)}_{G,k}$ instead of $\chi^{(t)}_{G,1}$. The problem with this approach is that the runtime of WL increases exponentially with k; even for $k = 2$ the cost often becomes infeasibly high when working with large graphs. To tackle this problem Morris et al. [Mor+17] proposed a combination of two optimizations:

1. **k-multisets:** The first optimization is to assign colors to vertex k-multisets instead of vertex k-tuples, where a k-multiset is any multiset $s \subseteq \mathcal{V}_G$ with $|s| = k$. This reduces the amount of colors that have to be refined in each WL iteration by a factor of $k!$. Based on definition 2.10, the multiset color refinement step is defined as

$$\chi^{(t+1)}_{G,k}(s) \coloneqq h\left(\chi^{(t)}_{G,k}(s), \{\!\{\chi^{(t)}_{G,k}(s \setminus \{v\} \cup \{u\}) \mid v \in s\}\!\} \mid u \in \mathcal{V}_G\}\!\}\right). \tag{2.8}$$

   Even though this simplification generally reduces the discriminative power of the kernel, for $k = 2$ specifically no information and therefore no power is lost by replacing the tuples (v, u) and (u, v) with $\{\!\{v, u\}\!\}$ since the undirectedness of graphs implies that $\chi^{(t)}_{G,2}(v, u) = \chi^{(t)}_{G,2}(u, v)$ even when doing tuple-based color refinement.

2. **Neighborhood localization:** The second optimization uses the fact that most real-world graphs tend to be sparse [Chu10], i.e. $|\mathcal{E}_G| = \mathcal{O}(|\mathcal{V}_G|)$. The multiset-based k-WL algorithm refines the color of a k-multiset s by hashing the colors of its neighbors as defined in eq. (2.8) where there is one neighbor per vertex $u \in \mathcal{V}_G$. By the sparsity assumption, most of those vertices u are however not connected with any of the vertices in s, i.e. $u \notin \Gamma_G(s)$ with $\Gamma_G(s) \coloneqq \bigcup_{v \in s} \Gamma_G(v)$. The refinement runtime can therefore often be reduced significantly by only considering "local" neighbors $u \in \Gamma_G(s)$ instead of the "global" neighborhood $u \in \mathcal{V}_G$ in eq. (2.8).

We call the kernel that only uses the first optimization *k-GWL* (k-dim. global WL) and the kernel that uses both optimizations *k-LWL* (k-dim. local WL). Morris et al. [Mor+17] have empirically shown that focusing on local graph structures via LWL often actually performs better than the computationally more expensive GWL kernel.

2.3.3 Graph Neural Networks

The last family of GC/GR approaches we will look at is that of *graph neural networks* (GNNs). The idea to feed a graph into a *neural network* (NN) was first described by Gori et al. [Gor+05]. Since then many variants and extensions of that idea have been

proposed [Wu+19]. We will focus specifically on the so-called *graph convolutional neural networks* (GCNNs) which can be divided into two variants: Spectral and spatial GCNNs.

Spectral GCNNs

The class of spectral GCNNs is motivated by spectral graph theory (see section 2.2.3). A spectral GCNN expects a graph G with real vertex feature vectors $x_G[v_i] \in \mathbb{R}^d$ as its input. Those feature vectors often are one-hot encodings of the labels $l_G[v_i]$ or, if no such information is provided, vertex embedding vectors (see section 2.3.1).

Definition 2.22. We call $X_G \coloneqq \begin{pmatrix} x_G[v_1] \\ \vdots \\ x_G[v_n] \end{pmatrix} \in \mathbb{R}^{n \times d}$ the *vertex feature matrix* of G.

Note that each of the d columns of X_G can be interpreted as a separate graph signal function $x_{G,j} : \mathcal{V}_G \to \mathbb{R}$ for $j \in [d]$. The core idea of spectral GCNNs is to learn a graph convolution kernel g, similar to the grid kernels in conventional *convolutional neural networks* (CNNs) [LB98]. The problem with this idea is that the convolution operation $g * x$ requires some notion of distance in order to "move" the kernel g over the function x. Graphs do not generally satisfy this requirement, i.e. the notion of a vertex distance $\|v_i - v_j\|$ is not clearly defined. Via the convolution theorem $g * x = \mathcal{F}^{-1}(\hat{g} \odot \hat{x})$ we can however still define a graph convolution operator that works directly in the spectral domain:

$$g * x_{G,j} \coloneqq U_G^\top (\hat{g} \odot (U_G x_{G,j})) \text{ with } \odot \text{ denoting element-wise multiplication.} \quad (2.9)$$

To see the connection to the convolution theorem, remember that the Laplacian eigenvector matrices U_G and $U_G^\top$ correspond to the FT $\mathcal{F}$ and inverse FT $\mathcal{F}^{-1}$ respectively. In this formulation of convolution the Fourier transformed kernel $\hat{g}$ can be interpreted as a spectral filter, i.e. it dampens or amplifies certain eigenvector components of a graph. Bruna et al. [Bru+13][Hen+15] first described a GNN architecture that learns such a spectral filter $\hat{g}$. Since a GNN has to accept many different graphs of varying size n, the filter can however not be learned as a direct mapping $\hat{g} : [n] \to \mathbb{R}$. Therefore it is not expressed in terms of the indices i of specific eigenvectors $u_{G,i}$ but instead in terms of the corresponding eigenvalues[3] via $\hat{g}(\Lambda_G) = \begin{pmatrix} \hat{g}(\lambda_{G,1}) & & \\ & \ddots & \\ & & \hat{g}(\lambda_{G,n}) \end{pmatrix}$. In order to learn such an eigenvalue filter $\hat{g} : \mathbb{R} \to \mathbb{R}$ via gradient descent, it requires some differentiable parameterization.

[3] Here either the unnormalized eigenvalues of L_G or the eigenvalues of some normalized Laplacian, e.g. L_G^{sym}, can be used.

Chebyshev filters One such parameterization is based on the family of recursively defined Chebyshev polynomials $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0 = 1$ and $T_1 = x$ [Def+16]. It describes a spectral filter as a linear combination $\hat{g}_\theta(\lambda) \coloneqq \sum_{k=0}^{K-1} \theta_k T_k(\lambda)$ with $\theta \in \mathbb{R}^K$. This restricts $\hat{g}_\theta$ to be a polynomial which allows us to rewrite the graph convolution from eq. (2.9) as

$$g_\theta * x_{G,j} \;=\; U_G^\top \hat{g}_\theta(\tfrac{2}{\max \lambda_G}\Lambda_G - I)\, U_G\, x_{G,j} \;=\; \hat{g}_\theta(\tfrac{2}{\max \lambda_G}L_G - I)\, x_{G,j} \qquad (2.10)$$

because $L_G = U_G^\top \Lambda_G U_G$ is an eigendecomposition of the Laplacian. The $\frac{2}{\max \lambda_G}\Lambda_G - I$ term normalizes the eigenvalues to $[-1, 1]$ which prevents vanishing and exploding gradients. The advantage of this formulation is that it can be evaluated without actually having to compute the expensive eigendecomposition of L_G. Instead, by interpreting $\hat{g}_\theta$ as a matrix polynomial, one only has to compute the powers $L_G^1, \ldots, L_G^{K-1}$ and the largest eigenvalue $\max \lambda_G$ which is generally much cheaper than computing the full spectrum.

Linear filters Kipf and Welling [KW17] simplify the Chebyshev filter from eq. (2.10) even further in the ambiguously named *graph convolutional network* (GCN) architecture[4]. By fixing $K = 2$, by using a single filter parameter $\theta \in \mathbb{R}$ and by assuming that $\max \lambda_G^{\text{sym}} \approx 2$ (see definition 2.21, page 15) the convolution operation is reduced to

$$g_\theta * x_{G,j} = \theta \left(\tilde{D}_G^{-\frac{1}{2}} \tilde{A}_G \tilde{D}_G^{-\frac{1}{2}} \right) x_{G,j} \quad \text{with } \tilde{A}_G = A_G + I \text{ and } \tilde{D}_G = D_G + I. \quad (2.11)$$

In this formulation the spectral filter is effectively just a linear function $\hat{g}_\theta(\lambda) = \theta\lambda$. Via this simplified notion of convolution for a single feature signal $x_{G,j}$, a convolutional neural network layer over all features $\{x_{G,j}\}_{j=1}^d$ can analogously be defined as

$$Z_G^{(t)} \coloneqq \sigma\left(\hat{A} Z_G^{(t-1)} \Theta^{(t)} \right) \quad \text{with } \hat{A} \coloneqq \tilde{D}_G^{-\frac{1}{2}} \tilde{A}_G \tilde{D}_G^{-\frac{1}{2}} \quad \text{and} \quad Z_G^{(0)} \coloneqq X_G. \qquad (2.12)$$

Here σ is some non-linearity, e.g. ReLU, and $\Theta^{(t)} \in \mathbb{R}^{d^{(t-1)} \times d^{(t)}}$ is a matrix of learned filter parameters. This type of convolutional layer takes a graph signal $Z_G^{(t-1)} \in \mathbb{R}^{n \times d^{(t-1)}}$ and the corresponding adjacency matrix $A_G \in \mathbb{R}^{n \times n}$ as input and outputs a convolved signal $Z_G^{(t)} \in \mathbb{R}^{n \times d^{(t)}}$. By stacking multiple of those convolutional layers, complex non-linear signal filters can be learned despite the linearity of the underlying spectral filter.

[4] We use "GCN" to refer to their proposed specific method in order to be consistent with other literature. We use "GCNN" to refer to the broader family of graph convolutional neural network methods.

Spatial GCNNs

Looking at the GCN layer defined in eq. (2.12), notice that, even though it is motivated by spectral graph theory, the spectrum λ_G is not actually directly used. This allows for a very different perspective on GCNs: Instead of interpreting convolutions as applications of linear spectral filters, we can interpret them as vertex neighborhood aggregations, similar to 1-WL. From this perspective a GCN can be seen as a so-called spatial GCNN because it operates directly in the vertex domain. To see why this is the case, we rewrite eq. (2.12) and compare it with the 1-WL color refinement strategy (see definition 2.9, page 8):

$$\begin{aligned} \text{GCN:} \quad & Z_G^{(t)}[v] = \sigma\left(\left(\mu(v)^2 Z_G^{(t-1)}[v] + \sum_{u \in \Gamma_G(v)} \mu(v)\mu(u) Z_G^{(t-1)}[u]\right)\Theta^{(t)}\right) \\ \text{1-WL:} \quad & \chi_{G,1}^{(t)}(v) = h\left(\chi_{G,1}^{(t-1)}(v), \{\!\{\chi_{G,1}^{(t-1)}(u) \mid u \in \Gamma_G(v)\}\!\}\right) \end{aligned} \tag{2.13}$$

Here $\mu(v) \coloneqq (|\Gamma_G(v)| + 1)^{-\frac{1}{2}}$ are normalization factors introduced by $\tilde{D}_G^{-\frac{1}{2}}$. The comparison shows that GCN convolutions can be understood as iterative refinements of continuous "color vectors" $Z_G^{(t)}[v] \in \mathbb{R}^{d^{(t)}}$. Then the continuous analogue to the injective 1-WL hash function $h : \mathcal{C}^* \to \mathcal{C}$ can be defined as $h_{\text{GCN}}(z_0, z_1, \ldots, z_m) = \sigma(\sum_{j=0}^{m} \mu(v_0)\mu(v_j) z_j \Theta^{(t)})$ which computes a single color vector from multiple color vectors z_j where v_j are the corresponding vertices.

A 1-WL bound on GNN power In the next step we will take a closer look at the relation between GCNNs that use vertex neighborhood convolution and 1-WL by comparing their respective discriminative power.

Proposition 2.23 (see Xu et al. [Xu+19, lemma 2 and theorem 3] for the proof)**.** *The discriminative power of a GCNN that convolves vertex feature vectors via a convolution operator of the form* $Z_G^{(t)}[v] = h^{(t)}\left(Z_G^{(t-1)}[v], \{\!\{Z_G^{(t-1)}[u] \mid u \in \Gamma_G(v)\}\!\}\right)$ *is upper-bounded by that of 1-WL, where* $h^{(t)} : \left(\mathbb{R}^{d^{(t-1)}}\right)^* \to \mathbb{R}^{d^{(t)}}$ *is an arbitrary vertex neighborhood hashing function. Moreover, iff.* $h^{(t)}$ *is injective, the GCNN has the same discriminative power as 1-WL.*

By proposition 2.23 the GCN architecture is strictly less powerful than 1-WL due to the fact that h_{GCN} is not an injective hash function. To overcome this limitation Xu et al. [Xu+19] proposed an alternative vertex neighborhood convolution architecture called *graph isomorphism network* (GIN):

$$Z_G^{(t)}[v] \coloneqq \text{MLP}^{(t)}\left(Z_G^{(t-1)}[v] + \sum_{u \in \Gamma_G(v)} Z_G^{(t-1)}[u]\right) \tag{2.14}$$

By leaving out the normalization factors $\mu(v)$ and by using a MLP instead of the

single fully-connected layer $\sigma \circ \Theta^{(t)}$ the resulting hashing function $h_{\mathrm{GIN}}(z_1, \ldots, z_m) = \mathrm{MLP}^{(t)}(\sum_{j=1}^{m} z_j)$ becomes injective[5], i.e. it assigns a unique color vector to each multiset of color vectors. By proposition 2.23 this implies that GIN has the same discriminative power as 1-WL. As expected, GIN therefore fits training data better than non-injective GCNNs like GCN. Interestingly GIN additionally seems to generalize better on test data than non-injective methods [Xu+19][Err+20]. It is not yet fully understood why this is the case.

Higher dimensional WL GNNs The majority of GNN methods uses a vertex neighborhood convolution scheme which limits their discriminative and, more importantly, computational power to that of 1-WL (see definition 2.16, page 11). To go beyond the limits of 1-WL, Morris et al. [Mor+19] proposed the first GCNN architecture inspired by the higher dimensional k-WL algorithms. Similar to the k-LWL kernel which we looked at in section 2.3.2, their so-called k-GNN uses the k-multiset and neighborhood localization optimizations to keep runtime costs feasible. The k-GNN convolution is described by

$$Z_G^{(t)}[s] \coloneqq \sigma\left(Z_G^{(t-1)}[s]W^{(t)} + \sum_{v \in s, u \in \Gamma_G(v)} Z_G^{(t-1)}[s \setminus \{v\} \cup \{u\}]W_\Gamma^{(t)}\right). \tag{2.15}$$

It convolves the feature vectors of k-multisets $s \subseteq \mathcal{V}_G$ instead of single vertices $v \in \mathcal{V}_G$. Apart from this fundamental difference to 1-WL bounded GNNs, k-GNNs additionally use two learnable parameter matrices $W^{(t)}, W_\Gamma^{(t)} \in \mathbb{R}^{d^{(t-1)} \times d^{(t)}}$ instead of the single $\Theta^{(t)}$ used in GCNs. This separation of the parameter matrix effectively fulfills the same purpose as the MLP in GIN; it makes the implicitly defined k-GNN hash function injective. This can be seen by realizing that a stack of k-GNN convolution layers with $W_\Gamma^{(t)} = \mathbf{0}$ for all but the first layer simulates a MLP.

As a final remark, note that the concept of "neighborhood" in eq. (2.15) is slightly different from that used by the k-WL algorithm (see definition 2.10, page 9): k-GNN considers a single k-multiset as a neighbor while k-WL uses k-sets of k-multisets as neighbors. We will get back to this difference later in section 4.2.

Graph Pooling

The graph convolution approaches that we just looked at all produce a set of convolved feature vectors $\left\{Z_G^{(T)}[s_i]\right\}_{i=1}^{m}$ after T convolutional layers. For 1-WL bounded architectures like GCN or GIN, there is one vector for each of the $m = |\mathcal{V}_G|$ vertices, while a k-WL inspired architecture like k-GNN produces $m \leq |\mathcal{V}_G|^k$ output vectors.

[5] Under the assumption that the set of possible input feature matrices $X \in \mathbb{R}^{n \times d^{(0)}}$ is countable. This is reasonable since real-world GC/GR domains typically only have a finite set of possible vertex feature vectors, e.g one-hot label encodings.

In order to solve the GC/GR problem, those vector sets need to be combined into a final predicted class or regression value. To do this, so-called *graph pooling layers* are used. Generally speaking there are two types of pooling approaches:

1. **Hierarchical Pooling:** This type of pooling is similar to that found in CNNs on pictures where the resolution is iteratively reduced. Similarly hierarchical graph pooling iteratively *coarsens* a graph every couple of convolutional layers. Graph coarsening works by merging vertices in a spectrum-preserving manner [LV18]. This is done until a single merged vertex is left whose feature vector then represents the entire graph [Yin+18].

2. **Global Pooling**: Alternatively pooling can also be performed in a single step after the convolutional layers. This type of pooling takes all m feature vectors and directly maps them to a single graph feature vector.

In this thesis we will focus on global pooling. The most simple global pooling approaches use static aggregation functions like component-wise $\min$, $\max$ or mean to produce a single graph feature vector $z_G \in \mathbb{R}^{d^{(T)}}$. This vector is then fed into a standard MLP to produce the final prediction. The state-of-the-art graph pooling approaches go beyond static aggregation and try to incorporate structural information. We will now briefly look at two such approaches.

SortPooling One way to combine feature vectors is called *SortPooling* [Zha+18]. It interprets the convolved vectors as continuous WL colors that encode different structural roles of vertices (or vertex k-multisets in case of k-GNNs). By imposing a component-wise lexicographic ordering on the set of color vectors, it can be reduced to a vertex permutation invariant top-p list of vectors $(z_1, \dots, z_p)$ for some fixed $p \in \mathbb{N}$. The final graph feature vector then is the concatenation $z_G = (\bigoplus_{i=1}^{p} z_i) \in \mathbb{R}^{d^{(T)}p}$. SortPooling is based on the idea that the convolutional layers will learn to use the most-significant lexicographic vector components to represent that vector's importance.

SAGPooling Another global pooling approach uses self-attention to explicitly assign a structural importance score to each vertex (or vertex set in case of k-GNNs). The so-called *self-attention graph pooling* (SAGPooling) [Lee+19] learns those attention scores via a separate stack of convolutional layers. The set of convolved feature vectors is then filtered down from the size m to $p = \lceil rm \rceil$ with $r \in [0, 1]$ by removing the vectors of the vertices with the lowest attention scores. Let $z_1, \dots, z_p$ be the remaining feature vectors. To obtain the final graph feature vector, SAGPooling uses $z_G = (\sum_{i=1}^{p} z_i) \oplus \max_{i=1}^{p} z_i$.

Learning to Aggregate on Graphs

3

In the previous chapter an introduction to two separate fields of research was given: 1. *Learning to aggregate* (LTA), 2. *Graph classification and regression* (GC/GR). In this chapter we will combine them and define an extension of LTA to the GC/GR problem. This will be done in three steps:

1. We begin with a formal definition of what actually constitutes an LTA method as opposed to non-LTA methods.

2. Using this definition, we will see that an SVM that uses one of the previously described graph embedding approaches can be interpreted as an LTA variant under certain conditions.

3. In the last step we look at the previously described GCNN architectures and formalize their relation to LTA.

3.1 A Generalized Definition of LTA

In order to formally define LTA, we must first decide on its defining characteristic. We propose that this characteristic should be the *localized explainability* of LTA predictions. As seen in section 2.1, an LTA score $y_C \in \mathcal{Y}$ for some multiset composition $C = \{\!\{c_1, \dots, c_n\}\!\}$ can always be tracked back to a set of local constituent scores $y_1, \dots, y_n \in \mathcal{Y}$. Under the assumption that each constituent c_i represents some human interpretable object, a composition's score y_C can therefore be explained by the presence of certain indicative constituents/objects c_i.

Based on this intuition we now give a generalized definition of LTA which applies to unstructured as well as structured input data. We assume that all compositions are represented by graphs $G \in \mathcal{G}$; an unstructured input is represented by a graph with one vertex per constituent ($\mathcal{V}_G = \{v_{c_1}, \dots, v_{c_n}\}$) and no edges ($\mathcal{E}_G = \emptyset$). Each composition has some target score $y_G \in \mathcal{Y}$ which could be a discrete class or continuous value. An *LTA model* $h : \mathcal{G} \to \mathcal{Y}$ assigns predictions $\hat{y}_G \in \mathcal{Y}$ to compositions G which ideally correspond to the true score y_G. Such a model must satisfy three criteria:

1. **Decomposition:** A given composition G must be decomposed into a set of constituents $c_{G,i}$ via a *decomposition function* $\psi : \mathcal{G} \to \mathcal{P}(\mathcal{G})$.

Definition 3.1. ψ is a *decomposition function* iff. it splits a graph into a subset of its subgraphs, i.e. $\forall G \in \mathcal{G} : \forall c_{G,i} \in \psi(G) : \exists s \in \mathcal{P}(\mathcal{V}_G) : c_{G,i} = G[s]$.

Note that the strict equality $c_{G,i} = G[s]$ is used in definition 3.1 instead of just requiring subgraph isomorphism ($c_{G,i} \simeq G[s]$) because a constituent $c_{G,i}$ represents a specific localized subgraph of a structured composition.

In the existing unstructured LTA approaches the decomposition function is implicitly defined as $\psi(G) \coloneqq \{G[v_{c_i}]\}_{v_{c_i} \in \mathcal{V}_G}$ since each vertex v_{c_i} corresponds to an interpretable constituent c_i by definition. For structured data however, a split into individual vertices is typically not appropriate. Molecular graphs from chemical datasets for example are meaningfully characterized by the presence of so-called *functional groups* consisting of multiple bonded atoms while a characterization on the level of individual atoms is generally less meaningful [MW97].

2. **Local evaluation:** The constituents $c_{G,i} \in \psi(G)$ must be evaluated via some function $f : \mathcal{G} \to \mathcal{Y} \times \mathbb{R}$. This *evaluation function* assigns a prediction $\hat{y}_{G,i} \in \mathcal{Y}$ and a weight $w_{G,i} \in \mathbb{R}$ to each constituent. A constituent's weight $w_{G,i}$ can intuitively be interpreted as a measure of the confidence that the local prediction $\hat{y}_{G,i}$ is indicative of the composition's global target score y_G. Learning local predictions and weights for all possible constituents is called the *disaggregation problem*.

   Note that there are no explicit constituent weights in the existing unstructured LTA approaches (i.e. implicitly all $w_{G,i} = 1$) because the explicitly given constituents are assumed to be equally indicative of y_G. For structured data however, where the decomposition $\psi(G)$ is not given as part of the input, this assumption does not necessarily hold. By weighting the constituents, an LTA model can reduce the relevance or even ignore constituents that turn out to be irrelevant for the compositions target score y_G.

3. **Aggregation:** Lastly a *weighted aggregation function* $\mathcal{A} : (\mathcal{Y} \times \mathbb{R}_{\geq 0})^* \to \mathcal{Y}$ must be applied. It combines the multiset of local constituent predictions and non-negative weights into a single global composition prediction.

   Definition 3.2. We call $\mathcal{A}$ a *weighted aggregation function* iff. it satisfies

   $$\text{idempotency: } \exists \eta > 0 : \forall y \in \mathcal{Y}, w \in \mathbb{R}^n_{\geq 0} \text{ s.t. } \max w \geq \eta : \mathcal{A}(\{\!\{(y, w_i)\}\!\}_{i=1}^n) = y$$
   $$\wedge \text{ zero invar.: } \forall y_0 \in \mathcal{Y}, S = \{\!\{(y_i, w_i)\}\!\}_{i=1}^n : \mathcal{A}(S \cup \{(y_0, 0)\}) = \mathcal{A}(S).$$

   The idempotency constraint requires aggregation functions to agree with uniform input scores y if at least one input is weighted above some threshold η. The zero invariance constraint requires aggregation functions to ignore inputs

with zero weight. Exemplary weighted aggregation function are:

- The *weighted mean* function $\mathrm{wmean}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^n) := \sum_{i=1}^n w_i y_i$ which requires $\sum w_i = 1$, $w_i \in [0, 1]$ and the existence of some scalar multiplication operator $\cdot : \mathbb{R} \times \mathcal{Y} \to \mathcal{Y}$, typically from some subfield $\mathcal{Y} \subseteq \mathbb{R}$ or possibly also some vector subspace $\mathcal{Y} \subseteq \mathbb{R}^d$.

- Another basic weighted aggregator is the *weighted majority vote* function $\mathrm{wmaj}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^n) := \arg\max_{y \in \mathcal{Y}} \sum_{y_i = y} w_i$ which returns the input with the highest total weight. Unlike wmean it can also be applied to score domains $\mathcal{Y}$ without a multiplication operator, e.g. sets of discrete classes.

- Alternatively an unweighted aggregation function like $\min$, $\max$, mean or OWA[1] also trivially satisfies definition 3.2 if all inputs y_i with $w_i = 0$ are filtered out and the weights for all remaining inputs are ignored.

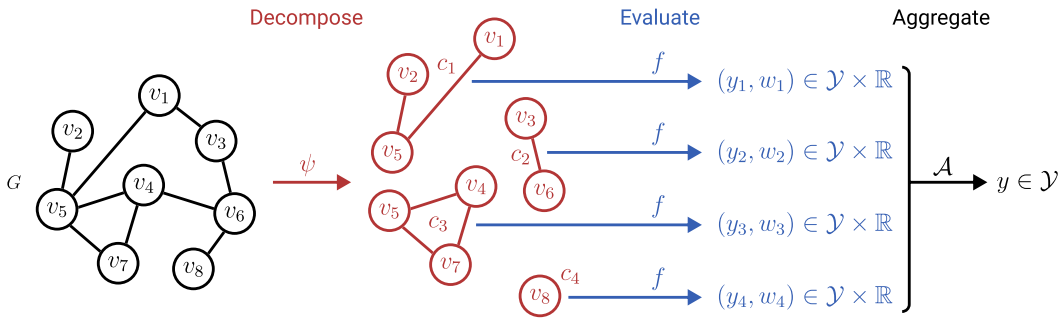


Figure 3.1. Overview of the generalized LTA architecture for structured data.

Based on the notion of decomposition, local evaluation and aggregation we can now define the concept of *LTA formulations*.

Definition 3.3. A model $h : \mathcal{G} \to \mathcal{Y}$ is in an *LTA formulation* iff. it is expressed as

$$h(G) := \mathcal{A}(\{\!\{ f(c_{G,i}) \mid c_{G,i} \in \psi(G) \}\!\}) \quad \text{with } \psi,\ f \text{ and } \mathcal{A} \text{ as defined above.}$$

Note that every model $h : \mathcal{G} \to \mathcal{Y}$ has a trivial recursive LTA formulation by choosing $\psi(G) = \{G\}$, $f(G) = (h(G), 1)$ and an arbitrary weighted aggregation function $\mathcal{A}$. Those trivial LTA formulations do not split compositions into locally evaluated constituents and therefore intuitively do not fulfill the postulated localized explainability characteristic of LTA. However since there is no commonly accepted formal criterion to decide whether a model's decisions are explainable [Lip18], we do not attempt to strictly distinguish between LTA and non-LTA methods. Instead the notion of LTA formulations should be seen as way to identify how "LTA-like" a model is:

[1] Even though OWA uses weights, it does not get those weights as part of the input and is therefore considered to be unweighted in this context.

- **Negative extreme:** If a model h only has trivial LTA formulations with the decomposition function $\psi(G) = \{G\}$, it is not considered to be an LTA model.
- **Positive extreme:** If a model has an LTA formulation with a decomposition function that returns interpretable constituents, it is considered to be an LTA model. By definition this is true for the single-vertex constituents $\psi(G) \coloneqq \{G[v_{c_i}]\}_{v_{c_i} \in \mathcal{V}_G}$ of LTA methods for unstructured data.
- **In-between cases:** An LTA method for structured data must produce models that lie somewhere in-between the two extremes.

The more "LTA-like" a given model is, the stronger its bias towards locally explainable predictions, which in turn reduces the potential expressive power of the model. Gilpin et al. [Gil+18] describe this trade-off between explainability and expressive power in more detail. However, when considering problem domains in which the true composition scores y_G are accurately described by an LTA-like generative process, a less expressive LTA-like model could generalize better than a more expressive non-LTA model. This idea is captured by the so-called *LTA assumption*.

Definition 3.4. A problem domain $\mathcal{D}$ satisfies the *LTA assumption* iff. there is an LTA method which produces models with an equal or lower out-of-sample error than the models produced by non-LTA methods for most training samples $\mathcal{D}_{\text{train}} \subseteq \mathcal{D}$.

Due to the fuzziness of the class of LTA methods, the LTA assumption is naturally also a fuzzy concept. Nonetheless evidence for its truthiness in a given domain $\mathcal{D}$ can be empirically obtained by comparing candidate LTA methods with the best known non-LTA method for $\mathcal{D}$, assuming that some cut-off condition for the required "LTA-ness" of an LTA method is agreed upon.

3.2 SVMs with Graph Embeddings as LTA Models

Based on the general definition of LTA from the last section, we will now see to what extent the GC/GR methods described in section 2.3 can be interpreted as LTA instances. This section explores the relation between SVMs that use graph embeddings and LTA. In sections 2.3.1 and 2.3.2 three different ways to map a given graph G to a vector $\varphi(G) \in \mathbb{R}^d$ were described: 1. Fingerprint embeddings, 2. skip-gram inspired embeddings and 3. kernel embeddings. One common way to solve the GC/GR problem via those embedding vectors is to train an SVM on them. We will now see that SVMs can be interpreted as LTA models if they are trained on so-called *substructure component embeddings* (SSCEs).

Definition 3.5. Given a a multiset $A = \{\!\{ \underbrace{a_1, \ldots, a_1}_{\gamma_A(a_1) \text{ times}}, \ldots, \underbrace{a_n, \ldots, a_n}_{\gamma_A(a_n) \text{ times}} \}\!\} \subseteq D$, the so-

called *multiplicity function* $\gamma_A : D \to \mathbb{N}_0$ maps each element of the domain D to its multiplicity in A with $\gamma_A(x) = 0 \Leftrightarrow x \notin A$.

Definition 3.6. A graph embedding $\varphi : \mathcal{G} \to \mathbb{N}_0^d$ is called a *substructure component embedding* (SSCE) iff. there are decomposition functions $\psi_{\varphi,i} : \mathcal{G} \to \mathcal{P}(\mathcal{G})$ and multiplicity functions $\gamma_{\varphi,i} : \mathcal{G} \to \mathbb{N}_0$ for all embedding components $i \in [d]$ s.t. $\forall G \in \mathcal{G}, i \in [d] : \varphi(G)[i] = \sum_{c \in \psi_{\varphi,i}(G)} \gamma_{\varphi,i}(c)$ where $\gamma_{\varphi,i}$ is the multiplicity function of a multiset of the constituents $\psi_{\varphi,i}(G)$. We call $\psi_{\varphi,i}$ an *underlying decomposition* of the i-th component of the embedding φ. Similarly the joint decomposition $\psi_\varphi(G) \coloneqq \bigcup_{i=1}^d \psi_{\varphi,i}(G)$ is an *underlying decomposition* of φ.

Intuitively definition 3.6 states that the value of each SSCE component must be derived from the number of constituents produced by some decomposition function where each constituent might be counted multiple times. Based on this requirement we now proof the main theorem which shows the relation between SVMs and LTA.

Theorem 3.7. *A binary SVM graph classifier h that applies a SSCE φ to its inputs has an LTA formulation that uses an underlying decomposition ψ_φ of φ.*

Proof. Let $h : \mathcal{G} \to \{-1, +1\}$ be a binary graph classifier expressed as $h = h_{\text{SVM}} \circ \varphi$ where $\varphi : \mathcal{G} \to \mathbb{N}_0^d$ is an SSCE and $h_{\text{SVM}} : \mathbb{R}^d \to \{-1, +1\}$ a standard SVM classifier. Additionally, let ψ_φ be some underlying decomposition of φ with $\{\gamma_{\varphi,i}\}_{i=1}^d$ being the corresponding multiplicity functions.

Based on this decomposition we now bring the SVM graph classifier h into an LTA formulation. If h is trained on a dataset $\mathcal{D}_{\text{train}} = \{(G_1, y_1), \ldots, (G_N, y_N)\}$, via the kernel trick it can be expressed as

$$
\begin{aligned}
h(G) &= \operatorname{sgn}\left(\sum_{j=1}^{N} \alpha_j y_j \langle \varphi(G), \varphi(G_j)\rangle + b\right) \quad \text{for some } \alpha \in \mathbb{R}_{\geq 0}^N \text{ and } b \in \mathbb{R} \\
&= \operatorname{sgn}\left(\sum_{i=1}^{d} \varphi(G)[i] \underbrace{\left(\sum_{j=1}^{N} \alpha_j y_j \varphi(G_j)[i]\right)}_{\beta_i} + b\right) = \operatorname{sgn}\left(\sum_{i=1}^{d} \varphi(G)[i]\beta_i + b\right) \\
&= \operatorname{sgn}\left(\sum_{c_t \in \psi_\varphi(G)} \underbrace{\sum_{i=1}^{d} \gamma_{\varphi,i}(c_t)\beta_i}_{z_t} + b\right) = \operatorname{sgn}\left(\sum_{c_t \in \psi_\varphi(G)} \underbrace{|z_t|}_{w_t} \underbrace{\operatorname{sgn} z_t}_{y_t} + \underbrace{|b|}_{w_b} \underbrace{\operatorname{sgn} b}_{y_b}\right) \\
&= \operatorname{wmaj}\left(\{\!\{(y_t, w_t) \mid c_t \in \psi_\varphi(G)\}\!\} \cup \{\!\{(y_b, w_b)\}\!\}\right).
\end{aligned}
$$

By choosing $f_h(c_t) \coloneqq (y_t, w_t)$ and $\mathcal{A}_h(S) = \operatorname{wmaj}(S \cup \{\!\{(y_b, w_b)\}\!\})$, the SVM model therefore has an LTA formulation with the decomposition function ψ_φ.

To complete the proof it now remains to show that f_h and $\mathcal{A}_h$ are in fact a local

evaluation function and a weighted aggregation function respectively. To see that f_h perform local evaluation, note that $f_h(c_t) \coloneqq (\operatorname{sgn} z_t, |z_t|)$ with $z_t \coloneqq \sum_{i=1}^{d} \gamma_{\varphi,i}(c_t)\beta_i$ only depends on the shared multiplicity functions $\gamma_{\varphi,i}$, the constants β_i and the constituent c_t; apart from c_t no other information from the input graph G is required by f_h which makes it a local evaluation function. To see why $\mathcal{A}_h$ satisfies definition 3.2, note that it inherits the idempotency property from wmaj because $\mathcal{A}_h$ ignores the bias "pseudo-constituent" (y_b, w_b) for all threshold weights $\max w_t \geq \eta > w_b$, similarly the zero invariance of wmaj is also directly inherited. This concludes the proof. □

The central statement of theorem 3.7 is that SVM graph classifiers that use an SSCE φ implicitly perform an LTA-like weighted majority vote aggregation of constituent scores y_t. Those constituent scores can in turn be expressed as a majority vote of the graph scores in the training dataset $\mathcal{D}_{\text{train}}$:

$$\begin{aligned} y_t \coloneqq \operatorname{sgn} z_t &= \operatorname{sgn}\left(\sum_{j=1}^{N} \underbrace{\left(\alpha_j \sum_{c_k \in \psi_\varphi(G_j)} \sum_{i=1}^{d} \gamma_{\varphi,i}(c_t)\gamma_{\varphi,i}(c_k)\right)}_{w_j \geq 0} y_j\right) \\ &= \mathrm{wmaj}(\{\!\{(y_j, w_j) \mid j \in [N]\}\!\}) \end{aligned} \tag{3.1}$$

Note that, if the constituents of G_j all come from different embedding components than those of G ($\forall i \in [d] : \psi_{\varphi,i}(G_j) \cap \psi_{\varphi,i}(G) = \emptyset$), then G_j has no influence on the score of G ($w_j = 0$).

To see what theorem 3.7 implies in practice, we will now apply it to the graph embedding approaches described in sections 2.3.1 and 2.3.2:

1. **Fingerprint embeddings:** Here the i-th embedding component represent the number of occurrences of a substructure S_i, i.e. $\varphi_{\mathrm{FP}}(G)[i] \coloneqq count_{S_i}(G)$ (see definition 2.16, page 11). Such a fingerprint embedding naturally is an SSCE with the decomposition functions $\psi_{\mathrm{FP},i}(G) \coloneqq \{G[\mathrm{set}(s)] \mid s \in \mathcal{V}_G^* \wedge G[s] \equiv S_i\}$ and the multiplicity functions $\gamma_{\mathrm{FP},i}(c) \coloneqq \mathbb{1}[c \simeq S_i]$ since those functions produce the subgraphs that are counted by $\varphi_{\mathrm{FP}}(G)[i]$ with multiplicity 1.

   Under the assumption that the substructure patterns S_i are chosen s.t. the their instances $c \in \psi_{\mathrm{FP},i}(G)$ are nontrivial interpretable constituents, the underlying joint decomposition $\psi_{\mathrm{FP}}(G) \coloneqq \bigcup_{i=1}^{d} \psi_{\mathrm{FP},i}(G)$ must also be nontrivial and interpretable. Thus, by theorem 3.7, SVMs trained on fingerprint embeddings are LTA models that produce locally explainable predictions. LTA

2. **`graph2vec`:** The skip-gram inspired `graph2vec` embedding produces vectors $\varphi(G) \in \mathbb{R}^d$ whose individual components do not have a clear interpretation. `graph2vec` embeds graphs that have common 1-WL colors closer to each other than graphs that do not share the same colors (see eq. (2.5), page 18). The

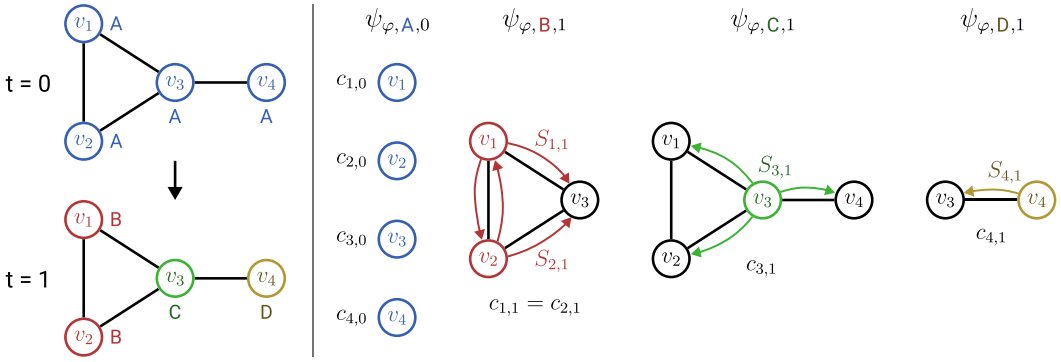


Figure 3.2. The constituents $c_{j,t}$ implied by the WL subtree embedding vector $\varphi(G) = (\underbrace{4,0,0,0}_{t=0}, \underbrace{0,2,1,1}_{t=1})$. The subtrees $S_{j,t}$ that span each constituent $c_{j,t}$ are visualized by the colored arrows that originate from the colored root vertices. Because $S_{1,1}$ and $S_{2,1}$ span the same constituent $c = c_{1,1} = c_{2,1} = G[\{v_1, v_2, v_3\}]$, its multiplicity is $\gamma_{\varphi,B,1}(c) = 2$; the multiplicity of all other constituents is 1.

resulting component values can be arbitrary reals, therefore this approach is not an SSCE which in turn implies that it does not have an LTA formulation as in theorem 3.7. non-LTA

3. **WL subtree kernel:** The individual components of the WL subtree kernel's embedding φ_{ST} represent the number of occurrences of a color $\kappa_i \in \mathcal{C}$, i.e. $\varphi_{\mathrm{ST}}(G)[i,t] := dist_{\chi^{(t)}_{G,1}}(\kappa_i)$ as defined in eq. (2.6) on page 18[2]. The occurrence of a color κ_i at some vertex $v_j \in \mathcal{V}_G$ in the t-th refinement step (i.e. $\chi^{(t)}_{G,1}(v_j) = \kappa_i$) in turn implies that the *breadth-first-search* (BFS) tree $S_{j,t} := \mathrm{BFS}(G, v_j, t)$ of depth t rooted at v_j is isomorphic to some WL color subtree S_{κ_i} (i.e. $S_{j,t} \simeq S_{\kappa_i}$). This relation between WL colors κ_i and subtrees S_{κ_i} was illustrated in fig. 2.10 on page 19.

   To show that φ_{ST} is an SSCE, we have to define decomposition functions $\psi_{\varphi,i,t}$ and multiplicity functions $\gamma_{\varphi,i,t}$ s.t. $\varphi(G)[i,t] = \sum_{c \in \psi_{\varphi,i,t}(G)} \gamma_{\varphi,i,t}(c)$. Since $\varphi(G)[i,t]$ counts the occurrences $S_{j,t}$ of a subtree S_{κ_i} in G, the SSCE requirement would be trivially satisfied if each $S_{j,t}$ were a constituent with multiplicity 1. This intuition is however not quite correct because the constituents of a graph G must be induced subgraphs of G, not BFS subtrees $S_{j,t}$ of G.

   To fix the previous intuition we "convert" the subtrees $S_{j,t}$ into proper subgraph constituents via $c_{j,t} := G[\mathcal{V}_{S_{j,t}}]$, i.e. the subgraphs that are spanned by the subtrees. The resulting decomposition functions are $\psi_{\varphi,i,t}(G) := \{c_{j,t} \mid v_j \in \mathcal{V}_G \wedge S_{j,t} \simeq S_{\kappa_i}\}$. As illustrated in fig. 3.2, the number of distinct constituents $c_{j,t}$ might be smaller than the number of BFS subtrees $S_{j,t}$ because two distinct subtrees might span the same set of vertices. To fix this discrepancy between the subtree occurrence count (which equals $\varphi(G)[i,t]$) and the number of con-

[2] To avoid confusion $\kappa_i \in \mathcal{C}$ is used for colors and $c_j \in \mathcal{G}$ for constituents in this context.

stituents, the multiplicities $\gamma_{\varphi,i,t}(c)$ of the constituents c have to correspond to the number of BFS subtrees they were spanned up by, i.e.

$$\gamma_{\varphi,i,t}(c) \coloneqq |\{S = \text{BFS}(c, v_{\text{root}}, t) \mid v_{\text{root}} \in \mathcal{V}_c \wedge S \simeq S_{\kappa_i} \wedge \mathit{complete}(S)\}| \quad (3.2)$$
$$\text{with } \mathit{complete}(S) \Leftrightarrow \forall \text{ leaf nodes } v_{\text{leaf}} \text{ of the tree } S : |\Gamma_c(v_{\text{leaf}})| = |\Gamma_G(v_{\text{leaf}})|.$$

The purpose of the $\mathit{complete}(S)$ condition is to only count the BFS subtrees of the constituent c that are also BFS subtrees of the complete graph G. To see why this is required, note that in fig. 3.2 the $c_{1,1}/c_{2,1}$ constituent contains the color subtree S_{B} three times (rooted at v_1, v_2 and v_3) even though G only contains it twice (rooted at v_1 and v_2). Also note that, since $\gamma_{\varphi,i,t}(c)$ must only depend on a given constituent c and not on the graph G it was decomposed from, the degree information $|\Gamma_G(v_{\text{leaf}})|$ of the constituent vertices $v \in \mathcal{V}_c$ is assumed to be statically encoded in the labels $l_c[v]$ or feature vectors $x_c[v]$.

Via the decomposition and multiplicity functions that were just described, φ_{ST} is in fact an SSCE with a nontrivial, subtree-based decomposition function. Additionally each constituent's diameter is upper bounded by $2T$ (with T being the maximum number of WL iterations) which guarantees that constituents are localized within a neighborhood of bounded size. By theorem 3.7, SVMs that use the WL subtree kernel therefore have nontrivial LTA formulations with localized constituents, i.e. they can be considered to be "LTA-like" models. However, unlike fingerprint embeddings, the WL subtree constituents are not manually chosen to be interpretable. This implies that the localized explainability characteristic of LTA is only partially satisfied since the SVM predictions are based on local constituent predictions that are not necessarily interpretable.

LTA-like

4. **WL shortest path kernel:** As defined in eq. (2.7) on page 19, the i-th component of the WL shortest path embedding φ_{SP} represents a 4-tuple $(t_i, a_i, b_i, d_i) \in \{0, \ldots, T\} \times \mathcal{C} \times \mathcal{C} \times \mathbb{N}_0$. The value $\varphi_{\text{SP}}(G)[i]$ is defined as the number of vertex pairs $v_a, v_b \in \mathcal{V}_G$ that have a shortest connecting path of length d_i and the WL color combination $\chi^{(t_i)}_{G,1}(v_a) = a_i$ and $\chi^{(t_i)}_{G,1}(v_b) = b_i$.

   To determine the number of such vertex pairs via an SSCE multiplicity function $\gamma_{\varphi,t_i,a_i,b_i,d_i}$, each connected pair of vertices v_a, v_b and the shortest path between them must occur together in at least one constituent, otherwise a multiplicity function cannot compute whether v_a and v_b are in fact d_i hops apart. One simple decomposition which guarantees that all shortest paths are part of at least one constituent simply splits a given graph into its connected components. Even though such a decomposition is non-trivial since it uses at least some structural information to determine the set of constituents, the fact that any pair of connected vertices must co-occur within a single constituent means

that constituents must span arbitrarily large distances within a given graph. Depending on ones domain-specific interpretation of *localized explainability*, this restriction can be seen to be not "LTA-like". Since we do not attempt to clearly separate LTA from non-LTA methods, φ_{SP} is categorized as an in-between case here. partially LTA-like

5. **k-LWL kernel:** The LTA interpretation of the k-LWL kernel is identical to that of the WL subtree kernel. The WL color of a k-multiset of vertices s after t refinement steps is described by the joint t-neighborhood of all vertices $v \in s$, i.e. by $\Gamma_G^t(s) \coloneqq \bigcup_{v \in s} \Gamma_G^t(v)$ where $\Gamma_G^t(v) \coloneqq \Gamma_G^{t-1}(v) \cup \bigcup_{u \in \Gamma_G(v)} \Gamma_G^{t-1}(u)$ and $\Gamma_G^0(v) \coloneqq \{v\}$. Analogous to WL subtree embeddings, those t-neighborhoods form the localized constituents of the k-LWL kernel via a BFS over neighboring k-multisets. Consequently SVMs using the k-LWL kernel can be interpreted as "LTA-like" methods with nontrivial localized decompositions. LTA-like

6. **k-GWL kernel:** The only difference between the k-LWL and the k-GWL kernel is their definition of the k-multiset neighborhood. Namely the refined color of a multiset $s \subseteq \mathcal{V}_G$ in k-GWL depends on the the colors of all vertices $\mathcal{V}_G$ since all multisets $s' = s \setminus \{v\} \cup \{u\}$ with $v \in s$ and $u \in \mathcal{V}_G$ are neighbors of s. Because of those global multiset neighborhoods, all the color subtrees of a graph span the entire graph. Therefore the k-GWL embedding is an SSCE with only the trivial decomposition function $\psi_\varphi(G) = \{G\}$. Thus the LTA formulation of k-GWL SVMs described in theorem 3.7 is also trivial which means that they are not LTA models. non-LTA

This concludes our overview of LTA interpretations of SVMs that use graph embeddings/kernels. Among the described approaches, fingerprint embeddings, the WL subtree kernel and the k-LWL kernel were shown to have LTA formulations with non-trivial local decomposition functions. Approaches like `graph2vec`, the WL shortest path kernel or k-GWL on the other hand, were shown to be less compatible with LTA.

3.3 GCNNs as LTA Models

Now we will look at GCNN methods from an LTA perspective. As shown in the last section, SVMs can be interpreted as LTA methods if they use an SSCE embedding; similarly GCNNs also have LTA formulations under certain conditions. In this section those conditions and the LTA formulations of the GCNNs that satisfy them are described.

Looking back, an overview of graph convolution and graph pooling layers was given in section 2.3.3. There are many ways to combine those layers in a concrete GNN but

we focus on *global pooling GCNN architectures* that use the following three steps:

1. **Convolution:** First a stack of T convolution layers is applied to the input feature matrix $X \in \mathbb{R}^{n \times d^{(0)}}$ where each row is a feature vector $x_i \in \mathbb{R}^{d^{(0)}}$ for the vertex $v_i \in \mathcal{V}_G$ or the vertex k-multiset $s_i \subseteq \mathcal{V}_G$ in case of k-GNNs. The convolution layers produce a convolved feature matrix $Z \in \mathbb{R}^{n \times d^{(T)}}$ where each row is a convolved feature vector $z_i \in \mathbb{R}^{d^{(\text{pool})}}$.

2. **Pooling:** Then the convolved feature matrix Z is reduced to a single graph feature vector $z_G \in \mathbb{R}^{d^{(\text{pool})}}$ via a pooling layer.

3. **MLP:** Lastly the final output $y_G \in \mathcal{Y}$ is computed by applying a standard MLP to the graph feature vector z_G.

$$h_{\text{GCNN}}(G) \coloneqq \text{MLP}(Pool(Conv(G)))$$

Figure 3.3 illustrates this three step architecture for vertex neighborhood convolutions like those used by GCNs [KW17] or GINs [Xu+19]. Intuitively the convolution layers implicitly define an LTA decomposition function ψ and a local evaluation function f while the pooling layer roughly corresponds to an LTA aggregation function $\mathcal{A}$. This intuition is now formalized.

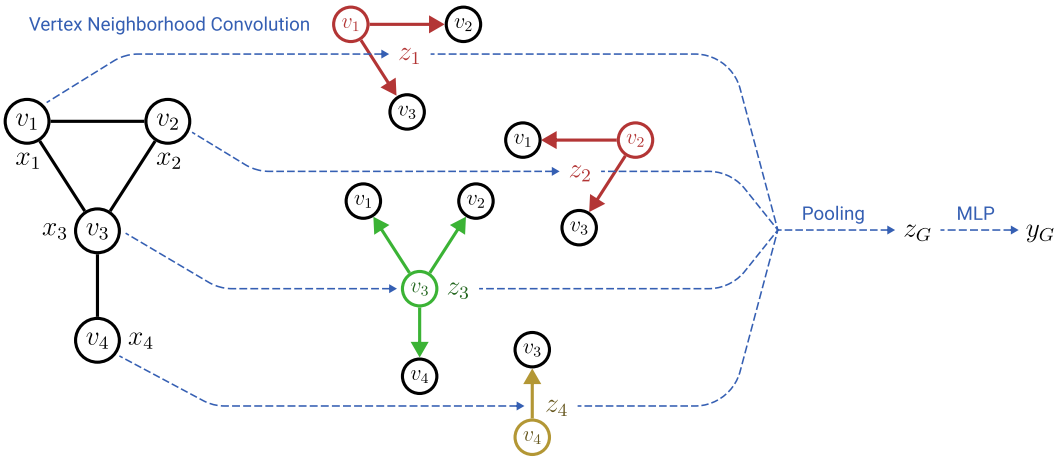


Figure 3.3. Computational steps of a GCNN that uses a single vertex neighborhood convolution layer. The colored BFS trees next to each convolved feature vector z_i show the vertices v_j whose input feature vectors x_j were used to compute z_i.

3.3.1 Graph Convolutions as Decomposition and Evaluation

To define an LTA formulation for GCNNs, we define two functions that are entirely based on the stack of convolution layers ($Conv$): 1. The decomposition function $\psi : \mathcal{G} \to \mathcal{P}(\mathcal{G})$. 2. The so-called *multi-score evaluation function* $f^* : \mathcal{G} \to (\mathcal{Y} \times \mathbb{R}_{\geq 0})^*$ which is similar to a regular LTA evaluation function f but assigns a multiset of weighted scores to each constituent instead of just a single one. Those two functions

make up the first half of a GCNN graph formulation. How exactly they are defined depends on the type convolution layer that is used.

Spectral convolutions with arbitrary filters Let us begin with general spectral graph convolution layers [Bru+13][Hen+15]. Following eq. (2.9) on page 21, after a single convolution the convolved feature vector z_i of a vertex v_i is defined as

$$z_i = \left(U^\top \hat{g}(\Lambda) U X\right)_i = \sum_{l=1}^{n} u_{l,i} \hat{g}(\lambda_l) \underbrace{\sum_{j=1}^{n} u_{l,j} x_j}_{d^{(T)}\text{-dim. vector of inner products}} = \sum_{j=1}^{n} \underbrace{\left(\sum_{l=1}^{n} \hat{g}(\lambda_l) u_{l,i} u_{l,j}\right)}_{\alpha_j} x_j \tag{3.3}$$

where $L = U^\top \Lambda U = \sum_{l=1}^{n} u_l \lambda_l u_l^\top$ is the eigendecomposition of the input graph's Laplacian and $\hat{g}$ is some (learned) eigenvalue filter.

Lemma 3.8. *For all graphs G consisting of the connected components $C_1, \ldots, C_m$ there is an $\eta \in \mathbb{R}$ s.t. if $\hat{g}(0) = \eta$, the convolved feature vector z_i of each $v_i \in \mathcal{V}_{C_l}$ with $l \in [m]$ depends on all input features x_j of the vertices $v_j \in \mathcal{V}_{C_l}$ from the same component.*

Proof Sketch. Note that by definition 2.19 on page 13 it follows that the first m eigenvalues of a graph with the connected components $C_1, \ldots, C_m$ are $\lambda_1 = \cdots = \lambda_m = 0$ with the corresponding nullspace-spanning unnormalized eigenvectors $u_l[v_j] = \mathbb{1}[v_j \in \mathcal{V}_{C_l}]$[†]. Consequently in eq. (3.3) the first m summands of each α_j sum up to $\hat{g}(0) u_{l,j} u_{l,i} = \eta$ for all $v_i \in C_l$ with $l \in [m]$. The lemma then follows by choosing η s.t. the last $n - m$ summands of every α_j do not sum up to η. □

Because a GCNN with an arbitrary learned spectral filter $\hat{g}$ can learn $\hat{g}(0) = \eta$, each convolved vertex feature vector z_i potentially depends on the entire connected component C_l with $v_i \in \mathcal{V}_{C_l}$. This means that any constituent scores computed based on z_i can only be computed for constituents that span entire connected components. In other words, spectral GCNNs with arbitrary filters generally only have LTA formulations with the decomposition function $\psi(G) = \{C_1, \ldots, C_m\}$. This restriction makes them an in-between case which is only partially LTA-like.

Vertex neighborhood convolutions Unlike spectral convolutions, spatial approaches like GCNs or GINs compute convolved feature vectors z_i which only depend on the T-neighborhood of each vertex $v_i \in \mathcal{V}_G$. Just like the BFS subtree evaluations of WL subtree kernel SVMs (see item 3, page 33), each z_i can therefore be interpreted as an evaluation of a local constituent c_i. As previously described, this means that two evaluations z_i, z_j with $i \neq j$ are computed for a single subgraph constituent $c_i = c_j$ iff. the BFS trees of v_i and v_j span the same vertices. This is illustrated in fig. 3.3 by z_1 and z_2 which are both evaluations of the constituent $c_1 = c_2 = G[\{v_1, v_2, v_3\}]$.

[†] We refer to Das [Das04] for a more in-depth discussion of graph Laplacian eigenvectors.

Vertex k-multiset neighborhood convolutions Analogously to single vertex convolutions, the k-multiset convolutions of k-GNNs produce local evaluation vectors z_i for the T-neighborhood of each k-multiset $s_i \subseteq \mathcal{V}_G$. By eq. (2.15) on page 24 the constituent c_i spanned by the vertices that have an influence on z_i can thus be written as $c_i = G\left[\bigcup_{v \in s_i} \Gamma_G^T(v)\right]$. This constituent definition is identical to that used in the LTA formulation of k-LWL SVMs (see item 5, page 35).

GCNN LTA decomposition Both, the single vertex neighborhood convolution as well as the vertex k-multiset neighborhood convolution produce vectors $z_i \in \mathbb{R}^{d^{(T)}}$, which are each based on the constituent c_i spanned by the T-neighborhood of some vertex v_i or vertex k-multiset s_i. Consequently we define the LTA decomposition function of such neighborhood convolution GCNNs as $\psi(G) \coloneqq \{c_i\}_{i=1}^n$.

GCNN LTA multi-score evaluation In addition to this decomposition function, $Conv$ also implicitly computes local constituent evaluations. If there is a translation function $\tau_1 : \mathbb{R}^{d^{(T)}} \to \mathcal{Y} \times \mathbb{R}_{\geq 0}$, the convolved vectors $z_i \in \mathbb{R}^{d^{(T)}}$ can be interpreted as constituent evaluations. One trivial example for such a translation function would be the identity $\tau_1(z) = z$ for a graph regression problem with the score domain $\mathcal{Y} = [0, 1]$ and where the last convolution layer uses a logistic activation function with two output dimensions s.t. $z \in [0, 1]^2$. Assuming that there is a translation function τ_1 we can define a multi-score evaluation function $f^*(c) \coloneqq \{\!\{\tau_1(z_i) \,|\, i \in [n] \wedge c = c_i\}\!\} = \{\!\{\tau_1(Conv(c)[v_i]) \,|\, v_i \in root(c)\}\!\}$. The function $root(c) = \{v_i \,|\, i \in [n] \wedge c = c_i\}$ returns the set of root vertices v_i (or root vertex k-multisets in case of k-GNN) whose BFS subtree of depth T in G spans exactly the vertices in c. To compute $root(c)$ given only a single constituent c and not the entire graph G, it has to be determined which BFS subtrees of c are also BFS subtrees of G. We already saw how to do this in the LTA formulation of WL subtree kernel SVMs (see $complete$ in eq. (3.2), page 34).

3.3.2 Graph Pooling as Aggregation

Based on the decomposition function $\psi : \mathcal{G} \to \mathcal{P}(\mathcal{G})$ and the multi-score evaluation function $f^* : \mathcal{G} \to (\mathcal{Y} \times \mathbb{R}_{\geq 0})^*$ which are implicitly defined by the stack of convolution layers $Conv$ of a given GCNN, we now complete our GCNN LTA formulation by defining an evaluation function $f : \mathcal{G} \to \mathcal{Y} \times \mathbb{R}_{\geq 0}$ as well as an aggregation function $\mathcal{A} : (\mathcal{Y} \times \mathbb{R}_{\geq 0})^* \to \mathcal{Y}$ based on the pooling layer $Pool : \mathbb{R}^{n \times d^{(T)}} \to \mathbb{R}^{d^{(\text{pool})}}$.

Definition 3.9. A variadic function $\xi : A^* \to B$ is called *associative* iff. there is a replacement operator $\rho : A^* \to A$ s.t. $\forall n \in \mathbb{N} : \forall (a_1, \ldots, a_n) \in A^n : \forall 1 \leq i \leq j \leq n :$ $\xi(a_1, \ldots, a_n) = \xi(a_1, \ldots, a_{i-1}, \rho(a_i, \ldots, a_j), a_{j+1}, \ldots, a_n)$. The heterogeneous algebra[3] $((A, B), (\xi, \rho))$ is called an *associative algebra* of ξ.

[3] See Birkhoff and Lipson [BL70] for a more detailed description of heterogeneous algebras.

Definition 3.10 (see Novotný [Nov02])**.** A function tuple $(\tau_1 : A \to C, \tau_2 : B \to D)$ is a *homomorphism* between two associative algebras $((A, B), (\xi, \rho))$ and $((C, D), (\xi', \rho'))$

$$\begin{aligned}\text{iff. } \forall n \in \mathbb{N} : \forall (a_1, \ldots, a_n) \in A^n : \tau_2(\xi(a_1, \ldots, a_n)) &= \xi'(\tau_1(a_1), \ldots, \tau_1(a_n)) \\ \wedge\, \tau_1(\rho(a_1, \ldots, a_n)) &= \rho'(\tau_1(a_1), \ldots, \tau_1(a_n)).\end{aligned}$$

Definition 3.11. A pooling layer $Pool : \mathbb{R}^{n \times d^{(T)}} \to \mathbb{R}^{d^{(\text{pool})}}$ is called an *associative weighted aggregation layer* iff. 1. there is an associative algebra of $Pool$ and 2. there is a weighted aggregation function $\mathcal{A} : (\mathcal{Y} \times \mathbb{R}_{\geq 0})^* \to \mathcal{Y}$ with an associative algebra and 3. there is a homomorphism $(\tau_1 : \mathbb{R}^{d^{(T)}} \to (\mathcal{Y} \times \mathbb{R}_{\geq 0}), \tau_2 : \mathbb{R}^{d^{(\text{pool})}} \to \mathcal{Y})$ from the associative algebra of $Pool$ to that of $\mathcal{A}$.

Using those definitions we can complete the connection between GCNNs and LTA:

Theorem 3.12. *A GCNN of the form $h(G) = Pool(Conv(G))$ has a nontrivial LTA formulation if $Conv$ is a stack of neighborhood convolution layers and $Pool$ is an associative weighted aggregation layer.*

Proof. We already saw that a stack of neighborhood convolutions computes the values of a multi-score evaluation function f^* for the subtree constituents returned by a decomposition function ψ. Since $Pool$ is assumed to be an associative weighted aggregation layer, there must be a replacement operator $\rho_{Pool} : \mathbb{R}^{n \times d^{(T)}} \to \mathbb{R}^{n \times d^{(T)}}$ and a homomorphism (τ_1, τ_2) to an associative algebra with the operators $\mathcal{A} : (\mathcal{Y} \times \mathbb{R}_{\geq 0})^* \to \mathcal{Y}$ and $\rho_{\mathcal{A}} : (\mathcal{Y} \times \mathbb{R}_{\geq 0})^* \to (\mathcal{Y} \times \mathbb{R}_{\geq 0})$. This allows us to rewrite the GCNN as

$$\begin{aligned}\tau_2(Pool(Conv(G))) &= \tau_2(Pool(\{\!\{ z_i \mid z_i = Conv(G)[v_i] \wedge v_i \in \mathcal{V}_G \}\!\})) \\ &= \tau_2(Pool(\{\!\{ Conv(c)[v_i] \mid c \in \psi(G) \wedge v_i \in root(c) \}\!\})) \\ &= \tau_2(Pool(\{\!\{ \rho_{Pool}(\{\!\{ Conv(c)[v_i] \mid v_i \in root(c) \}\!\}) \mid c \in \psi(G) \}\!\})) \\ &= \mathcal{A}(\{\!\{ \rho_{\mathcal{A}}(\{\!\{ \tau_1(Conv(c)[v_i]) \mid v_i \in root(c) \}\!\}) \mid c \in \psi(G) \}\!\}) \\ &= \mathcal{A}(\{\!\{ \underbrace{\rho_{\mathcal{A}}(f^*(c))}_{f(c)} \mid c \in \psi(G) \}\!\}) = \mathcal{A}(\{\!\{ f(c) \mid c \in \psi(G) \}\!\}).\end{aligned}$$

The LTA formulation above is for vertex neighborhood convolutions; a formulation for vertex k-multiset neighborhood convolutions can be obtained analogously which concludes the proof[4]. □

Note that theorem 3.12 requires GCNNs without a final MLP which is unlike the architecture proposed in the GCN [KW17] and GIN [Xu+19] papers. The reason for this is that the composition $\text{MLP} \circ Pool$ cannot be guaranteed to be homomorphic to a weighted aggregation function $\mathcal{A}$ due to the well-known universal approximation theorem [Hor91].

[4] The LTA formulation in this proof assumes that the GCNN's output is translated into the target score space via $\tau_2 : \mathbb{R}^{d^{(\text{pool})}} \to \mathcal{Y}$. All this means is that an LTA-like GCNN must produce outputs that are in $\mathcal{Y}$; for example if $Pool$ returns one-hot vector encodings of discrete classes, those must be decoded.

To conclude our analysis of the relation between LTA and GCNNs, we check which of the graph pooling layers described in section 2.3.3 satisfy definition 3.11 are therefore allowed in LTA-like GCNNs by theorem 3.12. For simplicity we assume that the target score domain is $\mathcal{Y} = [0, 1]$.

1. **Unweighted mean, minimum and maximum pooling:** If the last convolution layer has a single output dimension and an activation function with the signature $\sigma : \mathbb{R} \to [0, 1]$ (e.g. the logistic function), the simple unweighted pooling layers mean, min and max trivially satisfy definition 3.11 due to their associativity.

2. **SortPooling & SAGPooling:** Apart from such unweighted pooling layers we also looked at two state-of-the-art graph pooling layers: 1. SortPooling [Zha+18] works by aggregating the top-p convolved vectors $z_i \in \mathbb{R}^{d^{(T)}}$ based on a lexicographic ordering $\prec$, 2. SAGPooling [Lee+19] also aggregates the top-p vectors z_i but determines their ordering $\prec$ based on attention weights which are encoded as one component of each z_i. Both of those pooling approaches are not compatible with the LTA formulation from theorem 3.12 because top-p aggregation is generally not associative. This is shown by the fact that there is no $\rho : \mathbb{R}^{*\times d^{(T)}} \to \mathbb{R}^{d^{(T)}}$ s.t. for all aggregation functions $\mathcal{A}$ and vectors $z_1 \prec z_2$ with $\mathcal{A}(\{\!\{z_2, z_2\}\!\}) = z_2$ (see definition 3.2) it holds that

$$\begin{aligned} & \mathcal{A}(top_2(\{\!\{z_1, z_2, z_2\}\!\})) = \mathcal{A}(top_2(\{\!\{z_1, \rho(\{\!\{z_2, z_2\}\!\})\}\!\})) \\ \Leftrightarrow \quad & z_2 = \mathcal{A}(\{\!\{z_1, \rho(\{\!\{z_2, z_2\}\!\})\}\!\}) \qquad \not\Leftarrow \text{ if } \mathcal{A} \neq \max. \end{aligned}$$

   From the LTA perspective the non-associativity of SortPooling and SAGPooling implies that, if they were used in a GCNN, it could happen that some subtree evaluations $(y_i, w_i) \in f^*(c)$ of a single constituent c are considered by the aggregation function $\mathcal{A}$ while some are ignored depending on the subtree evaluations of other constituents, i.e. the constituent evaluation $f(c)$ could then depend on other constituents. This is forbidden in our definition of LTA.

Since neither SortPooling nor SAGPooling are compatible with LTA, we propose an associative weighted aggregation layer inspired by SAGPooling which uses a softmax-based attention filter instead of a top-p filter.

Definition 3.13. We define *softmax attention mean pooling* (SAMPooling) as

$$\mathrm{SAM}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^n) \coloneqq \frac{1}{\sum_{i=1}^n e^{w_i}} \sum_{i=1}^n e^{w_i} y_i.$$

If the stack of convolution layers of a GCNN produces vectors $z_i = (y_i, w_i) \in [0, 1] \times \mathbb{R}$, SAMPooling satisfies definition 3.11 and is therefore compatible with LTA. To see why this is the case, the associativity and homomorphicity to a weighted aggregation

function $\mathcal{A}_{\mathrm{SAM}}$ must be shown. The associativity of SAM follows directly from its commutativity and the fact that

$$\mathrm{SAM}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^{n}) = \frac{1}{e^{w'} + \sum_{i=j+1}^{n} e^{w_i}} \left(e^{w'} \underbrace{\mathrm{SAM}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^{j})}_{y'} + \sum_{i=j+1}^{n} e^{w_i} y_i \right)$$

$$\text{for all } j \in [n] \text{ with } w' \coloneqq \ln \sum_{i=1}^{j} e^{w_i}, \text{ i.e. } \rho_{\mathrm{SAM}}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^{j}) = (y', w').$$

The homomorphicity of SAM to a weighted aggregation function $\mathcal{A}_{\mathrm{SAM}}$ follows by choosing $\mathcal{A}_{\mathrm{SAM}}(\{\!\{(y_i, w_i)\}\!\}_{i=1}^{n}) \coloneqq (\sum_{i=1}^{n} w_i)^{-1} \sum_{i=1}^{n} w_i y_i$ and the trivial homomorphism $\tau_1(y, w) \coloneqq (y, e^w)$, $\tau_2(y) \coloneqq y$ between the algebras of SAM and $\mathcal{A}_{\mathrm{SAM}}$.

This concludes our analysis of the relation between LTA and GCNNs. We saw that spectral convolutions with arbitrary filters are only partially LTA-like since their constituents generally span entire connected components. GCNNs using vertex neighborhood convolutions on the other hand were shown in theorem 3.12 to have an LTA formulation with localized subtree constituents. The only requirement for this formulation to exist is that an associative pooling layer like mean, min, max or SAM is used to combine the convolved feature vectors into the final output.

Learning to Decompose Graphs

4

In the last chapter we saw how existing GC/GR approaches relate to LTA. Despite their differences, all described LTA formulations have one thing in common: Their decomposition functions ψ all split a given graph G into constituents spanned by BFS subtrees of G or simply into connected components of G. The only exception to this is the LTA formulation of SVMs that use fingerprint embeddings; there the constituents are all isomorphic to handpicked substructures.

Apart from the fingerprint embedding approach, which requires domain knowledge in order to pick meaningful substructure patterns, current GC/GR approaches use constituents that are at-best *localized* but not necessarily *interpretable*. The defining characteristic of LTA proposed in section 3.1, *localized explainability*, is therefore only partially satisfied by existing approaches. This shortcoming of the existing LTA formulations for structured input data gives rise to a new problem:

Definition 4.1. The *learning to decompose* (LTD) problem is solved by finding a graph decomposition function $\psi : \mathcal{G}_\mathcal{D} \to \mathcal{P}(\mathcal{G}_\mathcal{D})$ which splits all graphs $G \in \mathcal{G}_\mathcal{D}$ from a given domain $\mathcal{D}$ into constituent subgraphs which are individually "meaningful" in the domain $\mathcal{D}$.

As per our definition of LTA from section 3.1, the quality of an LTA formulation is determined by its chosen solution for the LTD problem. Since a comprehensive analysis of this problem would be beyond the scope of this thesis, we focus on the relation between LTD and GCNNs. The goal of this chapter is to answer the following two questions:

1. *Can a stack of graph convolution layers learn a decomposition function dynamically instead of using a static subtree decomposition?*

2. *What could be the foundation for such an "LTD-convolution" layer?*

We provide an answer to the first question in section 4.1 by showing how decomposition functions ψ can be learned via so-called *edge filters* as part of a convolutional GNN architecture. This establishes the connection between the LTD problem and graph convolutions on a high level. Then the second question is tackled in sections 4.2 and 4.3. There a novel graph convolution approach is proposed which could serve as a starting point for a convolution layer which solves the LTD problem.

4.1 Learning Constituents via Edge Filters

As we saw in section 3.3.1, the constituents of neighborhood convolutions are spanned by BFS trees of depth T, where T is the number of convolutional layers. From this perspective the problem of learning constituents corresponds to learning a pruning operator on the branches of BFS trees. Such a pruning operator filters the edges that are traversed in each BFS step. There are two general edge filtering strategies:

1. **Edge prefiltering:** Here the edge filtering and convolution operations are performed in independent steps: First the edges of a given graph are filtered, then the convolution layers are applied to the filtered graph. This so-called *edge prefiltering* strategy is illustrated in fig. 4.1. The main advantage of prefiltering is that it allows arbitrary combinations of edge filtering and convolution approaches. The main disadvantage is however that the same edges are removed in all BFS subtrees. This restricts the expressive power of the learned decomposition function as shown in fig. 4.2.

2. **Dynamic edge filtering:** By filtering edges as part of the convolution operation itself, more flexible decompositions can be obtained. In the *dynamic edge filtering* strategy the edge filter is part of the convolution operation and decides which neighbors of a given root node should be aggregated. Using this strategy a decomposition such as that shown in fig. 4.2 is learnable.

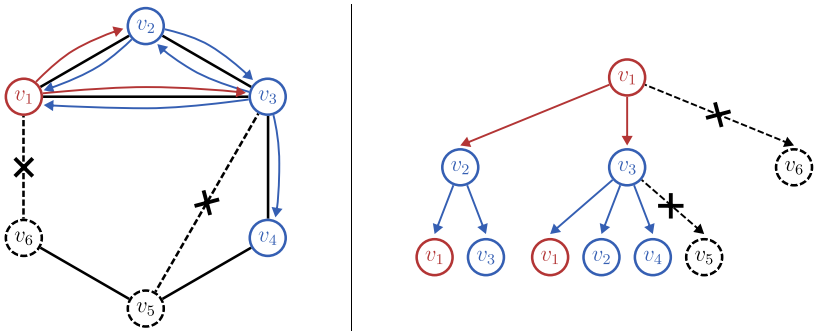


Figure 4.1. Illustration of a pruned BFS subtree when two edges are removed via prefiltering.

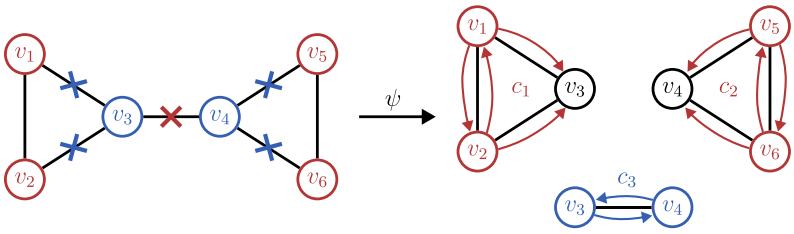


Figure 4.2. A graph decomposition obtained via dynamic edge filtering which cannot be modeled by edge prefiltering. Depending on the BFS root node v_{root} different edges are removed. For $v_{\text{root}} \in \{v_3, v_4\}$ the edges $\{(v_3, v_1), (v_3, v_2), (v_4, v_5), (v_4, v_6)\}$ are filtered out while for $v_{\text{root}} \in \{v_1, v_2, v_5, v_6\}$ the edge (v_3, v_4) is removed.

Irrespective of the chosen edge filtering strategy, the decision whether to remove a given edge $e_{ij} = (v_i, v_j)$ or not has to be made based on relevant information about e_{ij} and its surroundings. One way to encode the information about e_{ij} is via an edge feature vector $z_{ij} \in \mathbb{R}^d$ that is fed into the edge filter.

We propose that those edge feature vectors can be obtained via a 2-WL inspired graph convolution layer since 2-WL colors naturally represent the structural roles of edges as described in section 2.2.2. We already saw a GCNN architecture in section 2.3.3 which could potentially compute such 2-WL inspired edge feature vectors, the k-GNN [Mor+19]. Its variant for $k = 2$ produces edge feature vectors that can be used as the input for an edge filter. However, as we will see in the next section, there are significant limitations to the discriminative and computational power of 2-GNNs.

4.2 Limitations of the Existing 2-GNN

The k-GNN is a GCNN inspired by the k-WL algorithm, it convolves feature vectors of vertex k-multisets. In this section we will compare its $k = 2$ variant with the 2-WL algorithm. The main difference between the two boils down to their notion of "neighborhood". As already briefly mentioned in section 2.3.3, 2-GNNs define the neighbors of an edge $e_{ij} = (v_i, v_j)$ to be the edges that are incident to either v_i or v_j. In 2-WL on the other hand, the neighbors of e_{ij} are the edge pairs $\{(e_{il}, e_{lj})\}_{v_l \in \mathcal{V}_G}$, i.e. all possible paths of length two that start at v_i and end at v_j. This difference becomes clear when comparing the definition of convolution in 2-GNNs with that of color refinement in 2-WL (see eq. (2.15) on page 24 and definition 2.10 on page 9):

$$\text{2-GNN}^1\text{:}\quad Z_G^{(t)}[e_{ij}] = \sigma\left(Z_G^{(t-1)}[e_{ij}]W^{(t)} + \left(\sum_{v_l \in \Gamma_G(v_j)} Z_G^{(t-1)}[e_{il}] + \sum_{v_l \in \Gamma_G(v_i)} Z_G^{(t-1)}[e_{lj}]\right) W_\Gamma^{(t)}\right)$$

$$\text{2-WL:}\quad \chi_{G,2}^{(t)}(e_{ij}) = h\left(\chi_{G,2}^{(t-1)}(e_{ij}),\quad \{\!\{(\chi_{G,2}^{(t-1)}(e_{il}), \chi_{G,2}^{(t-1)}(e_{lj})) \mid v_l \in \mathcal{V}_G\}\!\}\right)$$

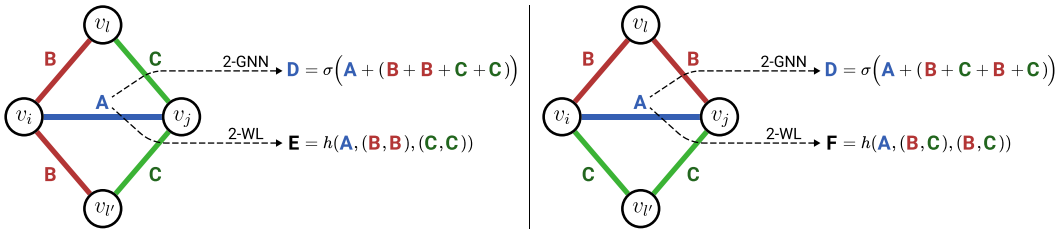


Figure 4.3. Two edge colorings on which 2-GNNs and 2-WL behave differently. A 2-GNN will refine the "color vector" of e_{ij} to **D** for both initial colorings. 2-WL on the other hand differentiates both colorings by preserving the color tuple information.

[1] To highlight the relation between 2-GNNs and 2-WL, a 2-GNN definition that is not generally correct is shown here; for self-loops with $i = j$ it incorrectly sums the feature vectors of neighboring edges twice. The correct general formula uses a single sum over $v_l \in \Gamma_G(v_j) \cup \Gamma_G(v_i)$.

We will now analyze what those different notions of neighborhood imply for the discriminative and computational power of 2-GNNs in comparison to 2-WL. In the first step we show that the discriminative power of 2-GNNs on all graphs $G \in \mathcal{G}$ is upper bounded by that of 1-WL on the so-called *edge neighborhood graphs* $G^{\mathcal{E}} \in \mathcal{G}^{\mathcal{E}}$.

Definition 4.2. The *edge neighborhood graph* of a given graph $G = (\mathcal{V}_G, \mathcal{E}_G)$ is defined as $G^{\mathcal{E}} \coloneqq (\mathcal{V}_{G^{\mathcal{E}}}, \mathcal{E}_{G^{\mathcal{E}}})$ with the vertices $\mathcal{V}_{G^{\mathcal{E}}} \coloneqq \{\{\!\{v, u\}\!\} \mid (v, u) \in \mathcal{E}_G \vee v = u\}$ and the edges $\mathcal{E}_{G^{\mathcal{E}}} \coloneqq \left\{(e, e') \in \mathcal{V}^2_{G^{\mathcal{E}}} \mid |e \cap e'| = 1\right\}$.

Proposition 4.3. *The discriminative power of all 2-GNNs h_2 is bounded by that of 1-WL on edge neighborhood graphs, i.e. $\forall\, G, H \in \mathcal{G} : G^{\mathcal{E}} \simeq_1 H^{\mathcal{E}} \rightarrow h_2(G) = h_2(H)$.*

Proof. By definition 4.2 it holds that $\forall e_{ij} \in \mathcal{E}_G : \Gamma_G(v_i) \cup \Gamma_G(v_j) = \Gamma_{G^{\mathcal{E}}}(e_{ij})$. Therefore the 2-GNN convolution operation defined in eq. (2.15) on page 24 can be rewritten as a vertex neighborhood convolution operator

$$Z_G^{(t)}[e] = \sigma\left(Z_G^{(t-1)}[e]W^{(t)} + \sum_{e' \in \Gamma_{G^{\mathcal{E}}}(e)} Z_G^{(t-1)}[e']W_\Gamma^{(t)}\right).$$

Proposition 4.3 then follows from proposition 2.23 (page 23). □

Lemma 4.4. *1-WL cannot distinguish the edge neighborhood graphs $G^{\mathcal{E}}$ and $H^{\mathcal{E}}$ of any pair of d-regular graphs G and H with n vertices.*

Proof. Let G and H be two d-regular graphs of size n. Their corresponding edge neighborhood graphs $G^{\mathcal{E}}$ and $H^{\mathcal{E}}$ both have $n^{\mathcal{E}} = n + \frac{nd}{2}$ vertices. n of those edge neighborhood vertices correspond to the vertices of G and H respectively, we will refer to them as *loop vertices* L_G/L_H. The remaining $\frac{nd}{2}$ edge neighborhood vertices correspond to the edges of G and H, we will refer to them as *edge vertices* E_G/E_H.

W.l.o.g. we define the initial colors of the loop vertices as $\chi^{(0)}(v) = \texttt{A}$ for all $v \in L_G \cup L_H$. The initial colors of the edge vertices are defined as $\chi^{(0)}(e) = \texttt{B}$ for all $e \in E_G \cup E_H$. Note that each loop vertex $\{\!\{v_i, v_i\}\!\}$ with $v_i \in \mathcal{V}_G \cup \mathcal{V}_H$ has d neighbors, the edges incident to v_i. Similarly each edge vertex $\{\!\{v_i, v_j\}\!\}$ has $2d$ neighbors, two of which are the loop vertices $\{\!\{v_i, v_i\}\!\}$ and $\{\!\{v_j, v_j\}\!\}$ with the remaining $2d-2$ neighbors corresponding to the edges that are incident to e_{ij}. This is illustrated in fig. 4.4.

After one color refinement step we get $\chi^{(1)}(v) = h(\texttt{A}, \{\!\{\underbrace{\texttt{B}, \ldots, \texttt{B}}_{d \text{ times}}\}\!\}) =: \texttt{C}$ for all loop vertices $v \in L_G \cup L_H$ and $\chi^{(1)}(e) = h(\texttt{B}, \{\!\{\texttt{A}, \texttt{A}, \underbrace{\texttt{B}, \ldots, \texttt{B}}_{2d-2 \text{ times}}\}\!\}) =: \texttt{D}$. This means that $\chi^{(0)}$ and $\chi^{(1)}$ are identical up to the color substitutions $\texttt{A} \rightarrow \texttt{C}$ and $\texttt{B} \rightarrow \texttt{D}$, i.e. $\chi^{(0)} \equiv \chi^{(1)}$, which in turn implies that 1-WL terminates after one iteration. Lemma 4.4 then directly follows since both $G^{\mathcal{E}}$ and $H^{\mathcal{E}}$ have n vertices with the final color C and $\frac{nd}{2}$ vertices with the final color D, i.e. $G^{\mathcal{E}} \simeq_1 H^{\mathcal{E}}$. □

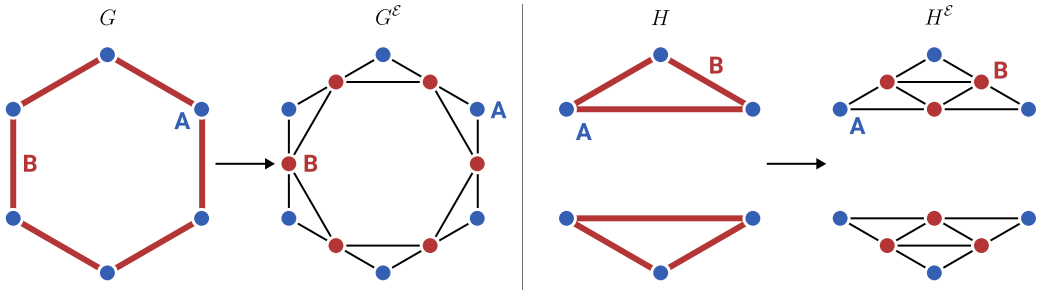


Figure 4.4. Illustration of the edge neighborhood graphs of two 2-regular graphs of size 6.

Proposition 4.5. *A 2-GNN cannot distinguish regular graphs of the same size and therefore has a lower discriminative power than 2-WL.*

Proof. The proposition directly follows from proposition 4.3, lemma 4.4 and the fact that 2-WL is able to distinguish most regular graphs [IL90, cor. 1.8.6]. □

As described in section 2.2.2 the discriminative power of a model is, by itself, not necessarily relevant for real-world graph datasets. In proposition 2.17 on page 12 we saw however that 2-WL not only has a higher discriminative power than 1-WL but also that it is able to count the number of m-cycles in a given graph for all $m \leq 7$. This is relevant because cycle counts are a commonly used metric in real-world domains such as social network and molecular structure analysis [Mil02][New03][Wel+07][AB73][Kek66]. To conclude this section we now show that 2-GNNs not only have a lower discriminative power than 2-WL but are also unable to detect cycles.

Proposition 4.6. *2-GNNs cannot detect m-cycles for all $m \geq 3$.*

Proof. Let n be the lowest common multiple of 3 and some $m > 3$. We define $c_3 := \frac{n}{3}$ and $c_m := \frac{n}{m}$. Based on that we define the following two graphs: Let G_3 be a graph consisting of c_3 disconnected cycles of length 3, analogously let G_m be a graph consisting of c_m disconnected cycles of length m. Since both G_3 and G_m are 2-regular and have the size n, any 2-GNN $h_2 : \mathcal{G} \to \mathcal{Y}$ must map both of them to the same $y \in \mathcal{Y}$ by proposition 4.5.

Let us assume that h_2 is able to detect cycles of length 3, i.e. triangles. By definition 2.16 on page 11 this would imply that there is a function $g : \mathcal{Y} \to \{0, 1\}$ s.t. $g(h_2(G_3)) = g(y) = 1$ and $g(h_2(G_m)) = g(y) = 0$ ↯. Conversely, assuming that h_2 is able to detect cycles of length $m > 3$, we obtain the contradiction $g(h_2(G_3)) = g(y) = 0$ and $g(h_2(G_m)) = g(y) = 1$ ↯. This concludes the proof. □

In this section we compared 2-GNNs with the 2-WL algorithm and found that they have a significantly lower discriminative and computational power than 2-WL. Motivated by those limitations we describe a novel graph convolution operator which is closer to 2-WL in the following section.

4.3 A Novel 2-WL Inspired GNN

We have seen that the main difference between 2-GNNs and 2-WL is their notion of "neighborhood". In this section we describe a novel convolution operator which uses edge tuple neighbors just like 2-WL to overcome the limitations of 2-GNNs. This will be done in three steps: 1. We begin by formally defining the 2-WL convolution operator. 2. Then the discriminative and computational power of this operator are analyzed. 3. Lastly we describe how 2-WL convolutions can be implemented efficiently on a modern *general purpose graphics processing unit* (GPGPU).

4.3.1 Definition of the 2-WL Convolution Operator

Similar to the 2-LWL kernel described in section 2.3.2, our 2-WL convolution operator reduces the computational cost of 2-WL via two simplifications:

1. **2-multisets:** Since we assume that graphs are undirected, e_{ij} and e_{ji} have identical feature vectors $x[e_{ij}] = x[e_{ji}] \in \mathcal{X}_{\mathcal{E}}$ and the same 2-WL neighborhood. Instead of refining/convolving the feature vectors of 2-tuples (v_i, v_j) we can therefore refine/convolve the feature vectors of 2-multisets $\{\!\{v_i, v_j\}\!\}$. This halves the number of feature vectors without affecting the discriminative or computational power of the 2-WL convolution operator. To simplify the notation we assume that $e_{ij} = e_{ji} = \{\!\{v_i, v_j\}\!\}$ in the rest of this section.

2. **Neighborhood localization:** Using the 2-multiset simplification, the original 2-WL algorithm refines the color of all multisets $e_{ij} \in \mathcal{V}_G^2$ by hashing its current color and the colors of all neighbors $\{\{\{e_{il}, e_{lj}\}\!\}\}_{v_l \in \mathcal{V}_G}$. This means that the time complexity of a single refinement step is $\mathcal{O}(n^3)$ for $n := |\mathcal{V}_G|$ which quickly becomes infeasible for large graphs.

   To address this issue we reduce both the number of colored edges as well as the number of neighbors of each edge. This is achieved by only considering the edges that are part of the so-called *r-th power of a given graph G* where $r \in \mathbb{N}$ is the freely choosable *neighborhood radius*.

   Definition 4.7. The *r-th power of a graph G* is defined as

   $$G^r := \left(\mathcal{V}_G, \left\{e_{ij} \in \mathcal{V}_G^2 \,|\, d_{\mathrm{SP},G}(v_i, v_j) \leq r\right\}\right)$$

   where $d_{\mathrm{SP},G}(v_i, v_j)$ is the length of the shortest path between v_i and v_j in G. The distance of a vertex $v_i \in \mathcal{V}_G$ to itself is defined as $d_{\mathrm{SP},G}(v_i, v_i) := 0$. Note that G^1 does not generally equal G because G^1 has self-loop edges $e_{ii} \in \mathcal{E}_{G^1}$ at all vertices.

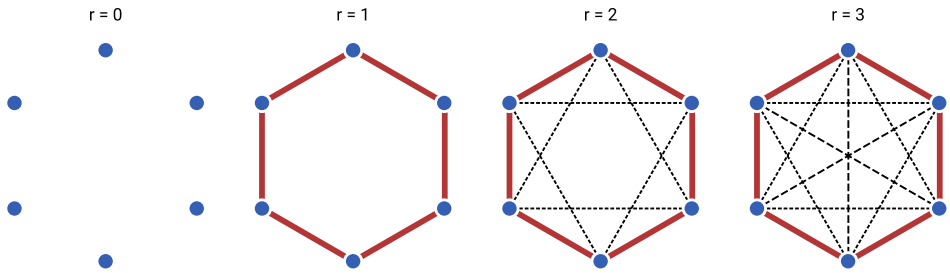


Figure 4.5. Illustration of the powers of the six-cycle graph for varying r. The self-loop edge at each of the vertices is not explicitly shown. For $r = 3$ all possible edges between the six vertices will be considered just as in the original 2-WL algorithm.

For the neighborhood radius $r = 1$ only the self-loop edges $\{e_{ii}\}_{v_i \in \mathcal{V}_G}$ and the edges $\mathcal{E}_G$ are considered; for $r > 1$ edges between indirectly connected vertices are considered as well. Figure 4.5 illustrates this for varying neighborhood radii. Through the reduction of the considered edges, the neighbors of each $e_{ij} \in \mathcal{E}_{G^r}$ are in turn reduced to the common r-neighbors of v_i and v_j, i.e. $\{\{\!\{e_{il}, e_{lj}\}\!\} \mid v_l \in \Gamma_{G^r}(v_i) \cap \Gamma_{G^r}(v_j)\}$.

Let us now consider what the reduced number of considered edges and the reduced number of edge neighbors implies for the runtime of a refinement step. If G is a sparse graph with the maximum vertex degree $d := \max_{v \in \mathcal{V}_G} |\Gamma_G(v)|$, the number of considered edges is bounded by $\mathcal{O}(nd^r)$ where each edge has at most $\mathcal{O}(d^r)$ neighbors. Consequently the time complexity of a refinement step becomes $\mathcal{O}(nd^{2r})$ which is a significant improvement over the $\mathcal{O}(n^3)$ bound of a full 2-WL refinement step (assuming $d \ll n$).

Based on the 2-multiset and the neighborhood localization simplifications we can now define the 2-WL convolution operator and the corresponding so-called *2-WL-GNN*.

Definition 4.8. The *initial feature matrix* $Z_G^{(0)}$ of the 2-WL convolution operator with the neighborhood radius $r \in \mathbb{N}$ contains both the vertex features $x[v_i] \in \mathcal{X}_\mathcal{V} = \mathbb{R}^{d_\mathcal{V}}$ as well as the edge features $x[e_{ij}] \in \mathcal{X}_\mathcal{E} = \mathbb{R}^{d_\mathcal{E}}$ of a given graph G. More specifically $Z_G^{(0)} \in \mathbb{R}^{|\mathcal{E}_{G^r}| \times (d_\mathcal{V} + d_\mathcal{E})}$ assigns a row vector $Z_G^{(0)}[e_{ij}]$ to all edges $e_{ij} \in \mathcal{E}_{G^r}$. Those initial edge feature vectors are defined by the following vector concatenation:

$$Z_G^{(0)}[e_{ij}] := \begin{pmatrix} \begin{cases} x[v_i] & \text{if } i = j \\ \mathbf{0} & \text{else} \end{cases} \end{pmatrix} \oplus \begin{pmatrix} \begin{cases} x[e_{ij}] & \text{if } e_{ij} \in \mathcal{E}_G \\ \mathbf{0} & \text{else} \end{cases} \end{pmatrix}$$

Definition 4.9. We define the *2-WL graph convolution operator* as

$$Z_G^{(t)}[e_{ij}] := \sigma\left(Z_G^{(t-1)}[e_{ij}] W_{\mathrm{L}}^{(t)} + \sum_{v_l \in \Gamma_{G^r}(v_i) \cap \Gamma_{G^r}(v_j)} \kappa^{(t)}\left(Z_G^{(t-1)}[e_{ij}], \{\!\{ Z_G^{(t-1)}[e_{il}], Z_G^{(t-1)}[e_{lj}] \}\!\} \right) \right)$$

$$\text{with } \kappa^{(t)}(z_{ij}, \{\!\{z_{il}, z_{lj}\}\!\}) := \left(z_{ij} W_{\mathrm{F}}^{(t)} \right) \odot \sigma_\Gamma\left((z_{il} + z_{lj}) W_\Gamma^{(t)} \right).$$

The convolution operator from definition 4.9 is parameterized by the three matrices $W_{\mathrm{L}}^{(t)}, W_{\mathrm{F}}^{(t)}, W_{\Gamma}^{(t)} \in \mathbb{R}^{d^{(t-1)} \times d^{(t)}}$ and uses two freely choosable activation functions σ and σ_{Γ}. There are three properties which motivate this particular choice of convolution layer:

1. **Simulation of MLPs:** By choosing $W_{\mathrm{F}}^{(t)} = \mathbf{0}$ the convolution behaves like a regular fully connected layer. A stack of 2-WL convolution layers can therefore simulate arbitrary MLPs.

2. **Perservation of edge pair information:** The 2-WL convolution layer computes a feature vector for each neighbor $\{\!\{e_{il}, e_{lj}\}\!\}$ of e_{ij} via $\sigma_{\Gamma}\left((z_{il} + z_{lj})\, W_{\Gamma}^{(t)}\right)$. A 2-GNN use a similar formulation but leaves out the inner nonlinearity σ_{Γ}; as we saw in fig. 4.3 this causes 2-GNNs to lose the edge pair information due to the commutativity and associativity of $+$. 2-WL-GNNs do not generally have this problem because $\kappa^{(t)}(\mathbf{A}, \{\!\{\mathbf{B}, \mathbf{B}\}\!\}) + \kappa^{(t)}(\mathbf{A}, \{\!\{\mathbf{C}, \mathbf{C}\}\!\}) \neq \kappa^{(t)}(\mathbf{A}, \{\!\{\mathbf{B}, \mathbf{C}\}\!\}) + \kappa^{(t)}(\mathbf{A}, \{\!\{\mathbf{B}, \mathbf{C}\}\!\})$ if σ_{Γ} is chosen to be a nonlinear activation function.

3. **Context-dependent neighborhood filtering:** Instead of filtering out all neighbors, $W_{\mathrm{F}}^{(t)}$ also allows to filter the neighborhood of e_{ij} more selectively. We can interpret $z_{ij} W_{\mathrm{F}}^{(t)} \in \mathbb{R}^{d^{(t)}}$ as a row vector of feature dimension weights. The feature dimensions of neighbors are rescaled via those weights which allows the model to aggregate different types of neighbors depending on z_{ij}.

4.3.2 Expressive Power of 2-WL-GNNs

In this section we will analyze the power of GNNs which use the 2-WL convolution operator that was just defined. Our goal is to show that such 2-WL-GNNs have a strictly larger discriminative power than 1-WL. We begin by proving that 2-WL-GNNs are at least as powerful as 1-WL.

Definition 4.10. A GNN $h_1 : \mathcal{G} \to \mathcal{Y}$ uses *weighted vertex neighborhood sums* if its convolutional layers can be described by

$$Z_G^{(t)}[v_i] = \mathrm{MLP}^{(t)}\left(w_{ii} Z_G^{(t-1)}[v_i] + \sum_{v_j \in \Gamma_G(v_i)} w_{ij} Z_G^{(t-1)}[v_l]\right).$$

Definition 4.10 includes GCNs [KW17] where the MLP only consists of a single layer and the weights are $w_{ij} = (|\Gamma_G(v_i)| + 1)^{-\frac{1}{2}} (|\Gamma_G(v_j)| + 1)^{-\frac{1}{2}}$ (see eq. (2.13), page 23). GINs [Xu+19] also trivially satisfy the definition (see eq. (2.14), page 23).

Theorem 4.11. *For each GNN h_1 that uses weighted vertex neighborhood sums, there is a 2-WL-GNN h_2 which simulates h_1, i.e. $\forall G \in \mathcal{G} : h_1(G) = h_2(G)$.*

Proof. We prove the theorem by construction. Let $G \in \mathcal{G}$ be an arbitrary input graph with $n := |\mathcal{V}_G|$ and $m := n + |\mathcal{E}_G|$. By definition h_1 is a GNN of the form $Pool_1(Conv_1(G))$ where $Conv_1$ is a stack of T weighted vertex neighborhood sum convolutions $\left\{S^{(t)} : \mathbb{R}^{n \times d^{(t-1)}} \to \mathbb{R}^{n \times d^{(t)}}\right\}_{t=1}^{T}$ with each corresponding $\mathrm{MLP}^{(t)}$ having K layers. $Pool_1$ combines the vertex feature vectors produced by $Conv_1$.

Let h_2 be a GNN of the form $Pool_2(Conv_2(G))$ where $Conv_2$ is a stack of $(2+K)T$ 2-WL convolution layers $\left\{S^{(t,k)} : \mathbb{R}^{m \times d^{(t,k-1)}} \to \mathbb{R}^{m \times d^{(t,k)}}\right\}_{(t,k) \in [T] \times [2+K]}$ with the neighborhood radius $r := 1$. Thus $Conv_2$ produces a feature vector for each of the m edges of G^1, i.e. one for each vertex and each edge of G. We denote the initial 2-WL feature matrix with $Z^{(1,0)} \in \mathbb{R}^{m \times d^{(0,0)}}$. The layers $\left\{S^{(t,2+K)}\right\}_{t=1}^{T}$ produce the feature matrices $Z^{(t,2+K)} = Z^{(t+1,0)}$ which are then fed as input into the successor layer $S^{(t+1,1)}$. Intuitively $Conv_2$ simulates each layer of $Conv_1$ via a stack of $2 + K$ 2-WL convolution layers. This is illustrated in fig. 4.6.

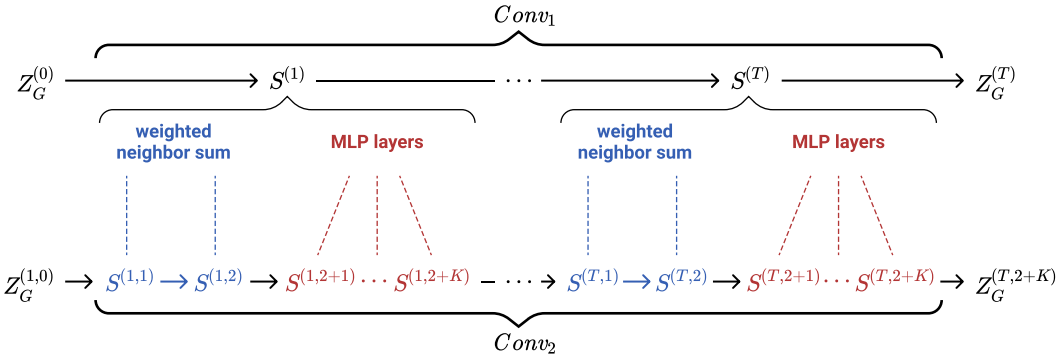


Figure 4.6. Illustration of the correspondence between $Conv_1$ and $Conv_2$.

Let $\varphi : \mathbb{R}^{d^{(T,2+K)}} \to \mathbb{R}^{d^{(T)}} \cup \{\texttt{nil}\}$ be a function which maps the final 2-WL feature vectors produced by $Conv_2$ to the output space of $Conv_1$ or the constant `nil`. Let $Pool_2\left(Z_G^{(T,2+K)}\right) := Pool_1(\{\!\{z_{ij} \mid z_{ij} = \varphi\left(Z_G^{(T,2+K)}[e_{ij}]\right) \wedge e_{ij} \in \mathcal{E}_{G^1} \wedge z_{ij} \neq \texttt{nil}\}\!\})$. Theorem 4.11 then follows if there is a function φ s.t. $\forall v_i \in \mathcal{V}_G : Conv_1(G)[v_i] = \varphi(Conv_2(G)[e_{ii}])$ and $\forall e_{ij} \in \mathcal{E}_G : \varphi(Conv_2(G)[e_{ij}]) = \texttt{nil}$. To guarantee that there is such a function φ we now inductively prove the following three invariants which have to hold for all $t \in \{0, \ldots, T\}$:

(P1) $Z_G^{(t,2+K)}[e_{ij}]_1 = \mathbb{1}[i = j]$, i.e. the first component of each 2-WL feature vector allows φ to decide whether that vector should be mapped to `nil`.

(P2) $Z_G^{(t,2+K)}[e_{ii}]_{2,\ldots,(d^{(t)}+1)} = Z_G^{(t)}[v_i]$, i.e. the second to $(1 + d^{(t)})$-th components of each self-loop feature vector in h_2 contain the corresponding convolved vertex feature vector at layer t in h_1.

(P3) $Z_G^{(t,2+K)}[e_{ij}]_{d^{(t)}+2} = w_{ij}$, i.e. the weights for the vertex neighborhood sums are encoded in the edge and self-loop feature vectors.

For $t = 0$ all three invariants hold by definition 4.8:

$$\forall v_i \in \mathcal{V}_G : Z_G^{(1,0)}[e_{ii}] \coloneqq (1) \oplus x[v_i] \oplus (w_{ii}) \text{ and } \forall e_{ij} \in \mathcal{E}_G : Z_G^{(1,0)}[e_{ij}] \coloneqq (0) \oplus \mathbf{0} \oplus (w_{ij}).$$

Assuming the invariants hold for $t-1$ we now show that they also hold for t. As illustrated in fig. 4.6 the layers $S^{(t,1)}$ and $S^{(t,2)}$ should compute the weighted vertex neighborhood sums

$$Z^{(t,2)}[e_{ii}]_{2,\dots,(1+d^{(t-1)})} = w_{ii} Z^{(t,0)}[e_{ii}]_{2,\dots,(1+d^{(t-1)})} + \sum_{v_j \in \Gamma_G(v_i)} w_{ij} Z^{(t,0)}[e_{jj}]_{2,\dots,(1+d^{(t-1)})}.$$

We now explicitly define parameter matrices for $S^{(t,1)}$ and $S^{(t,2)}$ s.t. this weighted sum is produced. Note that the weighted vertex neighborhood sum only requires scalar multiplication and vector addition, i.e. the $d^{(t-1)}$ vertex feature dimensions are mutually independent. W.l.o.g. this allows us to simplify notation by treating the vertex feature vectors as if they were scalars in the following definitions, i.e. we can assume $d^{(t-1)} = 1$ and $Z^{(t,0)}[e_{ii}] = (1, Z^{(t-1)}[v_i], w_{ii}) \in \mathbb{R}^3$. The layer $S^{(t,1)}$ is defined by

$$\begin{aligned}
Z^{(t,1)}[e_{ij}] &= Z^{(t,0)}[e_{ij}] W_{\mathrm{L}}^{(t,1)} + \sum_{v_l \in \Gamma_{G^1}(v_i) \cap \Gamma_{G^1}(v_j)} \left(Z^{(t,0)}[e_{ij}] W_{\mathrm{F}}^{(t,1)} \right) \odot \left(\left(Z^{(t,0)}[e_{il}] + Z^{(t,0)}[e_{lj}] \right) W_{\Gamma}^{(t,1)} \right) \\
&= \begin{cases} (1, 0, w_{ii}, 0) + \left(0, 0, 0, 2 w_{ii} Z^{(t-1)}[v_i]\right) + \sum_{v_l \in \Gamma_{G^1}(v_i)} \left(0, w_{il} Z^{(t-1)}[v_i], 0, 0\right) & \text{if } i = j \\ (0, 0, w_{ij}, 0) + \left(0, 0, 0, w_{ij} Z^{(t-1)}[v_i]\right) + \left(0, 0, 0, w_{ij} Z^{(t-1)}[v_j]\right) & \text{else} \end{cases} \\
&= \begin{cases} \left(1, \sum_{v_l \in \Gamma_{G^1}(v_i)} w_{il} Z^{(t-1)}[v_i], w_{ii}, 2 w_{ii} Z^{(t-1)}[v_i]\right) & \text{if } i = j \\ \left(0, 0, w_{ij}, w_{ij}(Z^{(t-1)}[v_i] + Z^{(t-1)}[v_j])\right) & \text{else} \end{cases}
\end{aligned}$$

$$\text{with } W_{\mathrm{L}}^{(t,1)} \coloneqq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, W_{\mathrm{F}}^{(t,1)} \coloneqq \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, W_{\Gamma}^{(t,1)} \coloneqq \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

The vertex neighborhood summation is completed via $S^{(t,2)}$ which is defined by

$$\begin{aligned}
Z^{(t,2)}[e_{ij}] &= \begin{cases} \left(1, -\sum_{v_l \in \Gamma_{G^1}(v_i)} w_{il} Z^{(t-1)}[v_i], w_{ii}\right) + \sum_{v_l \in \Gamma_{G^1}(v_i)} \left(0, w_{il} \left(Z^{(t-1)}[v_i] + Z^{(t-1)}[v_l]\right), 0\right) & \text{if } i = j \\ (0, 0, w_{ij}) & \text{else} \end{cases} \\
&= \begin{cases} \left(1, w_{ii} Z^{(t-1)}[v_i] + \sum_{v_l \in \Gamma_G(v_i)} w_{il} Z^{(t-1)}[v_l], w_{ii}\right) & \text{if } i = j \\ (0, 0, w_{ij}) & \text{else} \end{cases}
\end{aligned}$$

$$\text{with } W_{\mathrm{L}}^{(t,2)} \coloneqq \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \quad W_{\mathrm{F}}^{(t,2)} \coloneqq \begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad W_{\Gamma}^{(t,2)} \coloneqq \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Using the two layers $S^{(t,1)}$ and $S^{(t,2)}$ that we just defined, the weighted vertex neighborhood sum for all $v_i \in \mathcal{V}_G$ is contained in $Z^{(t,2)}[e_{ii}]$. Additionally for all $e_{ij} \in \mathcal{E}_{G^1}$ the indicators $Z^{(t,2)}[e_{ij}]_1 = \mathbb{1}[i = j]$ and the weights $Z^{(t,2)}[e_{ij}]_{d^{(t)}+2} = w_{ij}$ are preserved. This means that invariants (P1) and (P3) are satisfied after $S^{(t,2)}$. To complete the induction step it now remains to show that all three invariants hold after applying the layers $S^{(t,2+1)}, \ldots, S^{(t,2+K)}$. As previously mentioned, a 2-WL convolution layer is reduced to a fully connected layer if $W_{\mathrm{F}}^{(t)} = \mathbf{0}$. Via the universal approximation theorem [Hor91] we can therefore use $S^{(t,2+1)}, \ldots, S^{(t,2+K)}$ to simulate the K layers of $\mathrm{MLP}^{(t)}$ without changing the first and last dimension of each feature vector to preserve invariants (P1) and (P3). The resulting feature matrix $Z^{(t,2+K)}$ then satisfies all three invariants which completes the induction.

Using invariants (P1) and (P2) for $t = T$ we can therefore set

$$\varphi\left(Z_G^{(T,2+K)}[e_{ij}]\right) \coloneqq \begin{cases} Z_G^{(T,2+K)}[e_{ij}]_{2,\ldots,(d^T+1)} & \text{if } Z_G^{(T,2+K)}[e_{ij}]_1 = 1 \\ \texttt{nil} & \text{else} \end{cases}.$$

By our previous definition of $Pool_2$, this in turn implies that $Pool_2(Z_G^{(T,2+K)}) = Pool_1(Z_G^{(T)}) \iff h_2(G) = h_1(G)$ which concludes the proof. □

Corollary 4.12. *2-WL-GNNs have at least the same discriminative power as 1-WL.*

Proof. The corollary directly follows from the fact that 2-WL-GNNs can simulate GINs by theorem 4.11 and the fact that GINs have the same discriminative power as 1-WL by proposition 2.23 (see page 23) because they use injective vertex neighborhood hashing functions [Xu+19]. □

To complete our analysis of the expressive power of 2-WL-GNNs we now show that they are not just as powerful as 1-WL but in fact more powerful than 1-WL.

Proposition 4.13. *There are d-regular graphs G and H of size n which can be distinguished by 2-WL-GNNs.*

Proof. The proposition follows if we choose the six-cycle graph for G and the two three-cycles graph for H, both of which should be well-known at this point. Let $h_2 = Pool \circ S$ be a 2-WL-GNN with the neighborhood radius $r = 1$ which consists of a single 2-WL convolution layer $S : \mathbb{R}^{*\times 2} \to \mathbb{R}^{*\times 1}$ and the pooling layer $Pool = \min$. In accordance with definition 4.8 we set the initial feature vectors of the vertices v_i of G and H to $Z^{(0)}[e_{ii}] \coloneqq (1, 0)$ and the initial feature vectors of their edges e_{ij} to

$Z^{(0)}[e_{ij}] \coloneqq (0,1)$. Let the weight matrices of S be $W_{\mathrm{L}} \coloneqq \mathbf{0}$ and $W_{\mathrm{F}} = W_{\Gamma} \coloneqq \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. For simplicity we choose the identity activation functions $\sigma = \sigma_{\Gamma} = \mathrm{id}$.

By definition 4.9 all self-loops e_{ii} of G^1 and H^1 have the three neighbors $\{\{\!\{e_{ii}, e_{ii}\}\!\}, \{\{\!\{e_{ij}, e_{ji}\}\!\}, \{\{\!\{e_{il}, e_{li}\}\!\}\}$, i.e. the length-two walk along e_{ii} itself and the length-two walks to and from the two neighboring vertices $\Gamma(v_i) = \{v_j, v_l\}$. Therefore the convolved feature vector of all self-loops are $Z^{(1)}[e_{ii}] = (1) \odot ((1+1) + (1+1) + (1+1)) = 6$. However for the non-self-loops of G^1 and H^1, i.e. the edges of G and H, we get differing convolved feature vectors. The 2-WL neighbors of $e_{ij} \in \mathcal{E}_G$ are $\{\{\!\{e_{ii}, e_{ij}\}\!\}, \{\{\!\{e_{ij}, e_{jj}\}\!\}\}$. The 2-WL neighbors of $e'_{ij} \in \mathcal{E}_H$ are $\{\{\!\{e'_{ii}, e'_{ij}\}\!\}, \{\{\!\{e'_{ij}, e'_{jj}\}\!\}, \{\{\!\{e'_{il}, e'_{lj}\}\!\}\}$ where $v'_l \in \mathcal{V}_H$ is the common neighbor of v'_i and v'_j. The different neighborhood sizes of the edges of G and H imply that $\forall e_{ij} \in \mathcal{E}_G : Z^{(1)}[e_{ij}] = 4$ while $\forall e'_{ij} \in \mathcal{E}_H : Z^{(1)}[e_{ij}] = 6$. Thus $h_2(G) = \min\{4,6\} \neq \min\{6,6\} = h_2(H)$ which concludes the proof. □

Corollary 4.14. *The discriminative power of 2-WL-GNNs is strictly greater than that of the 1-WL algorithm.*

Proof. The corollary directly follows from corollary 4.12 and proposition 4.13 since 1-WL cannot distinguish regular graphs [IL90, cor. 1.8.5]. □

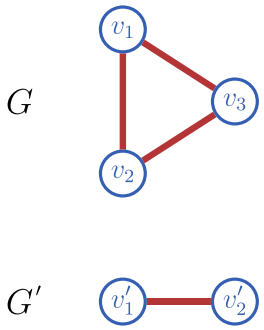
This concludes our analysis of the expressive power of 2-WL-GNNs. The key takeaway from this section is that 2-WL-GNNs are more powerful than all vertex neighborhood aggregation GCNNs because the power of the latter is upper bounded by 1-WL. Additionally we can conclude that 2-WL-GNNs are in fact also more powerful than 2-GNNs due to proposition 4.5.

Note that no statement regarding the discriminative or computational power of 2-WL-GNNs compared to 2-WL was made. It is easy to see that 2-WL-GNNs *generally* cannot have the same power as 2-WL due to the neighborhood localization simplification; e.g. for a small neighborhood radius of $r = 1$, nonexistent edges $e_{ij} \notin \mathcal{E}_G$ do not have a feature vector which implies that the proof of 2-WL's m-cycle counting ability no longer holds for $m \geq 4$ (see proposition 2.17, page 12). We leave a more thorough discussion of the relation between 2-WL-GNNs and 2-WL for future work.

4.3.3 Implementation of 2-WL-GNNs on GPGPUs

Apart from the theoretical expressive power of a model, it also has to be computable efficiently in order to be useful in practice. In this section we will therefore describe how 2-WL convolutions can be implemented on *general purpose graphics processing unit*s (GPGPUs).

Efficient high-level software libraries for the implementation of vertex neighborhood convolution approaches such as GCN or GIN already exist. They describe



idx.	edge	$Z^{(0)}$
1	e_{11}	$(1,0)$
2	e_{22}	$(1,0)$
3	e_{33}	$(1,0)$
4	e_{12}	$(0,1)$
5	e_{13}	$(0,1)$
6	e_{23}	$(0,1)$
7	e'_{11}	$(1,0)$
8	e'_{22}	$(1,0)$
9	e'_{12}	$(0,1)$

R_{L}	$R_{\Gamma,1}$	$R_{\Gamma,2}$
(1,	1,	1)
(1,	4,	4)
(1,	5,	5)
(2,	2,	2)
(2,	4,	4)
(2,	6,	6)
(3,	3,	3)
(3,	5,	5)
(3,	6,	6)
(4,	1,	4)
(4,	4,	2)
(4,	5,	6)
(5,	1,	5)
(5,	5,	3)
(5,	4,	6)
(6,	2,	6)
(6,	6,	3)
(6,	4,	5)
(7,	7,	7)
(7,	9,	9)
(8,	8,	8)
(8,	9,	9)
(9,	7,	9)
(9,	9,	8)

Figure 4.7. Exemplary 2-WL encoding of a batch of two small graphs.

convolutions via a message-passing abstraction in which vertex feature vectors are passed along their neighboring edges (see Battaglia et al. [Bat+18]). A few notable implementations of this abstraction are the *Graph Nets* library [GN], *PyTorch Geometric* [FL19][PyG], *Deep Graph Library* [Wan+19][DGL] and *Spektral* [Spe]. Since a message-passing model along edges is incompatible with the edge-pair neighborhoods of 2-WL, a custom convolution implementation is required for 2-WL-GNNs.

For this purpose we propose a sparse 2-WL graph representation which is inspired by the coordinate list adjacency format described by Fey and Lenssen [FL19]. Given a neighborhood radius r we encode a graph G using the following two matrices:

1. $Z_G^{(0)} \in \mathbb{R}^{m \times d^{(0)}}$: The initial feature matrix is represented directly as a dense floating point matrix with $m \coloneqq |\mathcal{E}_{G^r}|$ rows, each of which encodes the feature vector of an edge $e_{ij} \in \mathcal{E}_{G^r}$. Edge feature duplicates are prevented by only encoding edges with $i \leq j$ for some arbitrary vertex ordering of G.

2. $R_G \in [m]^{\gamma \times 3}$: The reference matrix R_G encodes the edge neighborhood information. It consists of $\gamma \coloneqq \sum_{e_{ij} \in \mathcal{E}_{G^r}} |\Gamma_{G^r}(v_i) \cap \Gamma_{G^r}(v_j)|$ rows, one for each 2-WL neighbor $\{\!\{e_{il}, e_{lj}\}\!\}$ of each edge e_{ij}. Each neighbor row is a vector $(r_{\mathrm{L}}, r_{\Gamma,1}, r_{\Gamma,2}) \in [m]^3$ of three index pointers to rows in $Z_G^{(0)}$. r_{L} points to the row index of the feature vector of e_{ij} while $r_{\Gamma,1}$ and $r_{\Gamma,2}$ point to the indices of e_{il} and e_{lj} respectively. We will refer to the three column vectors of R_G as $R_{G,\mathrm{L}}$, $R_{G,\Gamma,1}$ and $R_{G,\Gamma,2}$.

This encoding can also be used to represent graph batches by simply concatenating the rows of each graph's feature and reference matrices while shifting the index pointers to account for the concatenation offsets. Figure 4.7 illustrates how such a batch encoding might look like. After encoding a graph dataset as 2-WL matrices, convolutions can be computed efficiently on GPGPUs via the common *gather-scatter*

pattern from parallel programming [He+07]. The so-called $gather$ operator takes two inputs: A list Z of m row vectors and a list R of γ pointers into Z. It returns a list X of γ row vectors $X[i] = Z[R[i]]$ for $i \in [\gamma]$. The $scatter_\Sigma$ operator can be understood as the opposite of $gather$. $scatter_\Sigma$ takes a list X of γ row vectors and a list R of γ pointers from the range $[m]$. It returns a list Z of m row vectors $Z[i] = \sum_{j \in [\gamma] \wedge R[j]=i} X[j]$ for $i \in [m]$.

Using the $gather$ and $scatter_\Sigma$ operator, the 2-WL convolution operator from definition 4.9 can be computed via the following algorithm:

Algorithm 1 Parallel Implementation of a 2-WL Convolution Layer $S^{(t)}$

function $S^{(t)}(Z^{(t-1)} \in \mathbb{R}^{m \times d^{(t-1)}}, R \in [m]^{\gamma \times 3})$
  $Z_{\mathrm{L}} \coloneqq Z^{(t-1)} W_{\mathrm{L}}^{(t)}$ ▷ Matrix multiply: $\mathbb{R}^{m \times d^{(t-1)}} \to \mathbb{R}^{m \times d^{(t)}}$
  $Z_{\mathrm{F}} \coloneqq Z^{(t-1)} W_{\mathrm{F}}^{(t)}$
  $Z_{\Gamma} \coloneqq Z^{(t-1)} W_{\Gamma}^{(t)}$
  $X_{\Gamma,1} \coloneqq gather(Z_\Gamma, R_{\Gamma,1})$ ▷ Gather: $\mathbb{R}^{m \times d^{(t)}} \times [m]^{\gamma} \to \mathbb{R}^{\gamma \times d^{(t)}}$
  $X_{\Gamma,2} \coloneqq gather(Z_\Gamma, R_{\Gamma,2})$
  $X_\Gamma \coloneqq \sigma_\Gamma\left(X_{\Gamma,1} + X_{\Gamma,2}\right)$ ▷ Element-wise operations
  $Z_{\Sigma\Gamma} \coloneqq scatter_\Sigma(X_\Gamma, R_{\mathrm{L}})$ ▷ Scatter: $\mathbb{R}^{\gamma \times d^{(t)}} \times [m]^{\gamma} \to \mathbb{R}^{m \times d^{(t)}}$
  $Z^{(t)} \coloneqq \sigma\left(Z_{\mathrm{L}} + Z_{\mathrm{F}} \odot Z_{\Sigma\Gamma}\right)$ ▷ Element-wise operations
  return $Z^{(t)}$

All operations in this implementation scale well on parallel computing hardware, they are differentiable and are supported by common ML libraries like *TensorFlow* [Aba+15][🔗TF] and *PyTorch* [Pas+19][🔗PyT]. 2-WL convolutions can therefore be easily integrated into existing ML tooling and be optimized via well-known gradient-based methods like *Adam* [KB15]. This concludes our description of 2-WL-GNNs.

5 Evaluation

In chapter 3 the relation between LTA and existing GC/GR approaches was formally analyzed. There we saw that the LTA formulations of existing approaches mostly use static decomposition functions, e.g. BFS subtree decompositions. Motivated by the idea of dynamically learning decompositions via edge filters, we then proposed the novel 2-WL-GNN in chapter 4. The ides presented in both chapters will now be empirically evaluated. Do to so we differentiate between two mostly independent evaluation aspects:

1. **Evaluation of 2-WL-GNNs:** Even though it was motivated by LTA, a 2-WL-GNN is not generally more "LTA-like" than other approaches. Nonetheless, due to the theoretical advantages described in section 4.3.2 it is an interesting approach independently from its potential applications in LTA (see section 4.1). Thus the first aspect of our evaluation is to compare 2-WL-GNNs with the other previously described GC/GR methods in a general non-LTA fashion, i.e. with an added MLP after the pooling layer since this is how GNNs are typically evaluated in other works.

2. **Evaluation of the LTA assumption:** We previously described that a given domain problem satisfies the LTA assumption if its solutions can be described by an LTA formulation (see definition 3.4, page 30). The inherent bias of an LTA-like model towards such LTA formulations could potentially increase its generalization performance compared to more general non-LTA models. Therefore the second aspect of our evaluation is to compare the performance of the previously described LTA-like methods with that of non-LTA approaches on datasets from multiple problem domains.

This chapter will tackle those two aspects in three steps: ① We begin by describing the experimental setup used to obtain the evaluation results in section 5.1. ② We then present results on synthetically generated data in section 5.2. There we will illustrate the higher expressive power of 2-WL-GNNs when compared to other GC/GR approaches which confirms the theoretical results from section 4.3.2. ③ Finally the evaluation results on real-world datasets are described in section 5.3. There we will see how 2-WL-GNNs compare to other GNNs in practice as well as how LTA-like models compare to non-LTA models.

5.1 Experimental Setup

In our experimental evaluation we focus on two types of learners: SVMs using graph kernels and GCNNs. We evaluate those learners by comparing their test accuracies on multiple binary classification problems. To obtain those accuracies we follow the graph classification benchmarking framework recently proposed by Errica et al. [Err+20]. Their benchmarking framework is motivated by the observation that most recent publications in the field of GNNs do not provide reproducible results. To tackle this issue they evaluated multiple state-of-the-art methods using a unified model selection procedure:

Algorithm 2 k-fold Model Assessment

Input: Dataset $\mathcal{D}$, configurations Θ
Split $\mathcal{D}$ into k folds $F_1, \ldots, F_k$
for $i \leftarrow 1, \ldots, k$ **do**
  $\mathcal{D}_{\text{train/val}}, \mathcal{D}_{\text{test}} \leftarrow \left(\bigcup_{j \neq i} F_j\right), F_i$
  $\theta_{\text{best}} \leftarrow \textsc{Select}(\mathcal{D}_{\text{train/val}}, \Theta)$
  for $r \leftarrow 1, \ldots, R$ **do**
    $h_{i,r} \leftarrow \textsc{Train}(\mathcal{D}_{\text{train/val}}, \theta_{\text{best}})$
    $acc_{i,r} \leftarrow \textsc{Eval}(h_{i,r}, \mathcal{D}_{\text{test}})$
  $acc_i \leftarrow \text{mean}_{r \in [R]}\ acc_{i,r}$
return $\text{mean}_{i \in [k]}\ acc_i, \text{stddev}_{i \in [k]}\ acc_i$

Algorithm 3 Model Selection

function $\textsc{Select}(\mathcal{D}, \Theta)$
  Split $\mathcal{D}$ into $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}$
  for all $\theta \in \Theta$ **do**
    $h_\theta \leftarrow \textsc{Train}(\mathcal{D}_{\text{train}}, \theta)$
    $acc_\theta \leftarrow \textsc{Eval}(h_\theta, \mathcal{D}_{\text{val}})$
  $\theta_{\text{best}} \leftarrow \arg\max_{\theta \in \Theta}\ acc_\theta$
  return θ_{best}

We use this assessment strategy with $k = 10$ folds and $r = 3$ repeats per fold to smooth out differences caused by random weight initializations. For each dataset the same folds are used across the evaluated models; class proportions are preserved within each fold by using stratified splits. To keep the total runtime of the experiments feasible, a single 90%/10% holdout split into training and validation data is used instead of cross-validation. In each experiment training is performed with an early stopping condition which cancels the optimization if there is no improvement to the validation loss for more than p epochs. The patience period p is part of the hyperparameter configurations $\theta \in \Theta$.

Using this assessment strategy we evaluate SVMs with the following graph kernels:

1. **WL subtree kernel (WL_{ST})** with the iteration counts $T \in \{1, 2, 5\}$ to evaluate the influence of the depth of BFS subtrees which span LTA constituents.
2. **WL shortest path kernel (WL_{SP})** with the iteration count $T = 5$.
3. **2-LWL kernel** with the iteration count $T = 3$.
4. **2-GWL kernel** with the iteration count $T = 3$.

The gram matrices of the WL_{ST} and WL_{SP} kernels are computed via the *GraKeL* library [Sig+18][🔗GK]. For the gram matrices of the two dimensional WL kernels

we use a modified version[1] of the reference implementation provided by Morris et al. [Mor+17]. To train SVMs with those kernels *Scikit-Learn* [Ped+11][🔗SKL] is used.

For the evaluation of GCNNs we selected the following methods:

1. **Structure unaware baseline:** Errica et al. [Err+20] describe a simple model which simply applies a standard MLP to each individual vertex feature vector, then sums up the resulting feature vectors and applies another MLP to the vector sum. This approach does not use any structural information and therefore serves as a baseline to detect whether a GNN is able to exploit graph structure.
2. **GIN** is evaluated as described by Xu et al. [Xu+19], i.e. with a sum pooling layer and an appended MLP to produce the final prediction.
3. **2-GNN** is evaluated with both a static mean pooling layer and with SAMPooling (see definition 3.13, page 40). After the pooling layer a MLP is used to produce the final prediction.
4. **2-WL-GNN** *(our method)* is evaluated using the same configurations as 2-GNN. To test the LTA assumption we additionally evaluate it in LTA-like configurations, i.e. with a stack of 2-WL convolutions that produce a local prediction $y_{ij} \in [0, 1]$ for each edge e_{ij} and without the final MLP.

The baseline and GIN results are obtained using the PyTorch-based implementation provided by Errica et al. [Err+20]. For both 2-GNN and 2-WL-GNN a custom TensorFlow-based implementation is used. The code for all conducted experiments as well as the used dataset splits can be found on GitHub[2].

5.2 Evaluation on Synthetic Data

5.3 Evaluation on Real-World Data

[1] `https://github.com/Cortys/glocalwl`
[2] `https://github.com/Cortys/master-thesis`

6 Conclusion

6.1 Review

6.2 Future Directions

Appendix

A

Bibliography

[AB73] George W. Adamson and Judith A. Bush. „A method for the automatic classification of chemical structures“. In: *Information Storage and Retrieval* 9.10 (1973), pp. 561–568 (cit. on pp. 12, 16, 47).

[Aba+15] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 56).

[Abe18] Armita Abedijaberi. „Mining and Analysis of Real-world Graphs“. PhD thesis. Missouri University of Science and Technology, 2018 (cit. on p. 11).

[Alz+10] Afredo Alzaga, Rodrigo Iglesias, and Ricardo Pignol. „Spectra of symmetric powers of graphs and the Weisfeiler–Lehman refinements“. In: *Journal of Combinatorial Theory, Series B* 100.6 (2010), pp. 671–682. arXiv: `0801.2322v1 [math.SP]` (cit. on p. 14).

[Arv+19] Vikraman Arvind, Frank Fuhlbrück, Johannes Köbler, and Oleg Verbitsky. „On Weisfeiler-Leman Invariance: Subgraph Counts and Related Graph Properties“. In: *Fundamentals of Computation Theory*. Springer International Publishing, 2019, pp. 111–125. arXiv: `1811.04801v3 [cs.DM]` (cit. on p. 12).

[Bab+80] László Babai, Paul Erdős, and Stanley M. Selkow. „Random graph isomorphism“. In: *SIaM Journal on computing* 9.3 (1980), pp. 628–635 (cit. on pp. 7, 11).

[Bab15] László Babai. *Graph Isomorphism in Quasipolynomial Time*. Dec. 11, 2015. arXiv: `1512.03547v2 [cs.DS]` (cit. on p. 7).

[Ban+18] Priyanka Banerjee, Andreas O. Eckert, Anna K. Schrey, and Robert Preissner. „ProTox-II: a webserver for the prediction of toxicity of chemicals“. In: *Nucleic Acids Research* 46.W1 (2018), W257–W263 (cit. on p. 16).

[Bat+18] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, et al. *Relational inductive biases, deep learning, and graph networks*. June 4, 2018. arXiv: `1806.01261v3 [cs.LG]` (cit. on p. 55).

[BK05] K.M. Borgwardt and H. Kriegel. „Shortest-Path Kernels on Graphs“. In: *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 2005 (cit. on p. 19).

[BL70] Garrett Birkhoff and John D. Lipson. „Heterogeneous algebras“. In: *Journal of Combinatorial Theory* 8.1 (1970), pp. 115–133 (cit. on p. 38).

[Bru+13] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. *Spectral Networks and Locally Connected Networks on Graphs*. Dec. 21, 2013. arXiv: `1312.6203v3 [cs.LG]` (cit. on pp. 21, 37).

[Cai+92] Jin-Yi Cai, Martin Fürer, and Neil Immerman. „An optimal lower bound on the number of variables for graph identification“. In: *Combinatorica* 12.4 (1992), pp. 389–410 (cit. on p. 7).

[Chu10] Fan Chung. „Graph theory in the information age“. In: *Notices of the American Mathematical Society* 57 (June 2010) (cit. on p. 20).

[Das04] K.Ch. Das. „The Laplacian spectrum of a graph“. In: *Computers & Mathematics with Applications* 48.5-6 (2004), pp. 715–724 (cit. on pp. 14, 37).

[Def+16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. „Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering“. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 3844–3852. arXiv: 1606.09375v3 [cs.LG] (cit. on p. 22).

[Drw+14] Malgorzata N. Drwal, Priyanka Banerjee, Mathias Dunkel, Martin R. Wettig, and Robert Preissner. „ProTox: a web server for the in silico prediction of rodent oral toxicity“. In: *Nucleic Acids Research* 42.W1 (2014), W53–W58 (cit. on p. 16).

[Err+20] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. „A Fair Comparison of Graph Neural Networks for Graph Classification“. In: *International Conference on Learning Representations*. ICLR'2020. 2020. arXiv: 1912.09893v2 [cs.LG] (cit. on pp. 24, 58, 59).

[FL19] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. Mar. 6, 2019. arXiv: 1903.02428v3 [cs.LG] (cit. on p. 55).

[Fü17] Martin Fürer. „On the Combinatorial Power of the Weisfeiler-Lehman Algorithm“. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2017, pp. 260–271. arXiv: 1704.01023v1 [cs.DS] (cit. on p. 12).

[Gil+18] L. H. Gilpin, D. Bau, B. Z. Yuan, et al. „Explaining Explanations: An Overview of Interpretability of Machine Learning“. In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. 2018, pp. 80–89. arXiv: 1806.00069v3 [cs.AI] (cit. on p. 30).

[GL16] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. July 3, 2016. arXiv: 1607.00653v1 [cs.SI] (cit. on p. 17).

[Gor+05] M. Gori, G. Monfardini, and F. Scarselli. „A new model for learning in graph domains“. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* IEEE, 2005 (cit. on p. 20).

[Gu+15] Jiao Gu, Bobo Hua, and Shiping Liu. „Spectral distances on graphs“. In: *Discrete Applied Mathematics* 190-191 (2015), pp. 56–74. arXiv: 1402.6041v2 [math.SP] (cit. on p. 14).

[He+07] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. „Efficient gather and scatter operations on graphics processors“. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*. ACM Press, 2007 (cit. on p. 56).

[Hen+15] Mikael Henaff, Joan Bruna, and Yann LeCun. *Deep Convolutional Networks on Graph-Structured Data*. June 16, 2015. arXiv: 1506.05163v1 [cs.LG] (cit. on pp. 21, 37).

[Hor91] Kurt Hornik. „Approximation capabilities of multilayer feedforward networks“. In: *Neural Networks* 4.2 (1991), pp. 251–257 (cit. on pp. 39, 53).

[IL90] Neil Immerman and Eric Lander. „Describing graphs: A first-order approach to graph canonization“. In: *Complexity theory retrospective*. Springer, 1990, pp. 59–81 (cit. on pp. 11, 12, 47, 54).

[KB15] Diederik P. Kingma and Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. arXiv: 1412.6980v9 [cs.LG] (cit. on p. 56).

[Kek66] Aug. Kekulé. „Untersuchungen über aromatische Verbindungen. I. Ueber die Constitution der aromatischen Verbindungen.“ In: *Annalen der Chemie und Pharmacie* 137.2 (1866), pp. 129–196 (cit. on pp. 12, 47).

[Kie+17] Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. „The Weisfeiler-Leman dimension of planar graphs is at most 3“. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017. arXiv: 1708.07354v1 [cs.DM] (cit. on p. 12).

[Kri+20] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. „A survey on graph kernels“. In: *Applied Network Science* 5.1 (2020). arXiv: 1903.11835v2 [cs.LG] (cit. on p. 18).

[KW17] Thomas N. Kipf and Max Welling. „Semi-Supervised Classification with Graph Convolutional Networks“. In: *International Conference on Learning Representations* (2017). arXiv: 1609.02907v4 [cs.LG] (cit. on pp. 22, 36, 39, 50).

[LB98] Yann LeCun and Yoshua Bengio. „Convolutional Networks for Images, Speech, and Time Series“. In: *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258 (cit. on p. 21).

[Lee+19] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. „Self-Attention Graph Pooling“. In: *Proceedings of the 36st International Conference on Machine Learning*. ICML'19. Apr. 17, 2019, pp. 6661–6670. arXiv: 1904.08082v4 [cs.LG] (cit. on pp. 25, 40).

[Lip18] Zachary C. Lipton. „The Mythos of Model Interpretability“. In: *Queue* 16.3 (June 2018), 31–57. arXiv: 1606.03490v3 [cs.LG] (cit. on p. 29).

[LM14] Quoc Le and Tomas Mikolov. „Distributed Representations of Sentences and Documents“. In: *Proceedings of the 31st International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China, 2014, pp. II–1188–II–1196. arXiv: 1405.4053v2 [cs.CL] (cit. on p. 18).

[Lue+18] Thomas Luechtefeld, Dan Marsh, Craig Rowlands, and Thomas Hartung. „Machine Learning of Toxicological Big Data Enables Read-Across Structure Activity Relationships (RASAR) Outperforming Animal Test Reproducibility“. In: *Toxicological Sciences* 165.1 (2018), pp. 198–212 (cit. on p. 16).

[LV18] Andreas Loukas and Pierre Vandergheynst. „Spectrally Approximating Large Graphs with Smaller Graphs“. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 3237–3246. arXiv: 1802.07510v1 [cs.LG] (cit. on p. 25).

[MH16] Vitalik Melnikov and Eyke Hüllermeier. „Learning to Aggregate Using Uninorms“. In: *Machine Learning and Knowledge Discovery in Databases*. Springer International Publishing, 2016, pp. 756–771 (cit. on pp. 3, 4).

[MH19] Vitalik Melnikov and Eyke Hüllermeier. „Learning to Aggregate: Tackling the Aggregation/Disaggregation Problem for OWA“. In: *Proceedings of The Eleventh Asian Conference on Machine Learning*. Ed. by Wee Sun Lee and Taiji Suzuki. Vol. 101. Proceedings of Machine Learning Research. Nagoya, Japan: PMLR, 2019, pp. 1110–1125 (cit. on p. 5).

[Mik+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. „Distributed Representations of Words and Phrases and Their Compositionality“. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 3111–3119. arXiv: 1310.4546v1 [cs.CL] (cit. on p. 17).

[Mil02] R. Milo. „Network Motifs: Simple Building Blocks of Complex Networks“. In: *Science* 298.5594 (2002), pp. 824–827 (cit. on pp. 12, 47).

[Mor+17] Christopher Morris, Kristian Kersting, and Petra Mutzel. „Glocalized Weisfeiler-Lehman Graph Kernels: Global-Local Feature Maps of Graphs“. In: *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2017. arXiv: 1703.02379v3 [cs.LG] (cit. on pp. 20, 59).

[Mor+19] Christopher Morris, Martin Ritzert, Matthias Fey, et al. „Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (2019), pp. 4602–4609. arXiv: 1810.02244v3 [cs.LG] (cit. on pp. 24, 45).

[MP13] Brendan D. McKay and Adolfo Piperno. *Practical graph isomorphism, II*. Jan. 8, 2013. arXiv: 1301.1493v1 [cs.DM] (cit. on p. 7).

[MW97] A. D. McNaught and A. Wilkinson. „functional group“. In: *IUPAC Compendium of Chemical Terminology*. 2nd ed. IUPAC, 1997, p. 1116 (cit. on p. 28).

[Nar+17] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, et al. *graph2vec: Learning Distributed Representations of Graphs*. July 17, 2017. arXiv: 1707.05005v1 [cs.AI] (cit. on p. 18).

[New03] M. E. J. Newman. „The Structure and Function of Complex Networks“. In: *SIAM Review* 45.2 (2003), pp. 167–256 (cit. on pp. 12, 47).

[Nov02] Miroslav Novotný. „Homomorphisms of heterogeneous algebras“. In: *Czechoslovak Mathematical Journal* 52.2 (2002), pp. 415–428 (cit. on p. 39).

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, et al. „PyTorch: An Imperative Style, High-Performance Deep Learning Library“. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, et al. Curran Associates, Inc., 2019, pp. 8026–8037. arXiv: 1912.01703v1 [cs.LG] (cit. on p. 56).

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. „Scikit-Learn: Machine Learning in Python“. In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), 2825–2830. arXiv: 1201.0490v4 [cs.LG] (cit. on p. 59).

[Per+14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. „DeepWalk: online learning of social representations“. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*. ACM Press, 2014. arXiv: 1403.6652v2 [cs.SI] (cit. on p. 17).

[She+11] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. *Weisfeiler-Lehman Graph Kernels*. 2011 (cit. on pp. 18, 19).

[Shu+13] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. „The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains“. In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98 (cit. on pp. 13, 15).

[Sig+18] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, et al. *GraKeL: A Graph Kernel Library in Python*. June 6, 2018. arXiv: 1806.02193v2 [stat.ML] (cit. on p. 58).

[Wan+19] Minjie Wang, Lingfan Yu, Da Zheng, et al. *Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs*. Sept. 3, 2019. arXiv: 1909.01315v1 [cs.LG] (cit. on p. 55).

[Wel+07] Howard T. Welser, Eric Gleave, Danyel Fisher, and Marc Smith. „Visualizing the signatures of social roles in online discussion groups“. In: *Journal of Social Structure* (2007) (cit. on pp. 12, 47).

[WL68] Boris Weisfeiler and Andrei A. Lehman. „A reduction of a graph to a canonical form and an algebra arising during this reduction“. In: *Nauchno-Technicheskaya Informatsia* 2.9 (1968), pp. 12–16 (cit. on p. 7).

[Wu+19] Zonghan Wu, Shirui Pan, Fengwen Chen, et al. *A Comprehensive Survey on Graph Neural Networks*. Jan. 3, 2019. arXiv: 1901.00596v4 [cs.LG] (cit. on p. 21).

[WW86] Peter Willett and Vivienne Winterman. „A Comparison of Some Measures for the Determination of Inter-Molecular Structural Similarity Measures of Inter-Molecular Structural Similarity“. In: *Quantitative Structure-Activity Relationships* 5.1 (1986), pp. 18–25 (cit. on p. 16).

[Xu+19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. „How Powerful are Graph Neural Networks?“ In: *International Conference on Learning Representations*. ICLR'2019. 2019. arXiv: 1810.00826v3 [cs.LG] (cit. on pp. 23, 24, 36, 39, 50, 53, 59).

[Yin+18] Rex Ying, Jiaxuan You, Christopher Morris, et al. „Hierarchical Graph Representation Learning with Differentiable Pooling“. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, 4805–4815. arXiv: 1806.08804v4 [cs.LG] (cit. on p. 25).

[Zha+18] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. „An end-to-end deep learning architecture for graph classification“. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018 (cit. on pp. 25, 40).

Websites

[🔗DGL] Minjie Wang, Lingfan Yu, Da Zheng, et al. *Deep Graph Library*. URL: https://www.dgl.ai/ (visited on Apr. 4, 2020) (cit. on p. 55).

[🔗GK] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, et al. *GraKeL*. URL: https://ysig.github.io/GraKeL (visited on Apr. 4, 2020) (cit. on p. 58).

[🔗GN] *Graph Nets*. DeepMind. URL: https://github.com/deepmind/graph_nets (visited on Apr. 4, 2020) (cit. on p. 55).

[🔗NT] Brendan McKay and Adolfo Piperno. *nauty and Traces*. URL: http://pallini.di.uniroma1.it/index.html (visited on Feb. 21, 2020) (cit. on p. 7).

[🔗PT] Priyanka Banerjee, Robert Preissner, Andreas Eckert, and Anna K. Schrey. *ProTox-II - Prediction Of Toxicity Of Chemicals*. Charité – Universitätsmedizin Berlin. URL: http://tox.charite.de/protox_II/ (visited on Nov. 18, 2019) (cit. on p. 16).

[🔗PyG] Matthias Fey. *PyTorch Geometric*. URL: https://github.com/rusty1s/pytorch_geometric (visited on Apr. 4, 2020) (cit. on p. 55).

[🔗PyT] *PyTorch*. Facebook Inc. URL: https://pytorch.org/ (visited on Apr. 4, 2020) (cit. on p. 56).

[🔗SKL] *Scikit-Learn*. URL: https://scikit-learn.org/ (visited on Apr. 4, 2020) (cit. on p. 59).

[🔗Spe] Daniele Grattarola. *Spektral*. URL: https://spektral.graphneural.network (visited on Apr. 4, 2020) (cit. on p. 55).

[🔗TET] *Toxicity Estimation Software Tool (TEST)*. EPA. URL: https://www.epa.gov/chemical-research/toxicity-estimation-software-tool-test (visited on Nov. 18, 2019) (cit. on p. 16).

[🔗TF] *TensorFlow*. Google LLC. URL: https://www.tensorflow.org/ (visited on Apr. 4, 2020) (cit. on p. 56).

[🔗TT] *ToxTrack - Cheminformatics Modeling*. ToxTrack Inc. URL: https://toxtrack.com/ (visited on Nov. 18, 2019) (cit. on p. 16).

List of Figures

Erklärung zur Masterarbeit

Ich, Clemens Damke (Matrikel-Nr. 7011488), versichere, dass ich die Masterarbeit mit dem Thema *Learning to Aggregate on Structured Data* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinn nach entnommen habe, wurden in jedem Fall unter Angabe der Quellen der Entlehnung kenntlich gemacht. Das Gleiche gilt auch für Tabellen, Skizzen, Zeichnungen, bildliche Darstellungen usw. Die Masterarbeit habe ich nicht, auch nicht auszugsweise, für eine andere abgeschlossene Prüfung angefertigt. Auf § 63 Abs. 5 HZG wird hingewiesen.

Paderborn, 6. April 2020

Clemens Damke