# Summary of the lecture „MA INF 4316 - Graph Representation Learning“ by Dr. Pascal Welke (WiSe 2021/2022)

Fabrice Beaumont
Matrikel-Nr: 2747609
Rheinische Friedrich-Wilhelms-Universität Bonn

December 9, 2021

# Contents

# Chapter 0

# Motivations - Learning Tasks on Graphs

In this chapter we discuss different learning tasks on graph structured data.
A graph is a type of data structure which describes relationships (edges) between objects (vertices). For more formal definitions see definitions 0.2 and 0.1. Family trees, social network connections and molecules are examples for graphs.
Graphs may incorporate much information in form of attributes. Vertices for example may correspond to persons and may be attributed by name, age or gender. In this example edges may additionally contain information about relationship properties. Another example are molecular graphs, which have atom types as vertex labels and their edges indicate the type of covalent bond.
The three main learning paradigms of machine learning are:

- **Supervised Learning**: Find a data-consistent function that maps input to output values. In other words, given a set of training data $X = \{(x, f(x))\}$, find a hypothesis function $h \in \mathcal{H}$ such that $h \approx f$.
  (E.g. Linear Regression, Logistic Regression, Support Vector Machines, Nearest Neighbor, Naive Bayes Classifier, Decision Trees & Random Forests, (Deep) Neural Networks, ...)

- **Unsupervised Learning**: Find structures by identifying data correlations without explicit knowledge of memberships.
  (E.g. clustering, auto-encoders, (frequent) pattern mining, ...)

- **Reinforcement Learning**: Learn a function which maximizes the cumulative output rewards by choosing appropriate actions in an unknown environment.
  (Search space exploration to reason about the environment.)

Common supervised learning tasks on graphs are:

- Assign a color to unknown vertices
- Predict likely new edges
- Predict unknown graph class

Common unsupervised learning tasks on graphs are:

- Vertex clustering, Community detection
- Graph clustering
- Link prediction without supervision

The central problem of learning on graphs is how traditional machine learning approaches can be applied to graphs.
Linear Regression, Logistic Regression, Naive Bayes Classifiers, Decision trees and Random Forests expect *tabular data*, i.e. a set of example with a known, fixed set of features.
One may consider to use features of vertices to learn on them. But several non-trivial questions arise, for example:

- How to deal with connections between vertices?
- How to deal with vertices without features?
- What if the objects we want to learn from are graphs themselves? (Possibly with different numbers of vertices or edges.)

Note that the usage of a graph's adjacency matrix as vector representation is not desirable, since isomorphic graphs can be represented in different adjacency matrices.
Another approach to make traditional machine learning approaches applicable to graphs is to use **graph representations**. These vectorize data with respect to task specific properties. Generally, we seek approaches that incorporate aspects of the graph's structure.
It is highly non-trivial to determine which information crucial for a specific learning task shall be represented in which way in the graph representation.

**Example - Vertex classification:** Consider the graph depicted in Figure 1 and the task to reason about the color of unknown vertices by looking at training data.

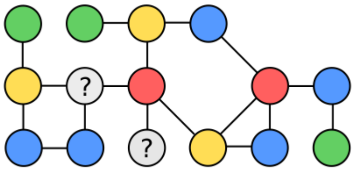


Figure 1 – Example of a graph with colors as vertex features.

A closer look reveals that, in this example, the color of a vertex corresponds to its degree. Thus $\deg(v)$ may be a simple representation of each vertex $v$.

**Example - Graph classification:** Consider the graph depicted in Figure 2 and the task to predict the class of an unclassified graph (e.g. the gray one in the figure).

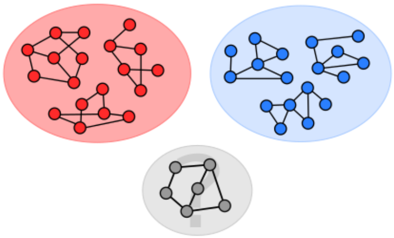


Figure 2 – Example of graph classification (clustering).

A common approach is to use sets of subgraphs to describe (similarity between) graphs. In this example it appears that graphs of the red class contain cycles of length four whereas graphs of the blue class contain triangles. Thus, a graph $G$ may be represented by a vector $\phi(G)$ containing cycle counts of length three and four.

**Example - Link prediction:** Consider the graph depicted in Figure 3 and the task to predict the likelihood of a not jet existing edge (e.g. the gray one in the figure).

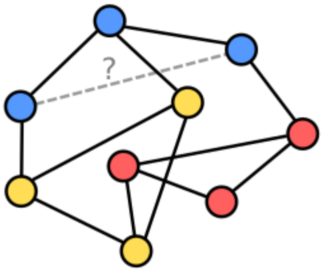


Figure 3 – Example of graph classification (clustering).

A common approach is to consider the shortest path distance and similarity in terms of neighborhoods or vertex attributes. In this example the vertices of same color are more likely to be connected by an edge. Thus one may represent edges using the adjacent vertex colors.

As the given examples indicate, it can be tedious to analyze graph data and choose a suitable graph representation. As it turns out, Machine Learning can be used for learning the graph representations themselves. In the following chapter we will cover two aspects of graph representation learning:

- **Learning with representations** of graphs and
- **learning representations** of graphs.

# Chapter 1

# Vertex classifications and local features

**Definition 0.1: Directed graph**

A **directed graph** (**digraph**) $G = (V, E)$ consists of a set of vertices $V(G)$ and a set of edges $E(G) \subseteq V \times V$.

**Definition 0.2: Undirected Graph**

A **undirected graph** $G = (V, E)$ consists of a set of vertices $V(G)$ and a set of edges $E(G) \subseteq 2^V$ such that

$$\forall e \in E(G) : \qquad |e| = 2$$

As variable naming conventions, we say that graphs have $n = |V(G)|$ vertices and $m = |E(G)|$ edges.

**Definition 0.3: Simple graphs**

A simple graph is a graph, where edges connot appear multiple times.

(That is, in the respective definitions of directed and undirected graphs, the set of edges is not a multiset.)

**Definition 0.4: Graph attributes and labels**

Given an undirected or directed graph $G$, one can define additional data associated with the graph, vertices or edges. Such data is called **attributes** (if the value domain is continuous) or **labels** (if the value domain is discrete/categorical).
This association can be formalized using **feature spaces** $\mathcal{X}$, $\mathcal{X}'$ and $\mathcal{X}''$:

$$f_G : G \to \mathcal{X}, \qquad f_V : V(G) \to \mathcal{X}', \qquad f_E : E(G) \to \mathcal{X}''$$

(A feature space is some product space $\mathcal{X} = (X_1, X_2, \ldots, X_d)$ where the $X_i$ are some appropriately chosen sets.)

**Example - Graph labels** Typical examples of graph vertex labels are atom names (chemical graphs).

**Definition 0.5: Graph isomorphism**

Let $G$ and $H$ be two graphs. $G$ and $H$ are **isomorphic**, if and only if there exists a bijective function between the vertices $\varphi : V(G) \to V(H)$ such that

- edges are preserved:
  $\{v, w\} \in E(G) \iff \{\varphi(v), \varphi(w)\} \in E(H)$
- vertex attributes are preserved:
  $\forall v \in V(G) : \ f_V(v) = f_V\big(\varphi(v)\big)$
- edge attributes are preserved:
  $\forall \{v, w\} \in E(G) : \ f_E(\{v, w\}) = f_E\big(\{\varphi(v), \varphi(w)\}\big)$

## 1.1 Property prediction for vertices

Learning functions (properties) on vertices is done under the assumption that there is a multiset of objects $X \subseteq \mathcal{X}$ and there exists an (partially) unknown function $y : X \to \mathcal{Y}$ and some pairs of objects are associated. This setting is formalized in definition 1.1.

**Definition 1.1: Supervised vertex property prediction on fixed graphs**

**Input:** a graph $G$ with labeling functions $f_V, f_E$,
a hypothesis class $\mathcal{H} \subseteq \{h |\ h : V(G) \to \mathcal{Y}\}$,
properties $y(v)$ for a set of vertices $v \in V' \subseteq V(G)$ and
a loss function $L_{V,y} : \mathcal{H} \to \mathbb{R}$.
**Output:** $\operatorname{argmin}_{h \in \mathcal{H}} L_{V,y}(h)$.

Note that one can extend hypothesis classes on graph vertices to their vertex representation. More explicitly, let $\mathcal{X}$ be a feature space and $r : V(G) \to \mathcal{X}$ some vertex representation on the graph $G$. Let $\mathcal{H} \subseteq \{h |\ h : \mathcal{X} \to \mathcal{Y}\}$ be a hypothesis class. Then the *extended hypothesis class*

$$\mathcal{H}' := \{h \circ r |\ h \in \mathcal{H}\} \qquad \text{(Eq. 1)}$$

is a hypothesis class on $V(G)$, too.
On top of that one can generalize the vertex representation $r$ to a set of several graph representation $\mathcal{R} \subseteq \{r |\ r : V(G) \to \mathcal{X}\}$ and yield the further extended hypothesis class $\mathcal{H}' := \{h \circ r |\ r \in \mathcal{R},\ h \in \mathcal{H}\}$ on $V(G)$.

### 1.1.1 Loss Functions - MERGE & MOVE WITH THE STATISTIC SCRIPT

To measure the quality of hypothesis, with respect to vertex property predictions, one can use a variety of loss functions. Most of these can be generalized to other applications of hypothesis evaluation.

**Definition 1.2: Classification Loss**

The **classification loss** of a hypothesis on vertices to predict their properties is defined as the probability of false predictions:

$$L_{V,y}(h) := \mathbb{P}\left[y(v) \neq h(v)\right] = 1 - \frac{1}{|V(G)|} \sum_{v \in V(G)} \delta\left(y(v), h(v)\right) \qquad \text{(Eq. 2)}$$

where

$$\delta(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}$$

**Definition 1.3: Regression**

The **mean squared error** can be used as a loss function of a hypothesis on vertex property prediction. Using it to improve the quality of the found hypothesis is called **regression**:

$$L_{V,y}(h) := \mathbb{E}\left[\left(y(x) - h(x)\right)^2\right] = \frac{1}{|V(G)|} \sum_{v \in V(G)} \left(y(v), h(v)\right)^2 \qquad \text{(Eq. 3)}$$

When evaluating a (machine learning) model during training, we test a hypothesis on empirical data and optimize for the best solution. Both the optimization and the empirical data induce a bias on the estimate of the quality of the hypothesis. This bias can be reduced using **cross validation** (TODO - see Figure **??**.

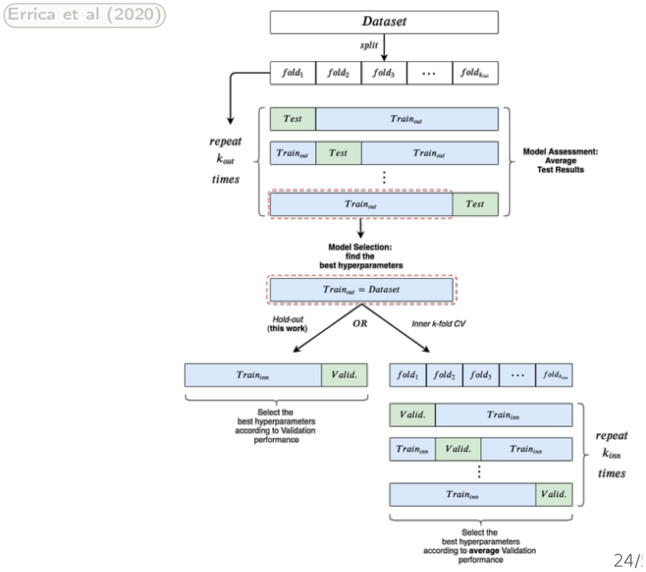


Figure 1.1 – TODO: replace image

Cross validation can be used to compare different hypothesis. If there is a statistically significant difference between their performances, one could be better than the other.

### 1.1.2 Baselines for graph learning

Interestingly, some baseline methods, which do not include specific graph structures may perform well, even better than GNNs on some graph datasets. In other words, improvements that do not clearly outperform structure-agnostic competitors should be viewed with moderation[14, 15, 17, 16].
Note that whenever information about individual vertices is available, one should consider using it extensively. In many cases, it is more valuable for the task at hand, than the actual graph structure that combines these.
To conclude, to evaluate a graph representation one should measure its performance against a model which ignores the graph structure. Thus when dealing with an attributed graph $G$ it can be beneficial to start with its vertex attributes $f_V : V(G) \to \mathcal{X}$ and first consider a hypothesis class over $\mathcal{X}$, only. Only after that, one may think about including graph representations into the hypothesis class.

## 1.2 Local Feature Counts

In this subsection we will define various simple vertex representations $f : V(G) \to \mathbb{R}$ for fixed graphs. Let $\mathcal{R}'$ be the set of all one-dimensional vertex representations. These representations can be grouped in the following way:

$$\mathcal{R} := \{\times_R : V(G) \to \mathbb{R}^{|R|} \mid R \subseteq \mathcal{R}'\} \tag{Eq. 4}$$

where $\times_R(v) = \big(r_1(v), r_2(v), \ldots, r_{|R|}(v)\big)$ for a vertex $v$ and representations $r_i \in R$.

**Definition 2.1: Vertex neighborhood**

Let $G$ be a undirected graph and $v \in V(G)$. The **neighborhood** of $v$ is the set

$$N(v) := \{w \mid \{v, w\} \in E(G)\}$$

If $G$ is directed, we define respectively the **outgoind neighborhood** as

$$N_{\text{out}}(v) := \{w \mid (v, w) \in E(G)\}$$

and the **incoming neighborhood** as

$$N_{\text{in}}(v) := \{w| \ (w, v) \in E(G)\}$$

**THEOREM 2.1 - Full Neighborhood Representation:**

By fixing an order of the vertices $V = \{v_1, \dots, v_n\}$ and defining

$$A(v_j)[i] := \begin{cases} 1 & \text{if } v_i \in N(v_k) \\ 0 & \text{if } v_i \notin N(v_k) \end{cases} \tag{Eq. 5}$$

one can represent a vertex by its neighborhood. This is called **Full Neighborhood Representation**.

The drawbacks of a Full Neighborhood Representation include:

- The space requirements grow quadratically with $n$.
- Learning algorithms that use this representation will likely overfit.
- It requires additional effort to check, whether a given vertex is already included in the representation.
- It requires additional effort to find neighbors of neighbors.

The first two drawbacks can be reduced, by so called Compressed Neighborhood Representations.

**Definition 2.2: Vertex degree**

Let $G$ be a undirected graph and $v \in V(G)$ be a vertex. The **degree** is the number of edges incident to $v$, i.e.

$$\delta(v) := |N(v)|$$

if $G$ is directed, we define respectively the **outdegree neighborhood** as the number of (incident) edges starting in $v$, i.e.

$$\delta_{\text{out}}(v) := |N_{\text{out}}(v)|$$

and the **indegree neighborhood** as the number of (incident) edges ending in $v$, i.e.

$$\delta_{\text{out}}(v) := |N_{\text{out}}(v)|$$

The neighborhood $N(v)$ of a vertex $v$ induces a subgraph $G[N(v)]$ with the vertices in $N(v)$ and all edges between these vertices:

$$G[N(v)] = \Big(N(v), \{\{w, w'\}|\ w, w' \in N(v)\}\Big)$$

The generalized definition is given in definition 2.4. This subgraph $G[N(v)]$ can have at most

$$\frac{\delta(v)\big(\delta(v) - 1\big)}{2}$$

edges, since every of the $\delta(v)$ neighbors of $v$ can have at most one edge, to all other $\delta(v)$ neighbors in $G[N(v)]$. Notice, not to count every edge twice in this argumentation. If there are exactly this many edges (without loops), the subgraph is *fully connected*.

**Definition 2.3: Neighborhood density[6, 2]**

The **density** of a vertex's neighborhood $v$ is defined as the ratio of all existing edges and all possible edges (in a fully connected neighborhood):

$$c(v) := \frac{2\big|E\big(G[N(v)]\big)\big|}{\delta(v)\big(\delta(v) - 1\big)}$$

**Definition 2.4: Induced subgraph**

Let $G$ be a graph and $X \subseteq V(G)$ a set of its vertices. The **subgraph** of $G$ **induced by** $X$ is the graph $H$ with

- $V(H) = X$ and
- $E(H) = \big\{\{v, w\}|\ \{v, w\} \in E(G)\ \wedge\ v, w \in X\big\}$

A **triangle graph** is a simple graph on three vertices with three edges.

**Definition 2.5: Triangle subgraphs**

Let $\mathcal{D}(v)$ be the set of all triangle subgraphs of a graph $G$ that contain the vertex $v$.

Let $\Delta(v) := |\mathcal{D}(v)|$ be its carnality.

**THEOREM 2.2 - Upper bound on the adjacent triangle count:**

Let $G$ be a graph and $v \in V(G)$. Then

$$\Delta(v) \leq \left|E(G[N(v)])\right|$$

**Proof:** Let $D \in \mathcal{D}(v)$ be an arbitrary triangle adjacent to $v \in V(G)$. Then $V(D) \subseteq N(v) \subseteq G[N(v)]$ since in a triangle all vertices are adjacent and thus all vertices in the triangle are adjacent to $v$ (i.e. contained in its neighborhood). Notice, that there are two kind of edges in $G[N(v)]$, the ones incident to $v$ ($v \in e$, there are $\delta(v)$ many of them) and these which are not ($v \notin e$). The ones incident to $v$ can be part of at most two triangles in $\Delta(v)$, while the edges that are not incident to $v$ can be part in exactly one triangle in $\Delta(v)$. That is because the two vertices that are adjacent to such an edge, together with $v$, completely define a triangle (if all edges are present). On the other hand, every triangle in $D(v)$ must contain exactly one edge, that is not incident to $v$ (and two which are incident to $v$).

Therefore the number of triangles $\Delta(v)$ is equal to the number of edges in $G[N(v)]$ that are not incident to $v$, which is a fraction of all edges in $G[N(v)]$:

$$\begin{aligned}\Delta(v) &= \left|\{e \in E(G[N(v)]) | \; v \notin e\}\right| \\ &\leq \left|\{e \in E(G[N(v)]) | \; v \notin e\}\right| + \delta(v) \\ &= \left|E(G[N(v)])\right|\end{aligned}$$

(Note that in the second equation, the equality is only achieved, if $v$ has no neighbors.) □

**Definition 2.6: Induced Subgraph Isomorphism Problem**

**Input:** two graphs $G$, $H$.

**Output:** the decision if $H$ is isomorphic to an induced subgraph of $G$.

**THEOREM 2.3 - Difficulty of the Ind. Subgraph Iso. Pb.:**

The induced subgraph isomorphism problem is **NP-complete**.

**Proof:** Let $K_n$ be a complete graph on $n$ vertices. Then solving the induced subgraph isomorphism problem for $K_n$ and $G$ solves the $k$-Clique Problem for $G$, which is NP-complete itself. □

Notice, that if one could count the induced subgraphs for two graphs efficiently, one could decide the induced subgraph isomorphism problem. Hence *counting induced subgraphs is NP-hard*.

But if one keeps the size of the induced subgraphs fix, the problem becomes tractable.

### 1.2.1 Graphlet Counting

**Definition 2.7: Graphlet**

Small connected induced subgraphs are called **graphlets**.

We would like to count, for each vertex $v$, the number of graphlets of a certain type, that contain $v$. Lets call these **$v$-incident graphlets**.

**Definition 2.8: Incident Induced Graphlets Count Problem**

**Input:** a graph $G$ and
a graphlet $H$ of size $k$.

**Output:** the number of $v$-incident induced subgraphs isomorphic to $H$ for all $v \in V(G)$:

$$r_H(v) := \left|\{X \subseteq V|\ v \in X \ \wedge\ G[X] \overset{\text{iso.}}{\equiv} H\}\right|$$

**Algorithm 1** Brute Force - Counting Incident Graphlets

**Input:** a graph $G$ and
a graphlet $H$ of size $k$.
**Output:** the number of $v$-incident induced subgraphs isomorphic to $H$ for all $v \in V(G)$.

```
gc(v) = 0 for all $v \in V(G)$
for all $X \subseteq V(G)$ with $|X| = k$ do
    if $G[X]$ is isomorphic to $H$ then
        gc(x) = gc(x) + 1 for all $x \in X$
```

**Runtime**: There are $\binom{n}{k}$ choices for $X$, and there need $\gamma(k) \in \mathcal{O}(k!\,k^2)$ graphs to be checked for isomorphism. Thus the total runtime for a constant $k$ is

$$\mathcal{O}\big(n^k\,\gamma(k)\big) = \mathcal{O}(n^k)$$

Since a lot of graph structured data can be considered Big Data (w.r.t. size; not speed), cubic runtime in the number of vertices is not acceptable.

**THEOREM 2.4 - Runtime bound on the Incident Induced Graphlets Count Problem[9]:**

Let $G$ be a bounded degree graph and let $d$ denote the maximum degree in $G$. If all graphlet structures size $k \in \{3, 4, 5\}$ are known (precomputed), then all $v$-incident graphlets of $G$ with size $k \in \{3, 4, 5\}$ can be enumerated in $\mathcal{O}(nd^{k-1})$ time.

Notice that the maximum degree $d$ of a graph is typically much smaller than the number of its vertices $n$.

**Proof:** Graphlets of size $k$ can be divided into two classes:

1. Graphlets that contain a simple path of length $k - 1$ and
2. Graphlets that to not contain such a path.

First, lets consider graphlets with paths of length $k - 1$ and one fixed vertex $v \in V(G)$. We can enumerate all such paths in $G$ starting from $v$ by a depth-first search in $\mathcal{O}(d^{k-1})$.

Each such path $P$ induces a subgraph $G[V(P)]$ containing $v$, which we can construct in $\mathcal{O}(k^2)$ time using appropriate data structures.
We can find the graphlet that $G[V(P)]$ is isomorphic to in $\gamma(k) \in \mathcal{O}(k!\, k^2)$ time and increase its count for $v$:

- Consider all $k!$ permutations of the vertex set of $G[V(P)]$ and
- check if the resulting adjacency matrix is identical to one of our precomputed graphlets ($\mathcal{O}(k^2)$, there are less than 30 of them).

Notice, that this approach has a couple of drawbacks:

1. If a graphlet may contain more than one path of length $k-1$, it will be counted/found multiple times!
2. If $v$ is only in the middle of a path of length $k-1$ including graphlet, not graphlets that contain $v$ are found!

There exist a solution for these drawbacks: Let $G[X]$ be a graphlet and $|X| = k$. Each $k-1$ path inducing $G[X]$ will be found exactly twice, one starting from each endpoint $v \neq v'$. Thus we will find $G[X]$ twice for each $k-1$ path in it. Thus increasing a counter for each $w \in X$ each time we find $G[X]$ will yield a final count of

$$2\#(k-1 \text{ paths in } G[X])$$

□

**Algorithm 2** Exact Graphlet Counting Algorithm

**Input:** a graph $G$ and
a graphlet $H$ of size $k$ and $1 \leq p$ paths of length $k-1$.
**Output:** the number of $v$-incident subgraphs isomorphic to $H$ for all $v \in V(G)$.

```
gc$(v) = 0$ for all $v \in V(G)$
for all $v \in V(G)$ do
    for all $k-1$ length paths $P$ in $G$ starting in $v$ do
        if $G[V(P)]$ is isomorphic to $H$ then
            gc$(x) =$ gc$(x) + 1$ for all $x \in X$
for all $v \in V(G)$ do gc$(v) = \frac{\text{gc}(v)}{2p}$          ▷ B.c. of double counting
```

To understand the runtime complexity, we need to consider how many graphlets without paths of length $k-1$ here are.

For $k=2$ There are no such graphlets.

For $k=3$ There are no such graphlets.

For $k=4$ There is only the 3-star.
We need to lookup all $\mathcal{O}\big(\binom{d}{3}\big)$ neighbor triplets of $v$ and check if they induce the 3-star graphlet. This can be done in $\mathcal{O}(d^3\gamma(4))$.

Fpr $k=5$ All connected graphlets with no 4-paths have the 3-star as subgraph.
Thus enumerate all occurrences of the 3-star and check the neighbors of each vertex (in the star) to see if they induce one of the graphlets. This takes $\mathcal{O}(d\gamma(5))$ per induced graph.

Thus the overall runtime is $\mathcal{O}(d^4\gamma(5))$.
Now, considering the relevance of the 3-star we notice that since it has only one center, they cannot be found twice. Thus one may think, that the normalization step of the counter can be avoided.
However, the counts for graphlets of size five need to be normalized by the number of subgraphs of the pattern that are isomorphic to the 3-star. Lets use this knowledge in another version of the algorithm.

**Algorithm 3** Exact Graphlet Counting Algorithm 2

**Input:** a graph $G$ and
the 3-star $H$.
**Output:** the number of $v$-incident subgraphs isomorphic to $H$ for all $v \in V(G)$.

```
gc($v$) = 0 for all $v \in V(G)$
for all $v \in V(G)$ do
    for all $X \subseteq N(v)$ with $|X| = 3$ do
        if $G[X \cup \{v\}]$ is isomorphic to $H$ then
            gc($x$) = gc($x$) + 1 for all $x \in X \cup \{v\}$
```

**Runtime:** For pattern $H$ with $|V(H)| = k$ the runtime is $\mathcal{O}(nd^{k-1})$
Repeating this for the set $\mathcal{G}_{3,4,5}$ for all 29 pairwise non-isomorphic graphs of size $k = 3, 4, 5$ is also in $\mathcal{O}(nd^{k-1})$.

**Algorithm 4** Exact Graphlet Counting Algorithm 3

**Input:** a graph $G$ and
the 3-star $H$ of size 5 without simple $k-1$ length paths and $1 \leq x$ 3-stars.
**Output:** the number of $v$-incident subgraphs isomorphic to $H$ for all $v \in V(G)$.

```
$\text{gc}(v) = 0$ for all $v \in V(G)$
for all $v \in V(G)$ do
    for all $X \subseteq N(v)$ with $|X| = 3$ do
        if $G[X \cup \{v\}]$ is isomorphic to $H$ then
            for $w \in \bigcup_{x \in X \cup \{v\}} N(x)$ do
                if $H$ is isomorphic to $G[X \cup \{v, w\}]$ then
                    $\text{gc}(x) = \text{gc}(x) + 1$ for all $x \in X \cup \{v, w\}$
for all $v \in V(G)$ do $\text{gc}(v) = \frac{\text{gc}(v)}{2p}$        ▷ B.c. of double counting
```

**Runtime:** For pattern $H$ with $|V(H)| = k$ the runtime is $\mathcal{O}(nd^{k-1})$
Repeating this for the set $\mathcal{G}_{3,4,5}$ for all 29 pairwise non-isomorphic graphs of size $k = 3, 4, 5$ is also in $\mathcal{O}(nd^{k-1})$.

(Note that in practice the order of the loops should be changed.)

With these algorithms at hand, we can proclaim that for all $H \in \mathcal{G}_{3,4,5}$ we can efficiently compute

$$r_H(v) \ := \ |\{X \subseteq V|\ v \in X \wedge G[X] \text{ is isomorphic to } H\}|$$

and hence we can efficiently compute the representation of graphs by $2, 3, 4, 5$-graphlets:

$$\times_{\{r_H|\ H \in \mathcal{G}_{2,3,4,5}\}} : V(G) \to \mathbb{R}^{29} \qquad \text{(Eq. 6)}$$

One may ask, how significant the runtime improvement from $\mathcal{O}(n^k)$ in Algorithm 1 to $\mathcal{O}(nd^{k-1})$ in Algorithm 4 really is. As it turns out this improvement is the change between an impractical and a practical algorithm since in the latter only connected subgraphs considered, which is as desired. Im most real world graphs, there are much less such connected sets than there are subsets.

One may also ask, how significant these results are, given that we only considered small graphlets. As it turns out, in many databases these small graphlets can be interpreted more easily than bigger ones. For example in social net-

works, many triangle graphlets may indicate that friends of friends tend to be friends as well. While many 2-paths imply the opposite. Nevertheless, larger graphlets can be useful in practice too.
Keep in mind, that so far we considered unlabeled graphs. Taking labels into account yields much more pairwise non-isomorphic graphs. Note that in practice, graphlet counting (and multiple other pattern based representation methods) tends to *work better with discretely labeled attributes* than with continuous attributes (e.g. $f_V : V(G) \to \mathbb{R}$). Thus it may be desirable to discretize given continuous labels.

## 1.3 Distance based vertex representations

We have seen that graphlet counting (as an example of local feature counting) extends the trivial neighborhood set representation to the immediate neighborhood. But it to does not go beyond a few neighbors.

### 1.3.1 Centrality measures with distances

**Definition 3.1: Geodesic graph distance**

Let $G$ be a graph and $v, w \in V(G)$. Then the (**geodesic**) **distance** $d(v, w)$ is the length of the **shortest path** in $G$ with $v$ and $w$ as its endpoints.
More explicitly, the **unweighted** (**geodesic**) **distance** is the number of edges in the path. On the other hand the **weighted** (**geodesic**) **distance** for some weight function $c : E(G) \to \mathbb{R}_{\geq 0}$ is the sum of weights of edges in this path.
If such a path between $v$ and $w$ does not exist, set $d(v, w) := \infty$.

A trivial but expensive approach to represent the graph structure is to compute and store all distances to all other vertices in the graph. This has time complexity

- $\mathcal{O}(nm)$ using BFS for *unweighted* graphs and
- $\mathcal{O}(n^2 \log(n) + nm)$ using Johnsons algorithm for *weighted* graphs

and space complexity $\Theta(n^2)$. For many (big) real world graphs this is infeasible.

**Definition 3.2: (Vertex) closeness centrality**

Let $G$ be a connected graph and $v \in V(G)$. The **closeness centrality** of $v$ is defined as

$$C_{\mathrm{c}}(v) := \begin{cases} \infty & \text{if } G \text{ is disconnected} \\ 0 & \text{if } G \text{ has only one vertex} \\ \frac{n-1}{\sum_{w \in V(G)\setminus\{v\}} d(v,w)} & \text{else} \end{cases} \qquad \text{(Eq. 7)}$$

That is the ratio of the distance of all vertices to $v$, if $v$ would be connected to every vertex - and the actual accumulated distances to all vertices.

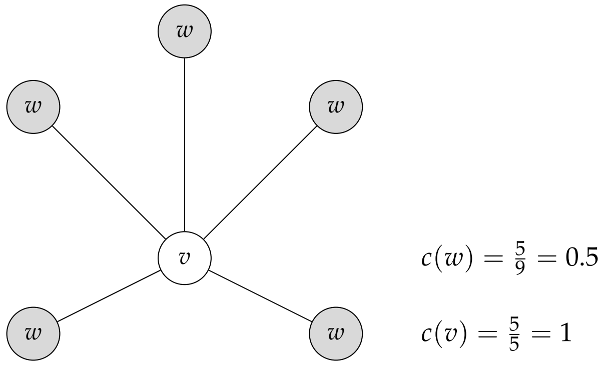


Figure 1.2 – Example for the closeness centrality measure

Note that this closeness centrality is a well-known centrality measure. A scaled version of it is used for example in the k-means cluster reassignment step. There one is optimizing with respect to the accumulated pairwise Euclidean distance:

$$\operatorname*{argmin}_{x \in \mathbb{R}^2} \sum_{y \in C_i} L_2^2(x,y) \; = \operatorname*{argmax}_{x \in \mathbb{R}^2} \frac{1}{\sum_{y \in C_i} L_2^2(x,y)} \; = \operatorname*{argmax}_{x \in \mathbb{R}^2} \frac{n-1}{\sum_{y \in C_i} L_2^2(x,y)}$$

**Definition 3.3: Harmonic closeness centrality**

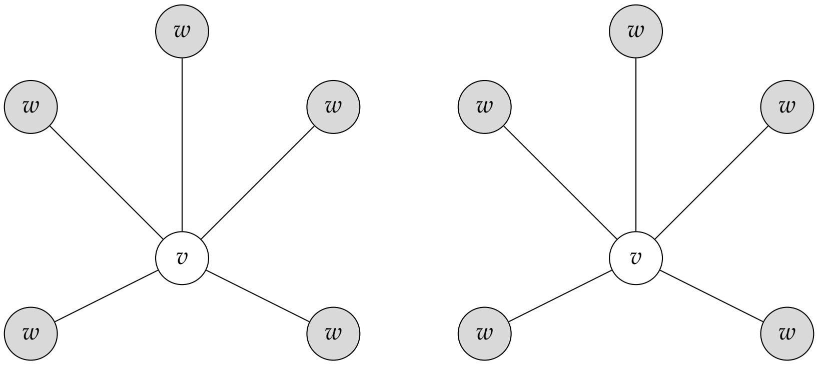
Let $G$ be a graph and $v \in V(G)$. The **harmonic closeness centrality** of $v$

is

$$C_{\text{hc}}(v) := \frac{1}{n-1} \sum_{w \in V(G) \setminus \{v\}} \frac{1}{d(v,w)} \qquad \text{(Eq. 8)}$$

while defining $\frac{n-1}{\infty} := 0$.

Thus $C_{\text{hc}}$ is well defined on disconnected graphs and again zero if there is only one vertex in $G$.



$c(v) = \frac{1}{9} * 4\frac{1}{1} = \frac{4}{9}$ $\qquad$ $c(w) = \frac{1}{9} * \left(\frac{1}{1} + 3\frac{1}{2}\right) = \frac{5}{18}$

Figure 1.3 – Example for the closeness centrality measure

Unfortunately, no algorithm is known to compute the closeness centrality for all vertices faster than by trivially computing all-pairs-shortest paths. This is possible in $\mathcal{O}(n^2 \log(n) + nm)$ in the weighted case.

**Definition 3.4: Sampled harmonic closeness centrality**

Let $G$ be a graph, $v \in V(G)$ and $L \subseteq V(G)$. The **sampled harmonic closeness centrality** of $v$ with respect to $L$ is

$$C_{\text{hc},L}(v) := \frac{1}{|L \setminus \{v\}|} \sum_{w \in L \setminus \{v\}} \frac{1}{d(v,w)} \qquad \text{(Eq. 9)}$$

while defining $\frac{|L|}{\infty} := 0$.

This is the harmonic closeness centrality, limited to a subset $L$ of vertices of the graph.

The sampled harmonic closeness centrality for all vertices an be computed in $\mathcal{O}\left(|L|(m+n\log(n))\right)$ by running Dijkstras algorithm starting from each vertex $l \in L$, which we call **landmarks**. This approach can be implemented to have $\mathcal{O}(n)$ space complexity (in addition to storing the graph itself).
Similar to the graphlet graph representation equation Eq. 6 we can express the representation in terms of these landmarks:

$$\times_{\{d_l|\ l\in L\}} : V(G) \to \mathbb{R}^{|L|} \qquad \text{(Eq. 10)}$$

A real life interpretation of this sampled harmonic closeness would be to define ones position by saying: “I am 1 km way from the university and 500 m way from the church.” The usefulness of this description depends on the selected landmarks $L$ of course. There is no generally optimal strategy to select them. Some example approaches are:

- random initialization
- high-degree graph vertices
- in 2D-embedded (planar) graphs, use vertices on the convex hull
- vertices with high (or low) closeness centrality
- informed choice: cover the graph such that most vertices have at least one landmark close by

### 1.3.2 Centrality measures without distances

The sampled closeness centrality proposed in the last subsection might not be good enough. On the other hand, more exact closeness computations are easily to expensive. On top of that, in some graphs, all vertices might be almost equally central (curse of dimensionality). Thus in this subsection we will explore alternatives to distances.

#### Degeneracy

Another way of defining centrality is via *density*. Intuitively, it would make sense for a graph to have a dense core and a sparse shell. To be efficient, one has to be careful to not solve the Clique problem along the way.

### Definition 3.5: Vertex degree

Let $G$ be a graph and $H$ be a subgraph of $G$. For $v \in V(H) \subseteq V(G)$ we denote the degree of $v$ in $H$ by $\delta_H(v)$.

### Definition 3.6: *k*-core component of a graph

A ***k*-core component** of a graph $G$ is an inclusion-wise maximal connected induced subgraph $G[X]$ such that each vertex $x \in X$ has at least degree $k$ in the induces subgraph

$$\delta_{G[X]}(x) \geq k$$

### Definition 3.7: *k*-core of a graph

The subgraph $H(k)$ of $G$ consisting of all $k$-core components is called the ***k*-core of** $G$.

### Definition 3.8: Degeneracy

The **degeneracy** $\deg(v)$ of $v \in V(G)$ is the largest $k$ such that $v$ is contained in the $k$-core of $G$.

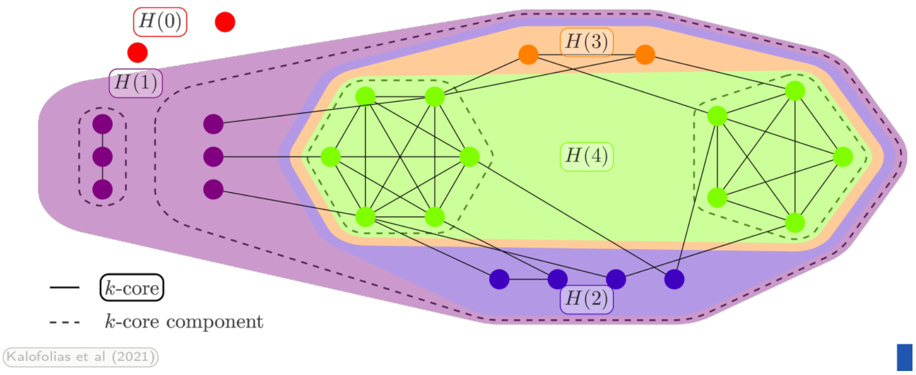


Figure 1.4 – Example graph with all $k$-cores and thus degeneracies visualized[18].

**Definition 3.9: Smallest first vertex ordering**

The vertices $v_1, \dots, v_n$ of $G$ are in **smallest first ordering** if for all $i \in \{1, \dots, n\}$ holds:

$$v_i \text{ is the vertex with smallest degree in } G\left[\{v_i, \dots, v_n\}\right]$$

**Algorithm 5** Degeneracy sweep

**Input:** a graph $G$.
**Output:** the degeneracies deg of all vertices $v \in V(G)$.

```
1: $k = 0$
2: for $i \in \{1, \dots, n\}$ do
3:     Remove $v = \operatorname{argmin}_{v \in V} d_{\min}$ and all incident edges from $G$
4:     $k = \max(d_{\min}, k)$
5:     $\deg(v) = k$
```

**Runtime**: $\mathcal{O}(n)$.
**Special property:** The algorithm processes the vertices in smallest first vertex order.

**THEOREM 3.1 - Complexity of degeneracy[3]:**

Let $G$ be a graph. The degeneracy of all vertices $v \in V(G)$ can be computed in linear time.

**Proof:** We have proven the claim if algorithm 5 runs in linear time. Therefore we need to show that $\deg(v) = k$ in line 5 correctly computes the degeneracy of $v$ in each iteration, since every vertex is touched exactly once in the for loop. Let $v_1, \dots, v_n$ be the smallest first vertex order used by the algorithm. Let $d_i := \delta_{G\left[\{v_i, \dots, v_n\}\right]}(v_i)$. Then

$$\forall i \in \{1, \dots, n\} : \qquad \delta_{G\left[\{v_i, \dots, v_n\}\right]}(v) \geq d_i \qquad \text{(Eq. 11)}$$

That is, $G\left[\{v_i, \dots, v_n\}\right]$ is a subgraph of the $d_i$-core of $G$.
This also implies that $G\left[\{v_i, \dots, v_n\}\right]$ is a subgraph of the $d_j$-core for all $1 \leq j \leq i$.

Hence, line 4 of the algorithm sets a lower bound on the degeneracy of each vertex coming after the current one in the smallest first ordering.
Now let $H$ ba a $k$-core component in $G$ and let $v_1, \ldots, v_n$ be any smallest first ordering. Let $i$ be the smallest index with $v_i \in V(H)$. We claim that it is $d_i \geq k$. To se this note that as $V(H) \subseteq \{v_i, \ldots, v_n\}$, $H$ is a subgraph of $G\big[\{v_i, \ldots, v_n\}\big]$. This implies

$$d_i = \delta_{G\big[\{v_i, \ldots, v_n\}\big]}(v_i) \geq \delta_H(v_i) = k$$

Thus line 4 of the algorithm also sets an upper bound on the degeneracy of each vertex. Thus the equality $d_i = k$ is proven. □

With respect to the space complexity of Algorithm 5 we note that in each iteration we have to find the remaining vertex with the smallest degree, remove it, and update the degrees of the (ex-)neighbors. We can do this in linear time using lists and arrays:

- Store $G$ in a standard adjacency list format.
- In an array $A$, for each $\delta \in \{1, \ldots, n\}$ store a **double linked list**, initialized with all vertices that have degree $i$.
- Om am array $B$, for each vertex, store its current degree and a pointer to its unique list entry maintained in $B$.
- Start with the first non-empty list in $A$, popping the first element $v$, decreasing the degrees of its neighbors by one in $B$, and moving them from their current list in $A$ to the list of their new degree (using the pointer in $B$ for constant time access to the list entries).
- Continue looking for the first non-empty list in $A$, *starting with the previous list* in $A$, checking if there is now a vertex with that degree.

Keep in mind that as $A$ contains double linked lists, removing and adding list items takes constant time.

## 1.4 Message passing

Note that the power set of a set $X$ can be denoted as $2^X$. Since multisets contain objects a non-negative number of times, the set of all multisets of objects from $X$ can be denotes as $\mathbb{N}^X$.

We will now introduce a very flexible framework. Assume that we have some feature representation $r_0 : V(G) \to \mathbb{R}^d$ for the vertices in our graph. It is reasonable that in many situations neighboring vertices influence each other. Consider a social network where users spread their content along connections to their affiliates. In turn, neighbors might be influenced by that and hence spread (a variant of) that information (aka. "retweet").

**Message passing** models this kind of behavior as a *simultaneous round based process*:

**Definition 4.1: Message passing**

**Message passing** models a simultaneous round based process. Let $v \in V(G)$ be a vertex, $k \geq 0$ and $r_0 : V(G) \to \mathcal{X}_0$ a feature representation.
$\{\{r_k(w) |\ w \in N(v)\}\}$ is the multiset of (old) representations of the neighbors of $v \in V(G)$. Define the following two functions:

- $\text{agg}_k : \mathbb{N}^{\mathcal{X}_k} \to \mathcal{X}'_k$ **aggregates** a set of (old) representations to some value.
- $\text{upd}_k : \mathcal{X}_k \times \mathcal{X}'_k \to \mathcal{X}_{k+1}$ **updates** the representation of $v$ given its old representation and the aggregate of the neighbors.

Omit $k$ in the notation of $\text{upd}_k$ and $\text{agg}_k$ when the functions are iteration independent ($\text{upd}_0 = \text{upd}_1 = \dots$).
With these notations message passing can be described as the following representation update:

$$r_{k+1}(v) = \text{upd}_k \left( r_k(v), \text{agg}_k \left( r_k(w) |\ w \in N(v) \right) \right) \qquad \text{(Eq. 12)}$$

In practice, message passing is often used with $\mathcal{X}_0 = \mathcal{X}'_0 = \mathcal{X}_1 = \mathcal{X}'_1 = \cdots = \mathbb{R}^d$ for some fixed $d$. Then the aggregation function $\text{agg} : \mathbb{N}^{IR^d} \to \mathbb{R}^d$ a set of (old) representations to some value from the same domain. The update function is then of the form $\text{upd} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$.

Note that the definition of message passing depends on the multiset of neighbors of a vertex. Thus the order in which those neighbors are (practically) process does not influence the resulting representation of the central vertex. More general, the method is **invariant under isomorphism**:

**THEOREM 4.1 - Message passing is invariant under isomorphism:**

Let $G$ and $H$ be two graphs and $r_0^G$ and $r_0^H$ two vertex representations with fixed aggregator and update functions:

$$r_1^G(v) = \text{upd}\left(r_0^G(v), \text{agg}\left(\{\{r_0^G(w) | \, w \in N(v)\}\}\right)\right)$$

$$r_1^H(v) = \text{upd}\left(r_0^H(v), \text{agg}\left(\{\{r_0^H(w) | \, w \in N(v)\}\}\right)\right)$$

Let $\varphi : V(G) \to V(H)$ be an isomorphism that respects $r_0^G$ and $r_0^H$:

$$r_0^G(v) = r_0^H(\varphi(v)) \qquad \text{(Eq. 13)}$$

Then message passing (the next iteration of the vertex representation) is invariant under isomorphism:

$$\forall k \geq 0 : \qquad r_k^G(v) = r_k^H\big(\varphi(v)\big)$$

**Proof:** The case $k = 0$ is true by definition of $\varphi$ (equation Eq. 13). First lets prove $k = 1$. Let $v \in V(G)$. We will use the preserved neighborhoods under isomorphism

$$w \in N(v) \iff \varphi(w) \in N\big(\varphi(v)\big) \qquad \text{(Eq. 14)}$$

and the following equality of multisets (using again equation Eq. 13):

$$\begin{aligned} \{\{r_0^G(w) | \, w \in N(v)\}\} &= \{\{r_0^H(\varphi(w)) | \, \varphi(w) \in N(\varphi(v))\}\} \\ &= \{\{r_0^G(w') | \, w' \in N(\varphi(v))\}\} \qquad \text{(Eq. 15)} \end{aligned}$$

Now it is:

$$\begin{aligned} r_1^G(v) &= \text{upd}\left(r_0^G(v), \ \text{agg}\left(\{\{r_0^G(w) | \, w \in N(v)\}\}\right)\right) && \text{by definition} \\ &= \text{upd}\left(r_0^H(\varphi(v)), \ \text{agg}\left(\{\{r_0^G(w) | \, w \in N(v)\}\}\right)\right) && \text{using Eq. 15} \\ &= \text{upd}\left(r_0^H(\varphi(v)), \ \text{agg}\left(\{\{r_0^H(w') | \, w' \in N(\varphi(v))\}\}\right)\right) && \text{using Eq. 14} \\ &= r_1^H(\varphi(v)) && \text{by definition} \end{aligned}$$

Now lets prove the statement for $k > 1$. □

We have just shown that the vertex representations computed by message passing do not depend on the order of the vertices. Note that instead of a multiset of neighbor representations, a set would work to.
Also note, that in iteration $k$ the representation of $v$ is determined by the (intermediate) representations of all vertices of distance at most $k$ from $v$, since more the neighborhood of the next iteration depends on its neighbors of the previous one.

### 1.4.1 Aggregator functions

Common choices for aggregator functions are:

- counts,
- means, sums, products,
- histograms,
- hashes and
- Neural Networks.

Lets consider these aggregator functions one by one.

#### Neighbor count aggregator

Consider the **neighbor count aggregator**. Since the graph structure does not change, the multiset of neighbor representations of any vertex $v$ has constant size in each iteration. Thus the aggregator will, for each vertex, produce a **constant value** over all iterations of message passing. This is rather trivial.

#### Means, sums and products aggregator

One can also generalize the **means**, **sums** and **products aggregator** by defining a binary operation that is commutative and associative $\circ : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$. For $X = \{\{x_1, x_2, \ldots, x_{|X|}\}\}$

$$\bigcirc_{x \in X} x = x_1 \circ x_2 \circ \cdots \circ x_{|X|}$$

is a well defined aggregator.

### Histogram aggregator

Lets take a closer look at **histogram aggregators**. Let $\mathcal{X}$ be a set and $B_1, B_2, \ldots, B_k \subseteq \mathcal{X}$ be bins. Then for $X \in \mathbb{N}^{\mathcal{X}}$ and $B_i$ we can define

$$\text{count}_{B_i}(X) := |\{\{x \in X| \; x \in B_i\}\}|$$

and $\text{hist}_{B_1,\ldots,B_k} : \mathcal{X} \to \mathbb{N}^k$ as

$$\text{hist}_{B_1,\ldots,B_k}(X) := \big(\text{count}_{B_1}(X), \ldots, \text{count}_{B_k}(X)\big)$$

Note that the histograms generalize multisets, since a multiset is a histogram where each bin contains a single object. One can also define **weighted versions** where one sums over weights of objects in a bin.
While histograms are very useful, whey are not straightforwardly applicable to message passing.

### Hash aggregator

We can define an aggregator that assigns each multiset an (ideally unique) identifier (hash). Abstractly, we only require that $\# : \mathbb{N}^{\mathcal{X}} \to \mathcal{X}'$ is a function, i.e. that the same multiset will always be mapped to the same value. If such a function is injective it is called a **perfect hash function**. There are multiple ways of implementing this in practice.

### Neural network aggregator

We can use any Neural Network $f_\phi : \mathbb{N}^{\mathbb{R}^d} \to \mathbb{R}^{d'}$ that accepts a set of representations as input and produces an output. In the simplest case, one can use a Perceptron with indegree equal to the maximum degree of the graph times $d$. One has to be careful to have a permutation invariant function, though.

## 1.4.2 Update functions

The update function $\text{upd} : \mathcal{X}_k \times \mathcal{X}'_k \to \mathcal{X}_{k+1}$ updates the representation of $v$ given its old representation and the aggregate of the neighbors.
Again, in principle, one can use any function. It does not have to be commutative, and in fact it will likely lose information if it is, since it then no longer

distinguishes between the representation of the current vertex and its neighborhood.

Standard update functions are:

- weighted sums and products of the results (for numerical data) $(+_{\alpha,\beta} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ like $+_{\alpha,\beta}(x, y) := \alpha \cdot x + \beta \cdot y$ or $\cdot_\alpha : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ like $\cdot_\alpha(x, y) := \alpha \cdot x \cdot y)$,
- Neural Networks (trainable functions),
- hash functions,
- …

### 1.4.3 Weisfeiler Lehman Relabeling Algorithm

**Definition 4.2: Weisfeiler Lehman Label Propagation Scheme**

The **Weisfeiler Lehman Label Propagation Scheme** is defined as

$$r^{\mathrm{WL}}_{k+1}(v) = \#_k\Big(r^{\mathrm{WL}}_k(v),\ \mathrm{id}_k\big(\{\!\{r^{\mathrm{WL}}_k(w) \mid w \in N(v)\}\!\}\big)\Big) \qquad \text{(Eq. 16)}$$

Where

$r^{\mathrm{WL}}_0 : V(G) \to \mathcal{X}_0$ maps to a discrete space,

$\mathrm{id}_k : \mathbb{N}^{X_k} \to \mathbb{N}^{X_k}$ is the identity function and

$\#_k : \mathcal{X}_k \times \mathbb{N}^{\mathcal{X}_k} \to \mathcal{X}_{k+1}$ is a perfect hash function.

Recall, that by definition the hash values in iteration $k$ encode the $k$-hop neighborhood of the vertex it has been assigned to.

**THEOREM 4.2 - Isomorphy invariance of Weisfeiler and Lehman's vertex representation[1]:**

Let $G, H$ be two graphs with vertex label functions $r_0^{\mathrm{WL},G}, r_0^{\mathrm{WL},H}$. If $G$ and $H$ are isomorphic (respecting $r_0^{\mathrm{WL},G}$, $r_0^{\mathrm{WL},H}$) then

$$\forall k \geq 0 : \qquad \{\{r_k^{\mathrm{WL},G}(v) \mid v \in V(G)\}\} = \{\{r_k^{\mathrm{WL},H}(v) \mid v \in V(H)\}\}$$

This means the vertex representation can reduce the graph isomorphy to multiset equality.

**Proof:** The proof follows directly from the invariance theorem (theorem 4.1) for the message passing framework: For an isomorphism $\varphi : V(G) \to V(H)$ we have

$$\forall v \in V(G) : \qquad r_{k+1}^{\mathrm{WL},G}(v) = r_{k+1}^{\mathrm{WL},H}(\varphi(v))$$

Hence, the multisets of vertex representations must be equal, as well, as $\varphi$ is a bijective function. □

Note however that the other direction is true. There are non-isomorphic graphs which can have identical label histograms, see Figure 1.5 for example.

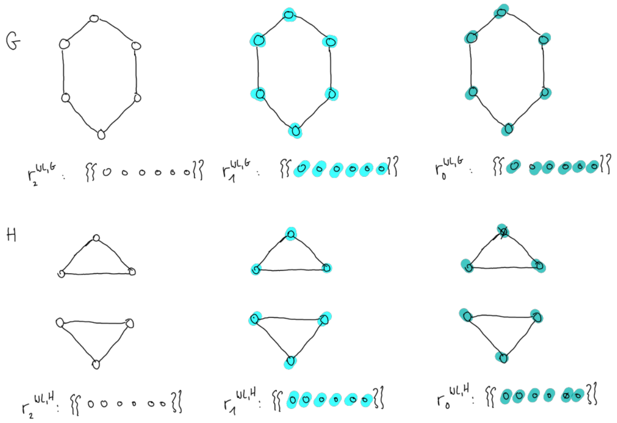

G
$r_2^{WL,G}$ : {{o o o o o o}}
$r_1^{WL,G}$ : {{o o o o o o}}
$r_0^{WL,G}$ : {{o o o o o o}}
H
$r_2^{WL,H}$: {{o o o o o o}}
$r_1^{WL,H}$: {{o o o o o o}}
$r_0^{WL,H}$: {{o o o o o o}}

Figure 1.5 – Example of non-isomorphic graphs with identical label histograms.

This result allows us to test if two graphs $G$ and $H$ are not isomorphic. Therefore simply compute the multisets $\{\{r_k^{\mathrm{WL},G}(v) \mid v \in V(G)\}\}$ and $\{\{r_k^{\mathrm{WL},H}(v) \mid v \in V(H)\}\}$.

If these multisets differ for any $0 \leq k \leq \max\left(|V(G)|,\ |V(H)|\right)$ the graphs $G$ and $H$ are not isomorphic.
Note that if the two multisets are identical, the graphs might be isomorphic or not!

**Efficient computation of the WL labels**

Recall the Weisfeiler Lehman labeling scheme presented in equation Eq. 16:

$$r_{k+1}^{\mathrm{WL}}(v) = \#_k\left(r_k^{\mathrm{WL}}(v),\ \mathrm{id}_k\left(\{\{r_k^{\mathrm{WL}}(w) \mid w \in N(v)\}\}\right)\right)$$

To compute it efficiently, we need a fast way to hash multisets $X$ from $\mathcal{X}$. Let $<$ be a total order on $\mathcal{X}$, then sorting all elements in $X$ according to $<$ returns a unique results. Thus we can compute a canonical string of symbols for each multiset. This would already be a perfect hash function.
With a canonical string of a neighbor label multiset, we can now build a lookup table (on the fly) that assigns each pair (central vertex label and canonical string of neighbor labels) a unique (integer) id.

### 1.4.4 Message passing on GNNs

We would like to train a GNN to implement the message passing, i.a. compute the update and aggregation function, which we have seen already in equation Eq. 12

$$r_{k+1}(v) = \mathrm{upd}_k\left(r_k(v), \mathrm{agg}_k\left(r_k(w) \mid w \in N(v)\right)\right)$$

In its oldest form, a GNN layer is defined for all $v \in V(G)$ as

$$r_{k+1}(v) = \sigma\left(W_k^{\mathrm{self}} r_k(v)\ +\ W_k^{\mathrm{neigh}} \sum_{w \in N(v)} r_k(w) + b_k\right) \qquad \text{(Eq. 17)}$$

where

- $W_k^{\mathrm{self}}, W_k^{\mathrm{neigh}} \in \mathbb{R}^{d_k \times d_{k+1}}$ are trainable parameter matrices,
- $\sigma : \mathbb{R}^{d_k+1} \to \mathbb{R}^{d_k+1}$ is an element-wise non-linear function (e.g. tanh, ReLU or sigmoid) and
- $b \in \mathbb{R}^{d_{k+1}}$ denotes trainable bias parameters[8].

For now, lets assume that the weights and biases are chose appropriately and focus on the forward pass.

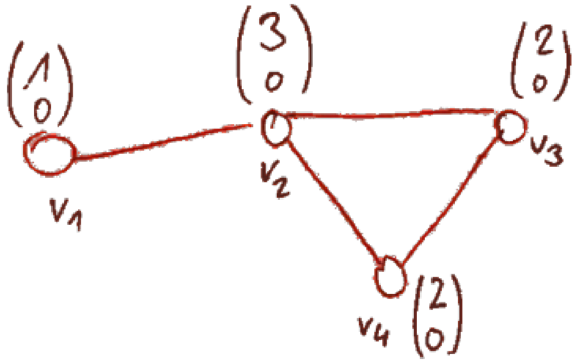
Let use $\theta_k$ to denote *trainable weights in iteration* $k$ of the GNN. We will interpret this as a flattened vector, i.e. $\theta_k \in \mathbb{R}^{2d_k d_{k+1}+d_{k+1}}$.

Lets identify the update and aggregator function. The latter is trivial (and ensures permutation invariance):

$$\mathrm{agg}_k\left(\{\{r_k(w) \mid w \in N(v)\}\}\right) = \sum_{w\in N(v)} r_k(v)$$

$$\mathrm{upd}_k = W_k^{\mathrm{self}} r_k(v) + W_k^{\mathrm{neigh}}\,\mathrm{agg}_k(\cdot) + b_{k+1}$$

**Example - Message Passing on a GNN** Lets compute the next two representations for this example graph with a simple GNN:



$\begin{pmatrix}1\\0\end{pmatrix}$
$\begin{pmatrix}3\\0\end{pmatrix}$
$\begin{pmatrix}2\\0\end{pmatrix}$
$v_1$
$v_2$
$v_3$
$v_4$
$\begin{pmatrix}2\\0\end{pmatrix}$

Figure 1.6

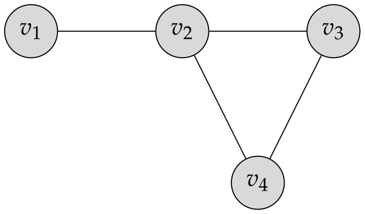


Figure 1.7 – Example for the closeness centrality measure

Initialize the GNN with

- $\sigma = \mathrm{id}$
- $b_i = \begin{bmatrix}0\\0\end{bmatrix}$
- $W_i^{\mathrm{self}} = \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}$
- $W_i^{\mathrm{neigh}} = \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$

We compute:

$$
\begin{aligned}
r_1(v_1) &= W_0^{\text{self}} r_0(v_1) + W_0^{\text{neigh}} \Big( r_0(v_2) \Big) + b_i \\
&= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} & &= \begin{bmatrix} 4 \\ 3 \end{bmatrix} \\
r_1(v_2) &= W_0^{\text{self}} r_0(v_2) + W_0^{\text{neigh}} \Big( r_0(v_1) + r_0(v_3) + r_0(v_4) \Big) + b_i \\
&= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \Big( \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} \Big) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 5 \\ 5 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} & &= \begin{bmatrix} 8 \\ 5 \end{bmatrix} \\
r_1(v_3) &= W_0^{\text{self}} r_0(v_3) + W_0^{\text{neigh}} \Big( r_0(v_2) + r_0(v_4) \Big) + b_i \\
&= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \Big( \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} \Big) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 5 \\ 5 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} & &= \begin{bmatrix} 7 \\ 5 \end{bmatrix} \\
r_1(v_4) &= r_1(v_3) & &= \begin{bmatrix} 7 \\ 5 \end{bmatrix}
\end{aligned}
$$

For the next iteration it is analogously:

$$\begin{aligned}
r_2(v_1) &= \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}4\\3\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\left(\begin{bmatrix}8\\5\end{bmatrix}\right) + \begin{bmatrix}0\\0\end{bmatrix} \\
&= \begin{bmatrix}7\\3\end{bmatrix} + \begin{bmatrix}13\\8\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix} && = \begin{bmatrix}20\\11\end{bmatrix} \\
r_2(v_2) &= \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}8\\5\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\left(\begin{bmatrix}4\\3\end{bmatrix} + \begin{bmatrix}7\\5\end{bmatrix} + \begin{bmatrix}7\\5\end{bmatrix}\right) + \begin{bmatrix}0\\0\end{bmatrix} \\
&= \begin{bmatrix}13\\5\end{bmatrix} + \begin{bmatrix}31\\18\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix} && = \begin{bmatrix}44\\23\end{bmatrix} \\
r_2(v_3) &= \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}7\\5\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\left(\begin{bmatrix}7\\5\end{bmatrix} + \begin{bmatrix}8\\5\end{bmatrix}\right) + \begin{bmatrix}0\\0\end{bmatrix} \\
&= \begin{bmatrix}12\\5\end{bmatrix} + \begin{bmatrix}25\\15\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix} && = \begin{bmatrix}37\\20\end{bmatrix} \\
r_2(v_4) &= r_2(v_3) && = \begin{bmatrix}37\\20\end{bmatrix}
\end{aligned}$$

Note that the numbers in this example tend to grow large rather quickly. In particular, one might expect, that the representations of high degree vertices will contain large values, with a large difference to the values of representations of low degree vertices.

This might lead to numerical instabilities and difficulties in optimization of the loss function.

To prevent this one may swap the sum aggregator with the *mean aggregator*. This also limits the influence of individual vertices in the training.

Another variation is to use the *symmetric normalization* aggregator:

$$\operatorname{agg}_k(\cdot) = \sum_{w \in N(v)} \frac{r_k(w)}{\sqrt{|N(v)|\,|N(w)|}}$$

### 1.4.5 Message passing on GCNs

A very popular variant of GNNs are **Graph Convolutional Neural Networks** (**GCNs**)[11]. These use symmetric normalization and do not distinguish between representations of a vertex $v$ and its neighbors $w \in N(v)$. Thus by construction, they are likely to lose information:

$$r_{k+1}(v) = \mathrm{ReLU}\Big(W_k \sum_{w \in N(G) \cup \{v\}} \frac{r_k(w)}{\sqrt{|N(v)|\,|N(w)|}} + b_k\Big) \qquad \text{(Eq. 18)}$$

Again, $W \in \mathbb{R}^{d_k \times d_{k+1}}$ and $b_k \in \mathbb{R}^{d_{k+1}}$ are trainable parameters.
Note that symmetric normalization (partially) inhibits to learn from the degree of a vertex.

### 1.4.6 Message passing on GINs

Another variant adds a multiple layer trainable Neural Network in each message passing step. These are called **Graph Isomorphism Networks** (**GINs**)[12]. Let $NN_{\theta_k}$ be a (multilayer) Neural Network (with trainable weights and elementwise non-linearities). Let $\epsilon_k \in \mathbb{R}$ be a trainable weights. Then the update step of a GINs, using these parameters, can be described as:

$$r_{k+1}(v) = \mathrm{NN}_{\theta_k}\left((1+\epsilon_k) r_k(v) + \sum_{w \in N(v)} r_k(w) + b_k\right) \qquad \text{(Eq. 19)}$$

There are a lot of freedoms when configuring GINs. Common hyper-parameters that can be learned are:

- The number of layers in the MLP (often only one or two hidden layers).
- The number of units per hidden layer (often 16 or 32 units).
- The non-linearity function (often ReLU).

### 1.4.7 Training GNNs

#### Training GNNs for vertex classification

Assume that we are in a supervised vertex classification setting. Thus for each vertex $v \in V(G)$ we want to predict a discrete class $y(v) \in C$ for some finite set of classes $C = \{c_1, \ldots, c_d\}$.
We are given a finite set of vertices $V \subseteq V(G)$ with known labels $\{y(v) |\, v \in V\}$. As usual we can use (stochastic) gradient descent, backpropagation and a differentiable loss function to train a suitable GNN. Neural Networks for classification tasks are often trained with the **negative log likelihood** loss functions

of a **softmax classification**:

$$L_{G,y}(h_\theta) = \sum_{v \in V} -\log\left(\vec{y}(v)^\intercal \operatorname{softmax}\left(r(v)\right)\right) \tag{Eq. 20}$$

where $\vec{y}(v) \in \mathbf{1}^d$ is a *one-hot vector* encoding the class of $v$. That is for $y(v) = c_i$ have the $i$-th unit vector as the one-hot encoding $\vec{y}(v) = e_i$. $r(v) \in \mathbb{R}^d$ is the (last hidden) representation of $v$.

**Definition 4.3: Softmax function**

The **softmax function**

$$\operatorname{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{d} \exp(x_j)}$$

maps a $d$-dimensional vector $x \in \mathbb{R}^d$ to a $d$-dimensional vector $\hat{y}$ with

$$\sum_{j=1}^{d} \hat{y}_i = 1$$

Hence, the entries of the output $\hat{y}$ can be interpreted as probabilities to belong to a certain class of $d$ classes.
In most cases, the dimension with the largest entry will be mapped to a value close to one - and the rest respectively close to zero (hence the name).

Recall that message passing Neural Networks compute some representation $r_k(v) \in \mathbb{R}^d d_k$ for some $d_k$ - that is one representation for all $k$ iterations over all $d$ possible representation values. $d_k$ usually depends on the dimensionality of the input representation $r_0 : V(G) \to \mathbb{R}^{d_0}$ and $d_k$ is not equal to the number of classes in the vertex classification problem $d = |C|$.

Hence, we have to transform the message passing representations to representations that are suitable as input to the softmax function. To do this, we can use another Neural Network $NN_{\theta'} : \mathbb{R}^{d_k} \to \mathbb{R}^d$ (with learnable parameters $\theta'$). Then we can minimize the loss function

$$L_{G,y}(h_{k,\theta,\theta'}) = \sum_{v \in V} -\log\left(\vec{y}(v)^\intercal \operatorname{softmax}\left(\mathrm{NN}_{\theta'}(r(v))\right)\right) \tag{Eq. 21}$$

Lets take a closer look at the parameters. It is:

$$v \overset{\text{given}}{\mapsto} r_0(v) \overset{\text{upd}_{\theta_0}}{\mapsto} r_1(v) \overset{\text{upd}_{\theta_1}}{\mapsto} \cdots \overset{\text{upd}_{\theta_{k-1}}}{\mapsto} r_k(v) \overset{\text{NN}_{\theta'}}{\mapsto} x(v) \overset{\text{softmax}}{\mapsto} \hat{y}(v)$$

Since all functions between the representations $r_0(v)$ and $\hat{y}(v)$ are differentiable with respect to the parameters $\theta_0, \ldots, \theta_{k-1}, \theta'$ ne can use the chain rule to automatically compute the gradient with respect to them:

$$\nabla L_{G,y}(h_{k,\theta_0,\ldots,\theta_{k-1},\theta'}) = \nabla\Big( -\log \circ \text{softmax} \circ \text{NN}_{\theta'} \circ \text{upd}_{\theta_{k-1}} \circ \cdots \circ \text{upd}_{\theta_0} \Big) \tag{Eq. 22}$$

The weights are updated according to some chosen strategy.

**Algorithm 6** GNN - Vertex representation - ONE EPOCH

**Input:** a training set $V \subseteq V(G)$ with known classes $y : V \to C$,
learning rage $\gamma$,
a Neural Network to learn the vertex representation, given a softmax $\text{NN}_{\theta'}$,
number of message passing steps $k$ and
the initial vertex representation $r_0(v)$ for all vertices $v \in V(G)$.
**Output:** updated weights $\theta_0, \ldots, \theta_{k-1}, \theta'$.

```
Run $k$ iterations of the GNN message passing for all $v \in V(G)$
Apply $\text{NN}_{\theta'}$ to the resulting representations $r_k(v)$ for all $v \in V(G)$
Compute $L_{G,y}(h_{k,\theta_0,\ldots,\theta_{k-1},\theta'})$
Update the weights by setting
    $(\theta_0, \ldots, \theta_{k-1}, \theta') = (\theta_0, \ldots, \theta_{k-1}, \theta') - \gamma \nabla L_{G,y}(h_{k,\theta_0,\ldots,\theta_{k-1},\theta'})$
```

Note that this algorithm processes the whole training dataset at once (i.e. with gradient descent).

Usually NNs are trained one mini-batch at a time (i.e. with stochastic gradient descent). To apply this to the GNN, we need to compute a (intermediate) representation for additional vertices (outside of the mini-batch).

**Algorithm 7** GNN - Vertex representation - ONE EPOCH - Mini-batch

**Input:** a training **batch** $B \subseteq V \subseteq V(G)$ with known classes $y : V \to C$,
learning rage $\gamma$,
a Neural Network to learn the vertex representation, given a softmax $\text{NN}_{\theta'}$,
number of message passing steps $k$ and
the initial vertex representation $r_0(v)$ for all vertices $v \in V(G)$.
**Output:** updated weights $\theta_0, \ldots, \theta_{k-1}, \theta'$.

```
for $i = 1, \ldots, k$ do
    Compute the $i$-hop neighborhoods $N_i(v)$ for all $v \in B$
for $i = 1, \ldots, k$ do
    Compute $r_i(w)$ for all $w \in N_{k-i}(v)$ for all $v \in B$
Apply $\text{NN}_{\theta'}$ to the resulting representations $r_k(v)$ for all $v \in B$
Compute $L_{G,y}(h_{k,\theta_0,\ldots,\theta_{k-1},\theta'})$
Update the weights by setting

    $(\theta_0, \ldots, \theta_{k-1}, \theta') = (\theta_0, \ldots, \theta_{k-1}, \theta') - \gamma \nabla L_{G,y}(h_{k,\theta_0,\ldots,\theta_{k-1},\theta'})$
```

Note that this algorithm processes the whole training dataset at once (i.e. with gradient descent).

Often, **weight sharing** is applied between GNN iterations. That is, we use the same functions upd and agg for each iteration $k$ of the message passing framework. This means that we set $\theta_0 = \theta_1 = \cdots = \theta_{k-1}$ and update these hyperparameters simultaneously from the different iterations during backpropagation. This can be beneficial but also a drawback for learning.

One may ask, if these algorithms 6 and 7 follow the principle of *supervised learning*.

**Definition 4.4: Supervised learning**

The **supervised learning** principle is fulfilled, it the learning algorithm works based on the following in- and outputs:

**Input:** a domain set $\mathcal{X}$,
a target domain set $\mathcal{Y}$,
a *hypothesis class* $\mathcal{H} \subseteq \{h | \ h : \mathcal{X} \to \mathcal{Y}\}$,

a finite set $X \subseteq \mathcal{X}$ and known target values $y(x)$ for all $x \in X$ and
a loss function $L_y : \mathcal{H} \to \mathbb{R}$.

**Output:** $\operatorname{argmin}_{h \in \mathcal{H}} L_y(h)$.

Now lets apply this to the vertex representation learning task. We have:

**Input:** a graph $G$ with labeling functions $f_V$ and $f_E$,
a *hypothesis class* $\mathcal{H} \subseteq \{h | \, h : V(G) \to \mathcal{Y}\}$,
a finite set $V' \subseteq V(G)$ and known target values $y(v)$ for all $v \in V'$ and
a loss function $L_{G,y} : \mathcal{H} \to \mathbb{R}$.

Now consider the aggregation step in the message passing framework (Eq. 12)

$$r_{k+1}(v) = \operatorname{upd}_k \Big( r_k(v), \operatorname{agg}_k \left( r_k(w) | \; w \in N(v) \right) \Big)$$

What happens when $w \in N(v)$ is not part of $V'$? The update operation requires information from objects that are *not* part of the training set in order to construct the (novel) vertex representations. On top of that, it is likely information from an object in the test or validation set. Thus actually, we require the full graph $G$ as input to our training. We cannot (by assumption) learn functions over vertices that were not part of $V(G)$ at the time of training. This is called **transductive learning** or **transductive inference**. We are trying to predict a target feature for a specifically known finite set of possible test instances $V(G) \backslash V'$.

# Chapter 2

# Transactional Graph Learning

Up to now, we have tried to represent and learn from vertices in a single graph. In this chapter, we will consider learning functions over whole graphs. Since the term Graph Learning is overloaded with meaning, we will call this task **Transactional Graph Learning**. We will call individual graphs (**graph**) **transactions**. The objective of Transactional Graph Learning is to determine whether a graph belongs to a **graph class**, which is a (typically finite) set of graphs, or not.

Examples of graph classes are:

- Graphs
- Forests (cycle free graphs)
- Trees (connected and cycle free graphs)
- Molecular graphs
  (Graphs that describe the structural formulas of valid molecules.)
-

Similar to definition 0.4 lets define additional data on graphs, their vertices and edges for a whole graph class.

**Definition 0.1: Graph class attributes and labels**

Given a graph class $\mathcal{G}$ containing transactions (graphs). The vertices and edges in these transactions may be labeled (or attributed).

We formalize this using appropriate **feature spaces** $\mathcal{X}$, $\mathcal{X}'$ and $\mathcal{X}''$:

$$f_G : \mathcal{G} \to \mathcal{X}, \qquad f_V : \bigcup_{G \in \mathcal{G}} V(G) \to \mathcal{X}', \qquad f_E : \bigcup_{G \in \mathcal{G}} E(G) \to \mathcal{X}''$$

Note that in one learning task, we assume that all graphs in a graph class are labeled with values from the same feature spaces and these values are meaningfully comparable.

**Definition 0.2: Transactional Graph Learning**

**Input:** a class of graphs $\mathcal{G}$,
a target domain set $\mathcal{Y}$,
a *hypothesis class* $\mathcal{H} \subseteq \{h |\, h : \mathcal{G} \to \mathcal{Y}\}$,
a finite set $D \subseteq \mathcal{G}$ and known target values $y(G)$ for all $G \in D$ and
a loss function $L_y : \mathcal{H} \to \mathbb{R}$.
**Output:** $\text{argmin}_{h \in \mathcal{H}} L_y(h)$.
Note that in Transactional Graph Learning one usually assumes that **isomorphic graphs behave identically**.

## 2.1 Multiset graph representations

In order to use standard machine learning algorithms, we are in need of a vector representation for graphs $r : \mathcal{G} \to \mathbb{R}^d$. The main idea is to use the vertex representations that are discussed in the previous chapter the represent a whole graph $G$.
A trivial example is the list of all occurring vertex degrees:

$$\boldsymbol{r}(G) := \bigoplus_{v \in V(G)} r(v) \qquad \in \mathbb{R}^{d|V(G)|}$$

Of course this is permutation sensitive. One may get around this by using multisets or sorted lists of vertex degrees. More challenging is the concern, that graphs can have different amounts of vertices. Hence this representation for $r_G \in \mathbb{R}^{d|V(G)|}$ and $r_H \in \mathbb{R}^{d|V(H)|}$ cannot be used to define a function $r : \mathcal{G} \to \mathbb{R}^{d'}$ with fixed $d'$.
Thus in general it is better to use a common feature space $\mathcal{X}$ and define the graph representation $r_{\text{MS}}$ as invariant multisets. We will use some vertex rep-

resentation (e.g. vertex degree) $r : V(G) \to \mathcal{X}$ and set:

$$r_{\text{MS}}(G) = \{\!\{ r(v) |\ v \in V(G) \}\!\} \qquad \in \mathbb{N}^{|\mathcal{X}|} \tag{Eq. 1}$$

Note that $\mathcal{X}$ can have a practically infeasible number of elements (possibly infinitely many). Thus such **multiset graph representations** are useful for *discrete* vertex representations. For example vertex degrees, vertex degeneracies, triangle counts, other individual graphlet counts or Weisfeiler-Lehman vertex labels.

For all these examples, the multiset representation can be used for *overlap based similarities*.

We can represent multisets as vectors with one dimension per element of $\mathcal{X}$ that stores a count. When dealing with finite transactional graph databases, we know that

$$|\{\!\{ r(v) |\ v \in V(G) \}\!\}| = |V(G)|$$

That means that there are always only finitely many entries that are nonzero for each graph $G \in \mathcal{G}$. That means we can use **sparse vectors** in practice to efficiently store the multiset representation $r_{\text{MS}}(G) \in \mathbb{N}^{|\mathcal{X}|}$ in our finite memory, even if $|\mathcal{X}| = \infty$.

**THEOREM 1.1 - Multiset graph representation respects isomorphism:**

Let $G, H \in \mathcal{G}$ be two graphs and $\varphi : V(G) \to V(H)$ an isomorphism that respects their vertex features $f_V : \bigcup_{G \in \mathcal{G}} V(G) \to \mathcal{X}'$. Let $r_{\text{MS}} : \mathcal{G} \to \mathbb{N}^{\mathcal{X}'}$ be a multiset graph representation of these features:

$$r_{\text{MS}}(G) = \{\!\{ f_V(v) |\ v \in V(G) \}\!\}$$

Then

$$r_{\text{MS}}(G) = r_{\text{MS}}(H)$$

**Proof:** Exercise. □

We say the multiset graph representation is *invariant under isomorphism*. They may or may not map non-isomorphic graphs to different values.

### 2.1.1 Histogram graph representations

Recall that we have earlier discussed *histogram aggregators*. Let $\mathcal{X}$ be a set and $B_1, B_2, \ldots, B_k \subseteq \mathcal{X}$ be bins. Then, for $X \in \mathbb{N}^{\mathcal{X}}$ and $B_i$ we can define

$$\text{count}_{B_i}(X) = |\{\{x \in X|\ x \in B_i\}\}|$$

and a histogram as a mapping $\text{hist}_{B_1,\ldots,B_k} : \mathbb{N}^{\mathcal{X}} \to \mathbb{N}^k$ as

$$\text{hist}_{B_1,\ldots,B_k}(X) = \left(\text{count}_{B_1}(X), \ldots, \text{count}_{B_k}(X)\right)$$

Note that histograms generalize multisets in the sense that *a multiset is a histogram where each bin contains a single object*. One can define a histogram aggregator that counts the occurrences of each individual object in (finite or countably infinite) $\mathcal{X}$. One can also define weighted versions where one sums over the weights of objects in a bin.

Histograms can be seen as discrete distributions. During their definition one can influence the dimensionality of the resulting graph representation.

**THEOREM 1.2 - Histogram graph representation respects multiset graph representations:**

Let $G, H \in \mathcal{G}$ be two graphs and $\varphi : V(G) \to V(H)$ an isomorphism that respects their vertex features $f_V : \bigcup_{G \in \mathcal{G}} V(G) \to \mathcal{X}'$. Let $r_{\text{MS}} : \mathcal{G} \to \mathbb{N}^{\mathcal{X}'}$ be a multiset graph representation of these features:

$$r_{\text{MS}}(G) = \{\{f_V(v)|\ v \in V(G)\}\}$$

Let $\text{hist}_{B_1,\ldots,B_k} : \mathbb{N}^{\mathcal{X}'} \to \mathbb{N}^k$ be a histogram representation:

$$\text{hist}_{B_1,\ldots,B_k}(X) = \left(\text{count}_{B_1}(X), \ldots, \text{count}_{B_k}(X)\right)$$

Then

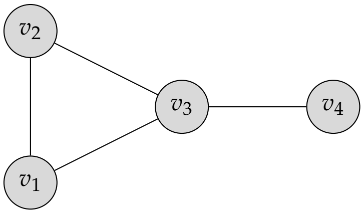
$$\text{hist}_{B_1,\ldots,B_k}\left(r_{\text{MS}}(G)\right) = \text{hist}_{B_1,\ldots,B_k}\left(r_{\text{MS}}(H)\right)$$

**Proof:** From theorem 1.1, we know that the multiset representations of isomorphic graphs are identical. Since $\text{hist}_{B_1,\ldots,B_k}$ is a function, we know that it hence must map the identical images of $G$ and $H$ to the same value:

$$r_{\mathrm{MS}}(G) = r_{\mathrm{MS}}(H)$$

□

Note that isomorphism respects the original labeling. Hence, graphs with different original label multisets can never be isomorphic. One may need to take extra care with respect to original labels in order to not have the representation denote an isomorphism where there is no one to be found.



| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $\sum$ |
|---|---|---|---|---|---|
| edge | 2 | 3 | 2 | 1 | 8 |
| 3-path | 1 | 2 | 1 | 2 | 6 |
| triangle | 1 | 1 | 1 | 0 | 3 |

Table 2.1 – Exapmle: Graphlet counting (of induced subgraphs).

## 2.2 GNNs for Transactional Graph Learning

Given a set of graphs for training, we can extend our neural network from computing vertex representations to graph representations. After a fixed number of message passing layers to compute vertex representations,we add a final aggregator layer that aggregates all vertex representations in the graph. Conceptually, for the last step we add a “hypervertex” that is connected to all vertices in the graph and aggregate over its neighbors and then update. The resulting **hypervertex representation** becomes the graph representation.

**Algorithm 8** GNN - Transactional Graph Learning - ONE EPOCH

```
Input:  a training set $D \subseteq \mathcal{G}$ with known classes $y : D \to C$,
        learning rage $\gamma$,
        a Neural Network to learn the graph representation, given a softmax $\text{NN}_{\theta'}$,
        number of message passing steps $k$ and
        the initial vertex representation $r_0(v)$ for all vertices $v \in V(G)$.
Output: updated weights $\theta_0, \dots, \theta_{k-1}, \theta_{\text{graph}}, \theta'$.

for $G \in D$ do
    Run $k$ iterations of your GNN message passing for all $v \in V(G)$
    Aggregate the vertex representations of all vertices in $G$
    Update the result to obtain a graph representation $r_{\text{graph}}(G)$
Apply $\text{NN}_{\theta'}$ to the resulting representations $r_{\text{graph}}(G)$ for all $G \in \mathcal{G}$
Compute $L_y(h_{k,\theta_0,\dots,\theta_{k-1},\theta_{\text{graph}},\theta'})$
Update the weights by setting

    $(\theta_0, \dots, \theta_{k-1}, \theta') = (\theta_0, \dots, \theta_{k-1}, \theta_{\text{graph}}, \theta') - \gamma \nabla L_{G,y}(h_{k,\theta_0,\dots,\theta_{k-1},\theta_{\text{graph}},\theta'})$
```

## 2.3 Fast Forward Computation of Message Passing GNNs

In the previous lectures (equation Eq. 17), we have formulated the message passing framework on a per vertex basis. If we have to compute all vertex representations, anyway, we can reformulate the message passing NN using *matrix vector operations* as follows:

$$R_{k+1} = \sigma\Big(R_k W_k^{\text{self}} + A_G R_k R_k W_k^{\text{neigh}} + b_k\Big) \tag{Eq. 2}$$

Again $W_k^{\text{self}}, W_k^{\text{neigh}} \in \mathbb{R}^{d_k \times d_{k+1}}$ and $b \in \mathbb{R}^{d_{k+1}}$ are trainable weights (matrices). $A_G$ is a (sparse) adjacency matrix of $G$ and $R_k \in \mathbb{R}^{|V(G)| \times d_k}$ is a row matrix of vertex representations.

## 2.4 Complete representations

Note that with theorem 1.1 we showed that the multiset graph representation is invariant under isomorphism, but may or may not map non-isomorphic graphs to different values. We now consider functions that map given graphs to different values *if and only if* they are not isomorphic:

**Definition 4.1: Complete representation of graph classes**

Let $\mathcal{G}$ be a graph class and $\mathcal{Y}$ be some (feature) space. We call $c : \mathcal{G} \to \mathcal{Y}$ a **complete representation function of** $\mathcal{G}$ if

$$\forall G, H \in \mathcal{G} : \qquad c(G) = c(H) \iff G \text{ is isomorphic to } H$$

and $c(G) = c(H)$ can be verified in *polynomial time* in size $\big(c(G)\big)$ and size $\big(c(H)\big)$.
Sometimes such functions are called **canonical functions** and their images are called **canonical strings** or **canonical labelings**.

Notice, that this property is a prerequisite to compute a (non-negative) distance between graphs. That is, if we cannot learn such functions, we will always have some objects that we cannot distinguish (by their image), although it might be necessary.
Furthermore, isomorphism defines an equivalence relation on any graph class $\mathcal{G}$. We can consider the quotient space $\mathcal{G}_{\text{iso}}$ that contains exactly one representative graph $G_X$ for each subset $X \subseteq \mathcal{G}$ of pairwise isomorphic graphs in $\mathcal{G}$.
Let $f : \mathcal{G} \to \mathcal{Y}$ be invariant under isomorphism. Then this function induces a function $f_{\text{iso}} : \mathcal{G}_{\text{iso}} \to \mathcal{Y}$ with

$$\forall X \subseteq \mathcal{G} : \qquad f_{\text{iso}}(G_X) = f(G_X)$$

If $f$ is a complete representation on $\mathcal{G}$, then the induced $f_{\text{iso}}$ is *injective* on $\mathcal{G}_{\text{iso}}$ (twp images are equal if and only if the inputs are equal).

**Definition 4.2: Expressiveness of functions**

Let $f, f' : \mathcal{G} \to \mathcal{Y}$ be two functions that are invariant under isomorphism.

$f'$ is **less expressive** than $f$, denoted $f' \leq_x f$, if

$$f(G) = f(H) \implies f'(G) = f'(H)$$

$f$ and $f'$ are **equally expressive** if $f \leq_x f'$ and $f' \leq_x f$.
$f'$ is **strictly less expressive** than $f$ (written as $f' <_x f$), if $f' \leq_x f$ and not $f \leq_x f'$.

**Example of a complete representation** Let $\mathcal{G}$ be the class of all unlabeled graphs (i.e. $f_G, f_V, f_E$ are all constant). Let $<_{\text{Adj}}$ be a total order on the adjacency matrices. We assign any graph $G$ its *smallest adjacency matrix among all permutations* of $V(G)$. Then two isomorphic graphs $G$, $H$ will be mapped to identical adjacency matrices.
More specifically, we can realize the total order on the adjacency matrices $<_{\text{Adj}}$ by *lexicographical ordering* the flattened string representation of any adjacency matrix $G$.
While this function indeed is a complete representation, it cannot be computed efficiently (as far as we know). In fact, the naive algorithm for the representation on the previous slides iterates over all permutations of the adjacency matrix and hence takes factorial time.

**THEOREM 4.1 - Complexity of complete representations:**

Computing a complete representation for some graph class $\mathcal{G}$ is Isomorphism-hard.
I.e. if we cannot decide if two graphs in $\mathcal{G}$ are isomorphic in polynomial time, then we cannot compute *any* complete representation function for $\mathcal{G}$ in polynomial time.

**Proof:** Suppose we can compute a complete representation $c : \mathcal{G} \to \mathcal{Y}$ for some suitable $\mathcal{Y}$ in polynomial time in size$(G)$ for all $G \in \mathcal{G}$. Then we can, for two graphs $G, H \in \mathcal{G}$, compute $c(G)$ and $c(H)$ in polynomial time and evaluate $c(G) = c(H)$ in polynomial time.
Hence, we have just found a polynomial time algorithm for checking isomorphism for graphs in $\mathcal{G}$. □

We do not know the complexity of the isomorphism problem for the class of all graphs. Isomorphism [10] and recently computing a complete representation for the class of all graphs[13] was shown to be solvable in **quasipolynomial time**.

Some more restricted graph classes have polynomial time complete representation functions. Various variants exists for trees[7], which allows to compute complete representations for forests, too. Also various graph classes allow for polynomial time algorithms, which in turn make it possible that a complete representation exists.

Note that this does not imply, that one cannot *learn* a complete representation for the class of all graphs. Supervised learning from a fixed finite training dataset can be seen as *constant time pre-processing*, i.e. it takes $\mathcal{O}(1)$ time. Thus, if we could learn a complete representation $c$ that is computable in polynomial time for any graph $G \in \mathcal{G}$, then this would imply a polynomial time algorithm for the isomorphism problem on $\mathcal{G}$.

GNNs promise to learn a suitable graph representation for any learning task. Thus lets explore if we can use them to learn complete representations for the class of all graphs.

Again, the Weisfeiler-Lehman vertex representations come to mind (definition 4.2). But we have the following result:

**THEOREM 4.2 - Incompleteness of Weisfeiler-Lehman representations:**

Weisfeiler-Lehman representations are strictly less expressive on the class of all graphs than compete representations, i.e.

$$R_k^{\mathrm{WL}} <_x c$$

for any $k \in \mathbb{N}$ and any complete representation $c$.

**Proof:** There are a few counterexamples. These are non-isomorphic graphs with identical Weisfeiler-Lehman representations[5]: □

## 2.5 Expressive power of GNNs

Before we present results about the expressiveness of GNNs, lets introduce a definition and a small theorem which we will use in the upcoming proofs:

**Definition 5.1: Strength of functions**

Let $f$ and $f'$ be functions on some domain $\mathcal{X}$. We say $f'$ is **less strong** than $f$ and write $f' \tilde{\leq} f$ if

$$\forall x, y \in \mathcal{X} : \qquad f(x) = f(y) \implies f'(x) = f'(y)$$

Note if $f$ and $f'$ are functions on a graph class $\mathcal{G}$ such that $f$ and $f'$ are invariant under isomorphism and $f' \tilde{\leq} f$, then $f'$ is also less expressive than $f$, i.e. $f' \leq_x f$.

**THEOREM 5.1 - Obedience of weaker functions:**

Let $f'$ and $f$ be two functions on some domain $\mathcal{X}$ and $f' \tilde{\leq} f$. Let $X, Y \subseteq \mathbb{N}^{\mathcal{X}}$ be two multisets. Then

$$\{\{f(x)|\ x \in X\}\} = \{\{f(x)|\ x \in Y\}\} \implies \{\{f'(x)|\ x \in X\}\} = \{\{f'(x)|\ x \in Y\}\}$$

**Proof:** The equality of two multisets $X', Y' \subseteq \mathbb{N}^{\mathcal{X}}$ means (by definition) that there exists a bijective function $b : X' \to Y'$ with $Y' \ni b(x) = x \in X'$.
Thus the equality of multisets $\{\{f(x)|\ x \in X\}\} = \{\{f(x)|\ x \in Y\}\}$ implies that there is such a bijective function $b$ such that

$$\forall x \in X : \qquad f(x) = b\big(f(x)\big)$$

But we also have that $b\big(f(x)\big) = f(y)$ for some $y \in Y$ and hence $f(x) = f(y)$. So we also have a bijection $b' : X \to Y$ such that $f(x) = f\big(b'(x)\big)$.
Now, by assumption $f(x) = f(y)$ implies $f'(x) = f'(y)$ and hence we have $f'(x) = f'\Big(b'(x)\Big)$ for all $x \in X$ and hence also for all $f'(x) \in \{\{f'(x)|x \in X\}\}$.

□

**THEOREM 5.2 - GNNs are at most as expressive as Weisfeiler-Lehman representations[12]:**

Let $\mathcal{G}$ be the class of all graphs and let $R_k : \mathcal{G} \to \mathbb{R}^d$ be a representation of $\mathcal{G}$ computed by a message passing Graph Neural Network $r_k$ with $k$ iterations and some final graph aggregation step $\text{agg}'\Big(\{\{r_k(v)|\ v \in V(G)\}\}\Big)$. Then

$$R_k \leq_x R_k^{\text{WL}}$$

**Proof:** The proof by induction has the following structure: We will show that for $R_k$ defined by $R_k(G) = \text{agg}'\left(\{\!\{r_k(v) |\ v \in V(G)\}\!\}\right)$ with

$$r_k(v) = \text{upd}_k\left(r_{k-1}(v),\ \text{agg}_k\left(\{\!\{r_{k-1} |\ w \in N(v)\}\!\}\right)\right)$$

for any choice of $\text{agg}'$, $\text{agg}_k$ and $\text{upd}_k$ it holds that

$$R_{k-1} \leq_x R_{k-1}^{\text{WL}} \implies R_k \leq_x R_k^{\text{WL}}$$

That means if $R_{k-1}$ is less expressive than $R_{k-1}^{\text{WL}}$, then $R_k$ is less expressive than $R_k^{\text{WL}}$.
We show this by showing that for

$$r_k^{\text{WL}} = \#\left(r_{k-1}^{\text{WL}},\ \text{id}\left(\{\!\{r_{k-1}(w) |\ w \in N(v)\}\!\}\right)\right)$$

for any choice of $\text{agg}_k$ and $\text{upd}_k$ it holds that if

$$r_{k-1}^{\text{WL}}(v) = r_{k-1}^{\text{WL}}(u) \implies r_{k-1}(v) = r_{k-1}(u)$$

then it is

$$r_k^{\text{WL}}(v) = r_k^{\text{WL}}(u) \implies r_k(v) = r_k(u)$$

**Induction start** : Note that we are now talking about vertex representations and not about general $R$ functions defined over graphs. It is $r_0 = r_0^{\text{WL}} = f_V$ and hence

$$r_0^{\text{WL}}(v) = r_0^{\text{WL}}(u) \implies r_0(v) = r_0(u) \qquad \text{i.e., } r_0 \tilde{\leq} r_0^{\text{WL}}$$

**Induction step** : Let $r_{k-1} \tilde{\leq} r_{k-1}^{\text{WL}}$. We will now look at the definition of $r_k^{\text{WL}}(v)$ and see how it is connected to $r_k(v)$. We will show that then $r_k(v) \tilde{\leq} r_k^{\text{WL}}$, i.e.,

$$\forall u, v \in \bigcup_{G \in \mathcal{G}} V(G): \qquad r_k^{\text{WL}}(v) = r_k^{\text{WL}}(u) \implies r_k(v) = r_k(u)$$

By definition $r_k^{\text{WL}}(v) = r_k^{\text{WL}}(u)$ is equivalent to

$$\#\left(r_{k-1}^{\text{WL}}(v),\ \text{id}\left\{\!\!\left\{r_{k-1}^{\text{WL}}(w) |\ w \in N(v)\right\}\!\!\right\}\right) = \#\left(r_{k-1}^{\text{WL}}(u),\ \text{id}\left\{\!\!\left\{r_{k-1}^{\text{WL}}(w) |\ w \in N(u)\right\}\!\!\right\}\right)$$

Since # is injective, this implies

- $r_{k-1}^{\mathrm{WL}}(v) = r_{k-1}^{\mathrm{WL}}(u)$ and

- $\{\!\{ r_{k-1}^{\mathrm{WL}}(w) |\; w \in N(v) \}\!\} = \{\!\{ r_{k-1}^{\mathrm{WL}}(w) |\; w \in N(u) \}\!\}$

By induction assumption we know that

$$r_{k-1}^{\mathrm{WL}}(v) = r_{k-1}^{\mathrm{WL}}(u) \implies r_{k-1}(v) = r_{k-1}(u)$$

Using theorem 5.1 on $r_{k-1} \tilde{\lesssim} r_{k-1}^{\mathrm{WL}}$ with the equality of the multisets in mind (second implication) gives

$$\forall u, v \in \bigcup_{G \in \mathcal{G}} V(G) : \qquad \{\!\{ r_{k-1}(w) |\; w \in N(v) \}\!\} = \{\!\{ r_{k-1}(w) |\; w \in N(u) \}\!\}$$

We now have shown that

$$r_k(v) = \mathrm{upd}_k \left( r_{k-1}(v),\, \mathrm{agg}_k \left( \{\!\{ r_{k-1} |\; w \in N(v) \}\!\} \right) \right) = \mathrm{upd}_k \left( r_{k-1}(v),\, \mathrm{agg}_k \left( \{\!\{ r_{k-1} |\; w \in N(v) \}\!\} \right) \right) = r_k($$

since the function arguments are identical. Thus we have now proven the intermediate claim that $r_k \tilde{r}_k{}^{\mathrm{WL}}$ for any $k$.

Using theorem 5.1 again, on

$$\begin{aligned} & \{\!\{ r_k^{\mathrm{WL}}(v) |\; v \in V(G) \}\!\} = \{\!\{ r_k^{\mathrm{WL}}(u) |\; u \in V(H) \}\!\} \\ \implies \qquad & \{\!\{ r_k^{\mathrm{WL}}(v) |\; v \in V(G) \}\!\} = \{\!\{ r_k^{\mathrm{WL}}(u) |\; u \in V(G) \}\!\} \end{aligned}$$

As the graph level aggregator is a function, this implies

$$\mathrm{agg}' \left( \{\!\{ r_k^{\mathrm{WL}}(v) |\; v \in V(G) \}\!\} \right) = \mathrm{agg}' \left( \{\!\{ r_k^{\mathrm{WL}}(v) |\; v \in V(G) \}\!\} \right)$$

Which is by definition $R_k \leq_x R_k^{\mathrm{WL}}$. □

**THEOREM 5.3 - Weisfeiler-Lehman representations are at most as expressive as GNNs:**

Let $\mathcal{G}$ be the class of all graphs and let $R_k : \mathcal{G} \to \mathbb{R}^d$ be a representation of $\mathcal{G}$ computed by a message passing GNN $r_k$ with $k$ iterations and some final graph aggregation step $\text{agg}'\left(\{\{r_k(v) |\ v \in V(G)\}\}\right)$. Then

$$R_k^{\text{WL}} \leq_x R_k$$

if the following conditions hold:

1. $r_k$ aggregates and updates vertex features with *injective* functions agg and upd in

$$r_{k+1}(v) = \text{upd}_k\left(r_k(v), \text{agg}_k\left(r_k(w) |\ w \in N(v)\right)\right)$$

2. $R_k$'s graph-level aggregation step, which operates on $\{\{r_k(v) |\ v \in V(G)\}\}$ is *injective*.

**Proof:** We have shown that any choice of $\text{agg}_k$ and $\text{upd}_k$ yields a message passing algorithm that is at most as expressive as the message passing algorithm that uses # and id. However, we have only used that # and id are injective functions. Hence the claim follows directly from the proof of theorem 5.2 by setting

$$\text{agg}_k = \text{id} \qquad \text{and} \qquad \text{upd}_k = \#.$$

□

Theorem 5.2 and 5.3 combined imply, that any message passing GNN can be at most as expressive as the Weisfeiler Lehman relabeling function. Note however, that there may exist GNNs that are equally expressive. We will see, that GINs get quite close.

### 2.5.1 The GIN theorem

**THEOREM 5.4 - The GIN theorem:**
Assume that $r_0 : \mathcal{G} \to \mathcal{X}$ maps to a countable space $\mathcal{X}$. There exists a function $f : \mathcal{X} \to \mathbb{R}^d$ such that for infinitely many choices of $\epsilon$ it is

$$\mathrm{upd}(c, X) = (1 + \epsilon) f(c) + \sum_{x \in X} f(x)$$

is unique for each air $x \in \mathcal{X}$, $X \in \mathbb{N}^{\mathcal{X}}$.

The requirement for countable domains makes this suitable for discretely labeled initial vertex representations $r_0$. We can use one-hot encoded representations of the initial labels. Summing such vectors results in an injective function over multisets.

We can use a NN to learn the function $f : \mathcal{X} \to \mathbb{R}^d$. Since multilayer perceprtons (with non-linearities) are universal function approximators[4] we can (theoretically) learn $f$ by replacing our update function $\mathrm{upd}_k$ with a sufficiently powerful multilayer perceptron $\mathrm{MLP}_\theta$:

$$r_{k+1}(v) = f\big(r_k(v)\big) = \mathrm{MLP}_\theta \left( (1 + \epsilon) r_k(v) + \sum_{w \in N(v)} r_k(w) \right)$$

Note that this is exactly the definition of a GIN[12] (equation Eq. 19).

# Chapter 3

# GOTO END

# Chapter 4

# Exercises

## 4.1 Sheet 0 - Python

## 4.2 Sheet 1

### 4.2.1 Assignment 1a - Bias of an estimator

…

**Solution:**

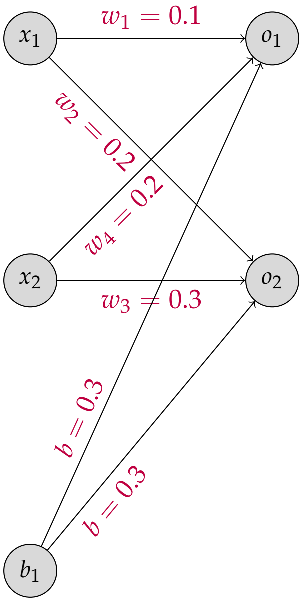


Figure 4.1 – Multi-Layer Perceptron

| $n$ | $x_1$ | $x_2$ | $p_1$ | $p_2$ |
|---|---|---|---|---|
| 1 | 0.1 | 0.4 | 0.1 | 0.9 |
| 2 | 0.8 | 0.2 | 0.95 | 0.05 |
| 3 | 0.6 | 0.5 | 0.4 | 0.6 |
| 4 | 0.3 | 0.9 | 0.75 | 0.25 |
| 5 | 0.3 | 0.5 | 0.9 | 0.1 |

Table 4.1 – Training batch

**Algorithm 9** Generative Adversarial Networks (GAN)

```
Input:  a discriminator function $D$,
        a generator function $G$,
        noise samples $Z$,
        true samples $X$ (with an underlying data generation distribution $\mathbb{P}_{\text{data}}[X]$),
        desired size of mini-batches $m$,
        a number $k$ of improvement iterations for the discriminator,
        stopping criterion $T_{\text{condition}}$
Output: a generator $G$ that is able to fool the trained discriminator $D$

while not $T_{\text{condition}}(f,\theta)$ do
    for $k$ steps do
        Sample a minibatch of noise samples $\{z^{(1)},\dots,z^{(m)}\}$
        Sample a minibatch of true samples $\{x^{(1)},\dots,x^{(m)}\}$
        Perform stochastic gradient ascend for the discriminator:
            $$\nabla_{\theta^{(d)}} \frac{1}{m} \sum_{i=1}^{m} \Big( \log\big(d(x^{(i)};\theta^{(d)})\big) + \log\big(1 - d\big(g(z^{(i)};\theta^{(g)}),\theta^{(d)}\big)\big) \Big)$$
    Sample a minibatch of noise samples $\{z^{(1)},\dots,z^{(m)}\}$
    Perform stochastic gradient descending for the generator:
        $$\nabla_{\theta^{(g)}} \frac{1}{m} \sum_{i=1}^{m} \Big( \log\big(1 - d\big(g(z^{(i)};\theta^{(g)}),\theta^{(d)}\big)\big) \Big)$$
```

# Bibliography

[1] Boris Weisfeiler and Andrei Leman. “The reduction of a graph to canonical form and the algebra which appears therein”. In: NTI, Series 2.9 (1968), pp. 12–16.

[2] Paul W Holland and Samuel Leinhardt. “Transitivity in structural models of small groups”. In: Comparative group studies 2.2 (1971), pp. 107–124.

[3] David W Matula and Leland L Beck. “Smallest-last ordering and clustering and graph coloring algorithms”. In: Journal of the ACM (JACM) 30.3 (1983), pp. 417–427.

[4] K Hornik, M Stinchombe, and H White. “Multilayer feed-forward network and universal approximator. Neural Network”. In: vol 2 (1989), pp. 7359–366.

[5] 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.

[6] Duncan J Watts and Steven H Strogatz. “Collective dynamics of â€˜small-worldâ€™networks”. In: nature 393.6684 (1998), pp. 440–442.

[7] Yun Chi et al. “Frequent subtree mining–an overview”. In: Fundamenta Informaticae 66.1-2 (2005), pp. 161–198.

[8] Franco Scarselli et al. “The graph neural network model”. In: IEEE transactions on neural networks 20.1 (2008), pp. 61–80.

[9] Nino Shervashidze et al. “Efficient graphlet kernels for large graph comparison”. In: Artificial intelligence and statistics. PMLR. 2009, pp. 488–495.

[10] László Babai. "Graph isomorphism in quasipolynomial time". In: Proceedings of the forty-eighth a
2016, pp. 684–697.

[11] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: arXiv preprint arXiv:1609.02907 (2016).

[12] Keyulu Xu et al. "How powerful are graph neural networks?" In: arXiv preprint arXiv:1810.00826 (2018).

[13] László Babai. "Canonical form for graphs in quasipolynomial time: preliminary report". In: Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Co
2019, pp. 1237–1246.

[14] Federico Errica et al. "A fair comparison of graph neural networks for graph classification". In: arXiv preprint arXiv:1912.09893 (2019).

[15] Till Schulz and Pascal Welke. "On the necessity of graph kernel baselines". In: ECML-PKDD, GEM workshop. Vol. 1. 2. 2019, p. 6.

[16] Karsten Borgwardt et al. "Graph kernels: State-of-the-art and future challenges". In: arXiv preprint arXiv:2011.03854 (2020).

[17] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. "A survey on graph kernels". In: Applied Network Science 5.1 (2020), pp. 1–42.

[18] Janis Kalofolias, Pascal Welke, and Jilles Vreeken. "SUSAN: The Structural Similarity Random Walk Kernel". In: Proceedings of the 2021 SIAM International Conferenc
SIAM. 2021, pp. 298–306.