Research notes: Graph kernels, Weisfeiler Lehman, Wasserstein Distance, . . .

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

April 27, 2022

Contents

Chapter 1

Best practices for scientific research and presentation

1.1 Best practice for doing research with a scientific paper

When reading a new paper, complete these three stages:

1. Read the paper on a **first pass** - to get an overview of **what happens**.

2. Read the paper on a **second pass** - to **understand** the material. Document

   - the main results, claims, notions and ideas
   - the structure (can it be improved when reciting the paper?)
   - the motivation of definitions
   - the used methods or ideas (e.g. for proofs)
   - where the material is difficult to understand
   - questions.

3. Prepare a presentation which restructures and compresses the material. Highlight interesting points, surprising results and also consider not fully understood passages.

1.2 Best practice for scientific talks

Basic structure of a scientific talk:

1. Introduction & Motivation
   (**WHAT** are we doing? **WHY** are we doing it?)
2. Basic definitions (if necessary)
3. Result ("′ High Level message"′)
4. Related work
5. Methodology, overview of intermediate results, examples and the investigated settings
   (**HOW** is it being done?)
6. Conclusion
   (**WHAT** has been achieved?)
7. Future Work & link to more information
   (**WHAT** else is there to do or take into consideration?)
8. Q & A

Tips and tricks for the presentation:

- Slides should be used parallel to the verbal communication. They should support the speaker in the background, or offer summarizing pictures which are explained.
- Avoid a lot of text, especially if it differs from the actually spoken words.
- Interact with the slides, do not let them run completely separated in the background.
- Use roughly **one slide per minute** and do not use more than **two main points per slide**.
- Prefer pictures over formulas.

Chapter 2

Mathematical basics

Graph theory is the mathematical theory of the properties and applications of graphs (networks).

Chapter 3

Definitions

This chapter contains definitions. If it exists, the means of computing a defined object are appended after the definition of the object. Proofs and theorems on the efficiencies of these means (algorithms) can be found in chapter 5.

3.1 Graphs

Definition 1.1: Graph

A **graph** is a tuple $G = (V, E)$ where V is a finite set of vertices and $E \subseteq 2^V$ is a set of undirected edges.
Use variables $n_G := |V|$ (the **size** of G) and $m_G := |E|$.

G is called

- **labeled**, if there exists a labeling function $\ell : V \to \Sigma$ for a finite **label alphabet** Σ. In this case write $G = (V, E, \ell)$.
- **attributed**, if for each vertex $v \in V$ there exists an associated vector $a(v) \in \mathbb{R}^m$.
  We call $a(v)$ **vertex attributes** (high-dimensional continuous vectors) and $l(v)$ **vertex labels** (integer).
- **weighted** if there exists a weight function on its edges $w : E \to \mathbb{R}$.
- **subtree** of a graph $O = (W, F)$ if G is a graph and $V \subseteq W$ and

$E \subseteq F$.

- **complete**, if it contains 2^V edges. That is, between every two vertices there is an edge. A complete graph with n vertices is denoted as the graph K_n.

Definition 1.2: Adjacency matrix

The edges and edge labels of a graph $G = (V, E)$ can be represented by an adjacency matrix $M \in \Sigma^{n_G \times n_G}$, where Σ is the set of edge labels. An entry $m_{i,j}$ in M denotes if there exists an edge ($(i,j) \in E$ or $\{i,j\} \in E$). If no such edge exists, set $m_{i,j} = 0$. If G has edge labels given by ℓ_E set $m_{i,j} = \ell_E(i,j)$. If $0 \in \Sigma$, the coding can be chosen differently.

The advantages of the adjacency matrix representation are:

- Space efficient for representing dense graphs.
- Edge (weight) lookup time in $\mathcal{O}(1)$.

The disadvantages of the adjacency matrix representation are:

- Space complexity of $\mathcal{O}(V^2)$.
- Iterating over all edges has time complexity $\mathcal{O}(V^2)$.

Definition 1.3: Adjacency list

The edges and edge labels of a graph $G = (V, E)$ can be represented by an adjacency list L. L contains for every vertex a list of tuples with neighboring vertices and the edge weights between them.

The advantages of the adjacency list representation are:

- Space efficient for representing sparse graphs.
- Allows for an efficient iteration over all edges.

The disadvantages of the adjacency list representation are:

- Less space efficient for denser graphs.
- Edge (weight) lookup time complexity is $\mathcal{O}(E)$.

Definition 1.4: Edge list

The edges and edge labels of a graph $G = (V, E)$ can be represented by an edge list L. L contains triplets (u, v, c) for every edge from vertex u to vertex v with edge weight c.

The advantages of the edge list representation are:

- Space efficient for representing sparse graphs.
- Allows for an efficient iteration over all edges.
- Arguably the simplest graph representation.

The disadvantages of the edge list representation are:

- Less space efficient for denser graphs.
- Edge (weight) lookup time complexity is $\mathcal{O}(E)$.

Definition 1.5: Bridge

Let $G = (V, E)$ be connected a graph with k connected components. An edge $e \in E$ is called a **bridge** if $(V, E \backslash e)$ has $k + 1$ connected components.

Definition 1.6: Articulation point

Let $G = (V, E)$ be connected a graph with k connected components. A vertex $v \in V$ is called an **articulation point** (**cut vertex**) if $(V \backslash v, E)$ has more than k connected components.

Definition 1.7: Neighborhoods in graphs

Let $G = (V, E)$ be a graph. The set

$$\mathcal{N}^1(v) = \mathcal{N}(v) := \{u \in V \mid (u, v) \in E\}$$

is called (**first-**) **neighborhood** of G.
The **degree** of v in G is given as $|\mathcal{N}(v)| = \deg(v)$. Set $d_G = \max_{v \in V} \deg(v)$ (the maximum degree in G).

For $0 < i \in \mathbb{N}$ define recursively higher neighborhoods of v:

$$\mathcal{N}^i(v) = \{u \in \mathcal{N}^{i-1}(v) | \ (u, v) \in E\}$$

(Instead of defining the vertices via existing adjacent vertices, one can also define it as the set of vertices that are reachable with paths of length of at most i (no multi sets).)

Definition 1.8: Walks, cycles and paths

A **walk** W in a graph $G = (V, E)$ is a sequence of vertices in a graph, in which consecutive vertices are connected by an edge:

$$W = (v_0, \ldots, v_k) \in V^k \quad \text{s.t.} \quad \forall i, j = 0, \ldots, k-1 : \ \{v_i, v_{i+1}\} \in E$$

A **cycle** is a **walk** $W = (v_0, \ldots, v_k)$ with identical start and end vertex, e.a. $v_0 = v_k$.
A **path** P in a graph $G = (V, E)$ is a walk $W = (v_0, \ldots, v_k)$ in G, where all vertices of the walk are distinct:

$$\forall i, j : \quad v_i \neq v_j$$

Definition 1.9: Graphs and Laplacian matrix

Graph $G = (V, E)$, $|V| = N$, connected, undirected, edge weighted.
Adjacency matrix $W \in \mathbb{R}^{N \times N}$.
$d(i)$ degree of $i \in V$. Degree matrix $D \in \mathbb{R}^{N \times N}$:

$$D_{i,j} = \begin{cases} d(i), & i = j \\ 0, & i \neq j \end{cases}$$

Laplacian matrix $L := D - W$.

Definition 1.10: Graph isomorphism

Two graphs G_1 and G_2 are **isomorphic** $G_1 \equiv G_2$, if there exists a bijective function between their vertices, that preserves all edges and labels.

Definition 1.11: Structure and depth preserving mapping

A**structure and depth preserving mapping** (SDM) between two rooted trees T and T' is a triple (M, T, T') with $M \subseteq V(T) \times V(T')$ satisfying

1. $\forall (v_1, v_1'), (v_2, v_2') \in M : \quad v_1 = v_2 \iff v_1' = v_2$ (definiteness)
2. $(r(T), r(T')) \in M$ (root preserving)
3. $\forall (v, v') \in M : \ (\text{par}(v), \text{par}(v')) \in M$ (structure preserving)

The set of all structure and depth preserving mappings between T and T' is denoted by $\text{SDM}(T, T')$.**[2021_Schulz_CONF]**

Note that SDMs preserve siblings and vertices can only be mapped onto vertices of the same depth. Also connected subtrees are mapped only onto connected subtrees.**[2021_Schulz_CONF]**
For an SDM (M, T, T') let $T = T_0, T_1, \ldots, T_k$ be a sequence of trees such that T_{i+1} is obtained from T_i by applying one of the following atomic transformations

- **Relabel**: If $(v, v') \in M$, then replace the label ov v in T_i by that of v'.
- **Delete**: If v is a leaf in T_i and it does not occur in a pair of M, then remove v from T_i.
- **Insert**: If v' is a vertex in T' which does not occur in a pair of M and for which the corresponding parent u already exists in T_i, then add a child to u with the label of v'.

Definition 1.12: Costs of structure and depth preserving mappings

Let T and T' be unfolding trees over the vertex label alphabet Σ and let $\gamma : \Sigma^\perp \times \Sigma^\perp \to \mathbb{R}$ a cost function (where $\perp$ is the blank symbol). Let (M, T, T') be a structure and depth preserving mapping (SDM) and N

and N' the vertices in T and T' that do not occur in any pair of M.
Then the **cost** $\gamma(M)$ of the SDM is given as

$$\gamma(M) := \underbrace{\sum_{(v,v')\in M} \gamma\big(\ell(v), \ell(v')\big)}_{\text{relabeling}} + \underbrace{\sum_{v\in N} \gamma\big(\ell(v), \bot\big)}_{\text{deletion}} + \underbrace{\sum_{v'\in N'} \gamma\big(\bot, \ell(v')\big)}_{\text{insertion}}$$

[2021_Schulz_CONF]

Note that the SDTED is simply the minimum cost of all SDM.
Note that the SDTED of unfolding trees corresponds to the Wasserstein distance
of the real-valued vector representation of the unfolding trees (see)**[2021_Schulz_CONF]**.

Algorithm 1 Naive computation of an SDTED

Input: two labeled rooted trees T, T' (w.l.o.g. $\ell = \ell' : V(T) \cup V(T') \to \Sigma$),
a cost function $\gamma : \Sigma^{\perp} \times \Sigma^{\perp} \to \mathbb{R}$.
Output: $\textsc{SdTed}(T, T')$.

Recall that the function $r(T)$ gives the root of a tree T and $F(v)$ gives the set of subtrees rooted at the children of vertex v.

```
1: procedure SDTED($T, T'$)
2:     $F := F(r(T)),\ F' := F(r(T'))$                                ▷ Trees below the root
3:     Pad $F$ and $F'$ with empty trees $T_\perp$ such that $|F| = |F'| = \deg(r(T)) + \deg(r(T'))$
4:     for all $T_i \in F,\ T'_j \in F'$ do
5:         $\delta_{i,j} = \begin{cases} \textsc{SdTed}(T_i, T'_j) & \text{if } T_i \not\equiv T_\perp \ \wedge\ T'_j \not\equiv T_\perp \\ \sum_{v \in V(T_i)} \gamma(\ell(v), \perp) & \text{if } T_i \not\equiv T_\perp \ \wedge\ T'_j \equiv T_\perp \\ \sum_{v' \in V(T'_j)} \gamma(\ell(v'), \perp) & \text{if } T_i \equiv T_\perp \ \wedge\ T'_j \not\equiv T_\perp \\ 0 & \text{otherwise} \end{cases}$    ▷ Define a distance matrix
6:     Let $S \subseteq F \times F'$ be a minimum cost perfect bipartite matching with respect to the distances $\delta$ return $\gamma(\ell(r(T)), \ell(r(T'))\perp) + \sum (T_i, T_j) \in S\delta_{i,j}$
```

Time complexity: Naively a exponential number of recursion calls is needed. But note that the number of i-unfolding trees in T and T' is bounded by $n = |V(T)|$ and $n' = |V(T')|$. Thus, for each recursion depth (depth of the unfolding trees), the algorithm needs to be invoked at most nn' times.
Also, once $\textsc{SdTed}(T_i, T_j)$ has been calculated, it can be **stored in a look-up table** for later calls from a higher recursion.
Thus at most $nn'h$ invocations of a minimum cost perfect bipartite matching algorithm (each of complexity $\tilde{\mathcal{O}}((2d)^3)$, d is the maximum degree of T and T') where h is the depth are necessary.
This gives a total runtime approximation of

$$\tilde{\mathcal{O}}(nn'h(2d)^3)$$

Remarks:

- Line 3 ensures, that both sets of rooted subtrees can contain the subtree set of the deeper recursion.
- The second case in the definition in line 5 considers the case that subtree T_i is not part of the mapping ($T_i \not\equiv T_\perp$ but $T'_j \equiv T_\perp$) - thus for a optimal SDM, all vertices in subtree T_i are **deleted**.
- The third case in the definition in line 5 considers the case that subtree T_i is not part of the T but the mapped tree T_j is part of T' (insertion) - thus

Definition 1.13: Weisfeiler-Lehman graph at height i

The **Weisfeiler-Lehman graph at height** i (**WL-graph at height** i) of the graph $G = (V, E, \ell) = (V, E, l_0)$ is the graph

$$G_i = (V, E, l_i)$$

(That is the original graph, with the labeling function that has been constructed after i iterations of algorithm 8.)

Definition 1.14: Weisfeiler-Lehman sequence up to height i

The **Weisfeiler-Lehman sequence up to height** i (**WL-sequence up to height** i) of the graph $G = (V, E, \ell) = (V, E, l_0) = G_0$ is the sequence

$$\{G_0, G_1, \ldots, G_i\} = \{(V, E, l_0), (V, E, l_1), \ldots, (V, E, l_i)\}$$

With the notation from above we can write: $G_i = r(G_{i-1})$.

Definition 1.15: Regular graphs

A graph G is called k**-regular**, if the number of common neighbors of a k element subset of vertices only depends on the isomorphism type of the subgraph induced by the k vertices.//
1-regular graphs are also called **regular**. 2-regular graphs are also called **strongly regular [1992_Cai_IEEE]**.

Definition 1.16: Strongly connected component

A **strongly connected component** (**SCC**) of a graph $G = (V, E)$ is a subset $C \subset V$ of vertices, such that between every two vertices in C there exists a path in $G[C]$.
One can think of SCCs as self-contained cycles.

Definition 1.17: Graph separator

A **separator** of a graph $G = (V, E)$ is a subset $S \subset V$ such that the induced subgraph on $V - S$ has no connected component with more than $|V|/2$ vertices.

Definition 1.18: Topological vertex ordering

A **topological ordering** (**topological vertex ordering**) of a digraph is an ordering of all vertices such that for each directed edge (v, w), v appears before w in the ordering.
Vividly explained, if the graph is arranged such that the vertices are sorted with respect to the topological ordering in a line from left to right, all edges will point to the right.

Trivially, graphs which contain a cycle do not have a topological ordering. Digraphs which posses a topological vertex ordering are called **directed acyclic graphs** (see definition 1.29).

Definition 1.19: First-order graph theory language

..

3.1.1 Trees

Definition 1.20: Tree

A **tree** T is a graph without circles.
A tree is called **rooted**, if one vertex $r(T) \in V$ is called root. For any $v \in V \backslash \{r(T)\}$ the **parent of** v is the unique neighbor of v on the path to $r(T)$ (par(v)). Accordingly the **children** of v are all vertices that have v as parent.**[2021_Schulz_CONF]**
The **subtree rooted in** v, denoted $T[v]$, is the subgraph T that is rooted at v and induced by all descendants of v.
$F(v)$ denotes the **set of subtrees** rooted at the children of v. **[2021_Schulz_CONF]**
If T is directed and every edge point away from the root the graph is

called an **arborescence** (**out-tree**). If all edges point towards the root, it is called an **anti-arborescence** (**in-tree**).

Definition 1.21: Subtree pattern

A **subtree** of a graph G is a connected subset of distinct vertices in G with an underlying tree structure. A **subtree-pattern** (**tree-walk**) extends the notation of subtrees, by allowing vertices to be equal.
Equal vertices in the graph G are treated as distinct vertices in the subtree-pattern. Thus it is still a cycle-free graph (a tree).
Similarly how a path extends the notation of a walk in a graph.

frametitle=LEMMA 1.1 - :, innertopmargin=10pt, linecolor=thmLnColor!40, linewidth=2pt, topline=true, style=MDFStyBlueScreen

LEMMA 1.1:
A connected graph G is a tree, iff it has $n_G - 1$ edges.

3.1.1.0.1 Proof: Let G be a tree. Since G by definition, G has no circles, there must exists a vertex l with exactly one neighbor, which we call a leaf. Removing this leaf (and the incident edge) does not close any circles, thus $G\backslash\{l\}$ is still a tree with $e_G - 1$ edges. Repeat this step until no more leaves exist. Since there are no circles, no two vertices can remain. And since leaves have exactly one neighbor, exactly one vertex must remain. Thus $n_G - 1$ leaves were iteratively removed and thus $n_G - 1$ edges must have been present in G.
Let G have $n_G - 1$ edges. We will give a proof by contradiction and assume that G is not a tree, meaning that there exists a circle C of $k > 2$ vertices and k edges. If there are no leaves in the graph, every vertex has at least two incident edges and thus there are at least n_G edges - which is a contradiction. This implies that G has leaves. Again, we can iteratively remove leaves (and their adjacent). The resulting reduced graphs G' will contain exactly one edge and one vertex less, than before, which implies that $e_{G'} = n_{G'} - 1$. If no more leaves can be removed, at least the circle C remains, since it contains vertices with at least two incident edges. If there are more circles, there exist vertices with more than two incident edges. Thus every remaining vertex has at least two incident edges and thus $e_{G'} \geq n_{G'}$ which is the expected contradiction. □

Definition 1.22: (Rooted) trees

A **tree** T is a graph without cycles.
A **rooted tree** is a tree T with a designated root vertex r.

The **height** h_T of a rooted tree is the maximum distance between the root and any other vertex in the rooted tree:

$$h_T = \max_{v \in V}\{k \mid (r, v_1, \dots, v_k) \text{ is a path}\}$$

One can extend the notion of subgraphs to **subtrees**. One can extend the notion of subtrees to **subtree patterns** ([**2008_Bach_ICML**]), by duplicating vertices and treating them as distinct.

Note that all subtree kernels compare subtree patterns, not subtrees.

Definition 1.23: Minimum spanning tree

Let $G = (V, E)$ be a connected graph. A **minimum spanning tree** (**MST**) is a tree $T = (V, S)$ contains all vertices of G and a subset of the edges E such that the sum of all edges is minimal. (If G is unweighted, consider every edge to have weight 1.)
Sometimes, only a MST is defined by only the edge set.

Definition 1.24: Tree edit distance

Let $\perp \notin \Sigma$ be a special **blank** symbol. For $\Sigma^\perp = \Sigma \cup \{\perp\}$ we define a cost function $\gamma : \Sigma^\perp \times \Sigma^\perp \to \mathbb{R}$ and require γ to be a metric.
An **edit script** or **edit sequence** from a labeled tree T into a labeled tree T' is a sequence of edit operations turning T into T'. An **edit operation** can

- *relabel* a single vertex v,
- *delete* a single vertex v (and connect all its children to the parent of v) or
- *insert* a single vertex w between v and a subset of the children of v.

The costs of such edits is defined by γ.

- Relabeling v from a to b costs $\gamma(a, b)$ and
- adding or deleting v costs $\gamma(\ell(v), \perp)$.

An edit script between T and T' of minimal cost is called **optimal** and its costs is called **tree edit distance**. **[2021_Schulz_CONF]**

Definition 1.25: Structure and depth preserving tree edit distance

Let T and T' be unfolding trees over the vertex label alphabet Σ and let $\gamma : \Sigma^{\perp} \times \Sigma^{\perp} \to \mathbb{R}$ a cost function (where $\perp$ is the blank symbol).
The **structure and depth preserving tree edit distance** from T to T' (SDTED(T, T')) is defined as the minimal cost of a SDM between T and T':

$$\textsc{SdTed}(T, T') = \min\{\gamma(M) \mid (M, T, T') \in \textsc{SdM}(T, T')\}$$

[2021_Schulz_CONF]

Definition 1.26: Unfolding tree

An **unfolding** tree is a rooted tree.
An i-unfolding tree T has a **real-valued vector representation** $\mathbb{V}(T)$ with

$$\mathbb{V}(T) = \big(\mathbb{V}_r(T), \mathbb{V}_c(T)\big)$$

where

- $\mathbb{V}_r(T)$ denotes the roots label $\ell\big(r(T)\big)$ and
- $\mathbb{V}_c(T)$ denotes the set of $(i-1)$-unfolding child trees $F\big(r(T)\big)$.

A simple realization is $\mathbb{V}_r(T) \in [0, 1]^{\Sigma}$ with entry 1 corresponding to the roots label and 0 everywhere else.
$\mathbb{V}_c(T)$ is made up of counts of isomorphic child trees and contains an entry for empty child trees.

Note that by construction, vertices close to a root v appear in unfolding trees of this root at smaller depth. Furthermore, the number of occurrences of any vertex in $T^i(G, v)$ grows exponentially with i once it has appeared for the first time.

Algorithm 2 Weisfeiler-Lehman vertex relabeling method

The method was originally designed to decide isomorphism between graphs [**2021_Schulz_CONF**, **1968_Weisfeiler_CONF**]. The key idea is to iteratively encode the label of each vertex and its neighbors into a new label.

Input: a labeled graphs $G = (V, E, \ell_0)$,
a number of iterations $h \in \mathbb{N}$,
a list of alphabets Σ_i (for $i = 1, \ldots, h$),
a perfect (injective) hash function $\mathcal{H} : \Sigma_i \times \Sigma_i^* \to \Sigma_{i+1}$ and
the set of all possible labels Σ_0 (finite and with a total order $>$).
Output: a re-labeled graph $G = (V, E, \ell_h)$.

```
1: for $i = 1, \ldots, h$ do
2:     for $v \in V$ do
3:         $N = \mathrm{sort}\big([\ell_i(u) |\ u \in \mathcal{N}(v)]\big)$
4:         $\ell_{i+1}(v) := \mathcal{H}(\ell_i, N)\ \in \Sigma_{i+1}$     ▷ The list of neighbor labels is sorted
```

Note: Two graphs G and G' are not isomorphic if the corresponding multisets $\{\!\{\ell_i(v) |\ v \in V(G)\}\!\}$ and $\{\!\{\ell_i(v') |\ v' \in V(G')\}\!\}$ are different for some i. Otherwise they **may or may not** be isomorphic.[**2021_Schulz_CONF**]
There are non-isomorphic graphs, which the test cannot distinguish. For example two regular graphs with the same number of vertices and vertex degrees, where one is connected and the other one is not.

Babai, Erdös and Selkow [**1980_Babai_SIAM**]: The 1-dim WL vertex labeling method computes normal forms for all but an $n^{-1/7}$ fraction of the n-vertex graphs.
By handling a few exceptions separately, this fraction can be improved to $c^{-n \log n / \log \log n}$ [**1979_Babai_CONF**]- which classifies the 1-dim WL vertex labeling method as an **average linear time canonical labeling algorithm**.

Definition 1.27: Unfolding tree

Consider algorithm 2. In each iteration i implicitly tree patterns of depth i (which are being compressed into labels) are constructed.
Each such tree, denoted by $T^i(G, v)$ is called the **depth-i unfolding tree**

(*i*-**unfolding tree**) of G at v.

Definition 1.28: Graph Neural Networks

Graph Neural Networks (**GNNs**) use the graph structure and vertex features to learn a representation vector of a vertex (h_v) or an entire graph (h_G). Modern GNNs follow a neighborhood aggregation strategy, where the representation of a vertex is update iteratively by aggregating representations of its neighborhood. After k iterations of aggregation, the vertex representation captures the structural information of its k-hop network neighborhood.

Let $h_v^{(k)}$ be the feature vector of vertex v at the k-th iteration (or layer). Initialize $h_v^{(0)}$ with the given features of the graph and let $\mathcal{N}_v$ be the neighborhood of vertex v. Using this, the k-th layer of a GNN can be described as

$$a_v^{(k)} = \text{AGGREGATE}\left(\left\{h_u^{(k-1)} \middle| u \in \mathcal{N}_u\right\}\right), \qquad h_v^{(k)} = \text{COMBINE}\left(h_u^{(k-1)}, a_u^{(k)}\right)$$

[2019_Xu_CONF]
Message-passing graph convolutional networks (GCNs) update the vertex representations from one layer to the other according to the formula

$$h_i^{l+1} = f(h_i^l, \{h_j^l\}_{j \in \mathcal{N}_i})$$

where h^l, $h^{l=1} \in \mathbb{R}^{n \times d}$ ($n = |V(G)|$ and d is the dimension of the vertex features). Since these updates are local and independent of the graph size, the space and time complexity is in $\mathcal{O}(E)$. For sparse graphs, this can be reduced to linear time $\mathcal{O}$ **[2020_Dwivedi_CONF]**. GCNs are called **isotrophic**, when the vertex update f treats every edge direction equally. Expressed with weight matrices $W_{1,2}^l \in \mathbb{R}^{d \times d}$ the update function has the form

$$h_i^{l+1} = \sigma\Big(W_1^l h_i^l = \sum_{j \in \mathcal{N}} W_2^l h_j^l\Big)$$

where σ is a non-linear point-wise activation like ReLU.**[2020_Dwivedi_CONF]**
GCNs are called **anisotrophic**, when every edge direction is treated dif-

ferently:

$$h_i^{l+1} = \sigma\Big(W_1^l h_i^l = \sum_{j \in \mathcal{N}} \eta_{i,j} W_2^l h_j^l\Big)$$

where $\eta_{i,j} = f^l(h_i^l, h_j^l)$ and f^l is a parameterized function whose weights are learned during training [**2020_Dwivedi_CONF**].
Weisfeiler-Lehman GNNs on the other hand are based on the WL test [88]. They can be distinguished in k-WL GNNs, if they can distinguish two non-isomorphic graphs w.r.t. the k-WL test. The layer update equation for 3-WL GNNs is defined as

$$h^{l=1} = \text{Concat}\left(M_{W_1^l}(h^l),\ M_{W_2^l}(h^l),\ M_{W_3^l}(h^l)\right)$$

with $h^k, h^{l+1} \in \mathbb{R}^{n \times n \times d}$ and $W_{1,2,3}^l \in \mathbb{R}^{d \times d \times 2}$ are M_W are 2-layer MLPs applied to the feature dimension.
RingGNNs [**2019_Chen_CONF**] achieve higher learning capacity than 2-WL GNNs. Their update equation is

$$h^{l=1} = \sigma\Big(w_1^l L_{W_1^l}(h^l) + w_2^l L_{W_2^l}(h^l) + w_3^l L_{W_3^l}(h^l)\Big)$$

with $h^k, h^{l+1} \in \mathbb{R}^{n \times n \times d}$ and $W_{1,2,3}^l \in \mathbb{R}^{d \times d \times 17}$ are L_W are linear layers and $w_{1,2}^l \in \mathbb{R}$. RingGNNs have the same space and time complexities as 3-WL GNNs.

Many GNN implementations with different neighborhood aggregation and graph-level pooling schemes have been proposed. Examples are - see page 1 in [**2019_Xu_CONF**]
Property: message-passing
Examples application domains for of GNNs:

- chemistry [**2015_Duvenaud_NIPS, 2017_Gilmer_CONF**]
- physics [**2019_Cranmer_NIPS, 2020_SanchezGonzalez_PMLR**]
- social sciences [**2016_Kipf_ICLR, 2019_Monti_CONF**] (graph of research paper citations)
- knowledge graphs [**2018_Schlichtkrull_LNCS, 2020_Chami_CONF**]
- recommendation [**2020_Chami_CONF, 2018_Ying_KDD**]
- neuroscience [**2017_Griffa**]

Historically, three classes of GNNs have been developed:

- First models: [**2009_Scarselli_IEEE, 2013_Bruna_CONF, 2016_Kipf_ICLR, 2016_Defferrard_NIPS, 2016_Sukhbaatar_NIPS, 2017_Hamilton_NIPS**] aimed at extending the original convolutional neural networks [**1995_LeCun_CONF, 1998_Lecun_IEEE**] to graphs. (**Message-passing GCNs**)
  Popular isotrophic GCNs are vanilla GCNs-Graph Convolutional Networks [**2016_Kipf_ICLR, 2016_Sukhbaatar_NIPS**] and GraphSage [**2017_Hamilton_NIPS**].

- Enhance original models with anisotropic operations on graphs [**1990_Perona_IEEE**] such as [**2016_Battaglia_NIPS, 2017_Marcheggiani_CONF, 2020_Mirhoseini_CONF, 2017_Velickovic_ICLR, 2017_Bresson_CONF**].

- GNNs that improve upon theoretical limitations of previous models [**2018_Xu_CONF, 2018_Morris_AAAI, 2019_Maron_NIPS, 2019_Chen_CONF, 2019_Murphy_ICML, 2019_Srinivasan**] (**WL-GNNs**)
  Examples are the GIN - Graph Isomorphism Network [**2018_Xu_CONF**].

The first two models can only differentiate simple non-isomorphic graphs and cannot separate automorphic vertices [**2020_Dwivedi_CONF**].
Message-passing GCNs profit from deep learning building blocks such as batching, residual connections and normalization. GCNs can highlz profit from sparse matrix computations [**2020_Dwivedi_CONF**]. Furthermore, "anisotropic GCNs which leverage attention [80] and gating [**2017_Bresson_CONF**] mechanisms perform consistently across graph, vertex and edge/level tasks, improving over isotropic GCNs on five out of seven datasets." Additionally, for link prediction tasks, learning features for edges as joint representations of incident vertices during message passing significantly boosts performance [**2020_Dwivedi_CONF**]. The expressivitz of GCNs can be increased using graph positional encodings with Laplacian eigenvectors [**2020_Dwivedi_CONF, 2003_Belkin_IEEE**].
Theoretically designed WL-GNNs are prohibitive in terms of space and time complexity and not amenable to batched training. Thus they are usually outperformed by GCNs [**2020_Dwivedi_CONF**].

3.1.2 Directed acyclic graphs

Definition 1.29: Directed acyclic graphs

A **directed acyclic graph** G is a directed graph with no cycles.

Notice, that all out-trees are DAGs.
Also notice, that on DAGs the **single source shortest path** (**SSSP**) problem can be solved efficiently in $\mathcal{O}(n+m)$ runtime complexity.

3.1.3 Bipartite graphs

Definition 1.30: Bipartite graphs

A **bipartite graph** $G = (U \cup V, E)$ is a graph whose vertices can be split into two groups, such that there are no edges between the vertices in U and no edges between the vertices in V. That is, all edges connect a vertex in U with a vertex in V. In the undirected case this means:

$$\forall e = \{u, v\} \in E : \qquad u \in U \wedge v \in V$$

In the directed case this means:

$$\forall e = (x, y) \in E : \qquad (x \in U \wedge y \in V) \ \vee \ (y \in U \wedge x \in V)$$

LEMMA 1.2 - Two-colourability of bipartite graphs:

graph G is bipartite, iff it is two colourable.

3.1.3.0.1 Proof: Assign all vertices in U to one color and all in V to the other color (or the other way around). □

LEMMA 1.3 - Cycle lengths in bipartite graphs:

graph G is bipartite, iff there is no cycle of odd length.

3.1.3.0.2 Proof: □

3.1.4 Examples of graph databases

The authors in **[2020_Dwivedi_CONF]** suggest, that small datasets like Cora, Citeseer and TU datasets are not able to statistically separate the performance of GNNs, as all GNNs perform almost statistically the same.
Open Graph Benchmark (OGB) **[2020_Hu_CONF]**

Domain	Construction	Dataset	Vertices	Total vertices	
Chemistry	Real-world	ZINC	9-37	277,864	Gra
Mathematical Modeling	Artificial graphs (SBM)	PATTERN	44-188	1 664 491	Vert
		CLUSTER	41-190	1 406 436	
Computer Vision	Semi-artificial	MNIST	40-75	4 939 668	Grap
		CIFAR10	85-150	7 058 005	
Combinatorial Optimization	Artificial	TSP	50-500	3 309 140	Edg
Social Networks	Real-world	COLLAB	235 868	235 868	Edg
Circular Skip Links	Artificial	CSL	41	6150	Grap

3.1.4.1 MUTAG

The dataset contains 188 mutagenic aromatic and heteroaromatic nitro compounds, and the task is to predict whether or not each chemical compound has mutagenic effect on the Gram-negative bacterium Salmonella typhimurium. (From `https://ysig.github.io/GraKeL/0.1a8/documentation/introduction.html`)

3.1.4.2 ZINC

- Real-world
- Molecular dataset
- 250K graphs
- Example usage: Graph Regression

3.1.4.3 PATTERN

- Generated with Stochastic Block Models (SBM)
- Vertex classification; identification of subgraphs

3.1.4.4 CLUSTER

- Generated with Stochastic Block Models (SBM)
- Vertex classification

3.1.4.5 MNIST

-
- Image classification

Sanity check dataset. Expect 100% classification.

3.1.4.6 CIFAR10

-
- Image classification

3.1.4.7 TSP

-
- Link prediction

3.1.4.8 COLLAB

-
- Link prediction (proposed bz OGB [**2020_Hu_CONF**])

3.1.4.9 CSL

- Synthetic dataset
-

3.2 Laplacian eigenvectors

Used as positional embeddings (PE) - see [**2020_Dwivedi_CONF**] for reference (page 7).

3.3 Kernels

Definition 3.1: $\mathcal{R}$-Convolution

Decompose graph G into substructures and define a kernel value $k(G, G')$ as a combination of substructure similarities.

Often $\mathcal{R}$-Convolution kernels discard valuable information such as the distribution of the substructures.

Several different graph kernels have been defined in machine learning. They can be categorized into three classes: Graph kernels based on

- walks (Kashima et al., 2003; Gärtner et al., 2003) and paths ([**2005_Borgwardt_CONF**]),
- limited-size subgraphs (**graphlets**, [**2004_Horvath_KDD**, **2009_Shervashidze_PMLR**]) and
- subtree patterns ([**2003_Ramon_CONF**, **2008_Mahe_CONF**]).

1. Compute the number of matching pairs of random walks (resp. paths). $\mathcal{O}(n^6)$ [**2003_Gaertner_CONF**]. In terms of Kronecker products possible in $\mathcal{O}(n^3)$ (Vishwanathan et al., 2010)
Speed up with random walks of fixed size [**2007_Harchaoui_IEEE**]. By [**2005_Borgwardt_IEEE**] in $\mathcal{O}(n^4)$.

2. Graphlets represent graphs as counts of all types of subgraphs of size $k \in \{3, 4, 5\}$. Based on sampling or exploitation of the low maximum degree of graphs ([**2009_Shervashidze_NIPS**]) - unlabeled graphs only.
Cyclic pattern kernels count pairs of matching cyclic patterns ([**2004_Horvath_KDD**]).
NP-hard for general graphs.
Count identical pairs of rooted subgraphs containing vertices up to a certain distance from the root, with roots which are located at a certain distance from each other (De Grave (2019)

3. Refined Ramon-Gärtner kernel [**2008_Mahe_CONF**] (applications in chemoinformatics and hand-written digit recognition). α-ary subtrees with at most α children per vertex ([**2008_Mahe_CONF**], [**2008_Bach_ICML**]) - feasible on

small graphs with many distinct vertex labels.

All these graph kernels scale at least $\mathcal{O}(n^3)$ on large labeled graphs with more than 100 vertices.

Definition 3.2: Kernel

Kernels are a class of similarity functions.
Let X be a set and $k : X \to X \to \mathbb{R}$ be a function associated with a Hilbert space $\mathcal{H}$ such that

$$\exists \phi : X \to \mathcal{H} \text{ s.t.} \qquad k(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}}$$

Then $\mathcal{H}$ is a **reproducing kernel Hilbert space** (**RKHS**) and k is said to be a **positive definite kernel**.

A positive definite kernel can be interpreted as a dot product in a high-dimensional space. Recal SVMs and the kernel trick.

Informally, a kernel is a function of two objects that quantifies their similarity. Mathematically, it corresponds to an inner product in a reproducing kernel Hilbert space (Schölkopf and Smola, 2002).
Graph kernels are instances of the family of so-called R-convolution kernels by [**1999_Haussler_CONF**]. R-convolution is a generic way of defining kernels on discrete compound objects by comparing all pairs of decompositions thereof. Therefore, a new type of decomposition of a graph results in a new graph kernel.
Given a decomposition relation R that decomposes a graph into any of its subgraphs and the remaining part of the graph, the associated R-convolution kernel will compare all subgraphs in two graphs. However, this all subgraphs kernel is at least as hard to compute as deciding if graphs are isomorphic [**1979_Garey_BOOK**]. Therefore one usually restricts graph kernels to compare only specific types of subgraphs that are computable in polynomial runtime.

Definition 3.3: Graph kernel

A **graph kernel** is a symmetric, positive semidefinite function $k : \mathcal{G} \times$

$\mathcal{G} \to \mathbb{R}$ on the set of graphs $\mathcal{G}$, such that there exists a map $\phi : \mathcal{G} \to \mathcal{H}$ into a Hilbert space $\mathcal{H}$ such that

$$k(G_i, G_j) = \langle \phi(G_i), \phi(G_j) \rangle_{\mathcal{H}}$$

for all $G_i, G_j \in \mathcal{G}$ where $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is the inner product in $\mathcal{H}$.

Basically, a graph kernel is a function that measures the similarity of two graphs. https://ysig.github.io/GraKeL/0.1a8/documentation/introduction.html

Definition 3.4: Weisfeiler-Lehman kernel

Let k be any positive semi-definite base kernel on graphs, that we will call the **base kernel**. Then the **Weisfeiler-Lehman kernel** (**WL-kernel**) with h iterations with k is defined as

$$k_{\text{WL}}^{(h)}(G, G') = \sum_{i=0}^{h} k(G_i, G_i')$$

($\{G_0, \dots, G_h\}$ and $\{G_0', \dots, G_h'\}$ are the WL-sequences of G and G' respectively.)
It counte the common multiset strings (WL features) in two graphs [**2009_Shervashidze_NIPS**]:

$$k_{\text{WL}}^{(h)}(G, G') = \left|\{(s_i(v), s_i(v')) | f\left(s_i(v)\right) = f\left(s_i(v')\right), i \in \{1, \dots, h\}, v \in V, v' \in V'\}\right|$$

A more general definition can be derived with weights on the GL-graph iterations $0 \leq \alpha_i \in \mathbb{R}$:

$$k_{\text{WL}}^{(h)}(G, G') = \sum_{i=0}^{h} \alpha_i k(G_i, G_i')$$

This allows for example to emphasize larger substructures - when for example labels in higher iterations contribute more to the overall similarity.[**2021_Schulz_CONF**]

Definition 3.5: Ramon-Gärtner subtree kernel

This kernel compares all pairs of vertices by iteratively comparing their neighborhoods. Let $G = (V, E, \ell)$ and $G = (V', E', \ell')$ be two graphs. The

Ramon-Gärtner subtree kernel $k^{(h)}_{\text{Ramon}}$ is defined as

$$k^{(h)}_{\text{Ramon}}(G, G') = \sum_{v \in V} \sum_{v' \in V'} k_h(v, v')$$

where

$$k_h(v, v') = \begin{cases} \delta\big(\ell(v), \ell'(v')\big) & \text{if } h = 1 \\ \lambda_r \lambda_s \sum_{R \in \mathcal{M}(v,v')} \prod_{(w,w') \in R} k_{h-1}(w, w') & \text{if } h > 1 \end{cases}$$

and

$$\mathcal{M}(v, v') = \{R \subseteq \mathcal{N}(v) \times \mathcal{N}(v') | \big(\forall (u, u'), (w, w') \in R : u = w \iff u' = w'\big) \wedge \big(\forall (u, u') \in R : \ell(u) = \ell'(u')\big)\}$$

Intuitively, the kernel iteratively compares all matching $\mathcal{M}(v, v')$ between neighbors of two vertices v from G and v' from G' [**2003_Ramon_CONF**, **2009_Shervashidze_NIPS**].

This kernel is the first defined subtree kernel [**2009_Shervashidye_NIPS**].
Let both graphs have n vertices, m edges, a maximum degree of d and let h be the height of the subtree. Then the runtime complexity of the subtree kernel for one pair of graphs is $\mathcal{O}(n^2 h 4^d)$, including a comparison of all pairs of vertices (n^2) and a pairwise comparison of all matchings in their neighborhoods in $\mathcal{O}(4^d)$, which is repeated in h iterations.
This runtime is not feasible for larger graphs. See the fast subtree kernel of Shervashidze and Borgwardt instead [**2009_Shervashidye_NIPS**].

3.3.0.0.1 Example of a WL-kernel: Consider as base kernel the subtree-kernel k:

$$k(G_i, G'_i) = \sum_{v \in V} \sum_{v' \in V'} \delta(\ell_i(v), \ell_i(v'))$$

Here δ is the Kronecker delta. The WL-kernel k^h_{WL} on this base kernel simply counts the pairs of matching labels of all h WL-iterations.
With complexity $\mathcal{O}(h|E(G)|)$ the WL subtree kernel is highly efficient.[**2021_Schulz_CONF**]

Definition 3.6: Weisfeiler-Lehman subtree kernel

Let G and G' be graphs. Define $\hat{\Sigma}_i \subseteq \Sigma$ as the set of labels that occur at least one in G or G' after iteration i of algorithm **??**. Let Σ_0 be the set of

original vertex labels of G and G'.
Assume all Σ_i are pairwise disjoint (meaning every label occurs exactly once). Define for every i, $\Sigma_i = (\sigma_{i,1}, \dots, \sigma_{i|\Sigma_i|})$ as the ordered tuple of the set $\hat{\Sigma}_i$.
Define a map $c_i : \{G, G'\} \times \Sigma_i \to \mathbb{N}$ such that $c_i(G, \sigma_{i,j})$ is the number of occurrences of the letter $\sigma_{i,j}$ in the graph G.

The **Weisfeiler-Lehman subtree kernel** (**WL-subtree kernel**) on G and G' with h iterations is defined as:

$$k^{(h)}_{\text{WLsubtree}}(G, G') = \langle \psi^{(h)}_{\text{WLsubtree}}(G), \psi^{(h)}_{\text{WLsubtree}}(G') \rangle$$

where

$$\psi^{(h)}_{\text{WLsubtree}}(G) = \big(c_0(G, \sigma_{0,1}), \dots, c_0(G, \sigma_{0,|\Sigma_0|}), \dots, c_h(G, \sigma_{h,1}), \dots, c_h(G, \sigma_{h,|\Sigma_h|})\big)$$

(analogue for G').

That is the WL-subtree kernel counts the *common original and compressed labels* in the two graphs.

Note that due to the runtime of algorithm 8, one can compute the WL-subtree kernel in $\mathcal{O}(hm)$.

Definition 3.7: Relaxed Weisfeiler-Lehman subtree kernel

Let $\mathcal{G}$ be a set of graphs and Θ_i a set of hard clustering functions (i.e. partitionings) of the set of depth-i unfolding trees $\mathcal{T}^{(i)}$ appearing in the graphs in $\mathcal{G}$.

Regard each element of Θ_i as a function $\rho : \mathcal{T}(i) \to [k]$, where k is the number of clusters defined by ρ.
Then, for any graphs $G, G' \in \mathcal{G}$ and depth parameter $h \in \mathbb{N}$, the **relaxed Weisfeiler-Lehman subtree kernel** (**R-WL-subtree kernel**) is defined as:

$$k^h_{\text{R-WL}}(G, G') = \sum_{i=0,\dots,h} \sum_{\rho \in \Theta_i} \sum_{v \in V} \sum_{v' \in V'} \delta\Big(\rho\big(T^i(G, v)\big),\ \rho\big(T^i(G', v')\big)\Big)$$

(where δ is the Kronecker delta). **[2021_Schulz_CONF]**

Clearly $k^h_{\text{R-WL}}(G, G')$ is positive semi-definite and equivalent to the original WL-subtree kernel for $\Theta_i = \{\rho_i\}$.

To compute the WL-subtree kernel on N graphs, see algorithm 3.

Idea: Do not apply the WL-subtree kernel definition (definition 3.6) N^2-fold. Instead, process all N graphs simultaneously and conduct the steps in algorithm 8 on each graph G in each iteration.

Algorithm 3 Weisfeiler-Lehman subtree kernel computation on N graphs - one iteration

Input: two labeled graphs $G = (V, E, \ell)$, $G' = (V', E', \ell')$ (w.l.o.g. $\ell = \ell' : V \cup V' \to \Sigma$), the set of all possible labels Σ (ordered and finite, ideally $|\Sigma| \approx Nh(h+1)$ at iteration h). (sufficiently large, to allow f to be injective)
Output: the two graphs re-labeled: $G = (V, E, \hat{\ell})$, $G' = (V', E', \hat{\ell}')$.

```
1: Multiset-label determination:
     • Assign a multiset-label $M_i(v)$ to each vertex $v$ in $G$ which consists of
       the multiset $\{l_{i-1}(u)|\ u \in \mathcal{N}(v)\}$
2: Sorting each multiset:
     • Sort elements in $M_i(v)$ in ascending order and concatenate them into
       a string $s_i(v)$.
     • $s_i(v) := l_{i-1}(v)\ s_i(v)$
3: Label compression:
     • Map each string $s_i(v)$ to a new compressed label, using any injective
       function $f : \Sigma^* \to \Sigma$, e.a.
         $$f(s_i(v)) = f(s_i(w)) \iff s_i(v) = s_i(w)$$
4: Relabeling:
     • $\forall v \in G \cup G' :\ l_i(v) := f(s_i(v))$.
5: if $\{l_i(v)|\ v \in V\} \neq \{l_i(v')|\ v' \in V'\}$ then
6:     Terminate: "'$G$ is not identical to $G'$"'
7: else if $i == n$ then
8:     "'$G$ and $G'$ are (most likely) isomorphic"'.[1992_Cai_CONF][1979_Babai_CONF]
9: else
10:    Continue labeling ($i \to i+1$).
```

Note that feasible definitions for f is to sort all neighborhood strings using radix sort. The resulting complexity of this step would be linear in the sum of the size of the current algabet and the total length of strings ($\mathcal{O}(Nn + Nm) = \mathcal{O}(Nm)$).
An alternative implementation of f would be by means of a perfect hash function.

Time complexity: $\mathcal{O}(Nhm + N^2hn)$ - where $m = |E|$ and h is the number of iterations.

Definition 3.8: Ramon-Gärtner subtree kernel

[2003_Ramon_CONF] For two graphs $G = (V, E, \ell)$ and $G' = (V', E', \ell)$ define the following function which compares all matchings $\mathcal{M}(v, v')$ between neighbors of two identically labeled vertices $v \in G$ and $v' \in G'$ with associated weights $\lambda_v, \lambda_{v'}$:

$$k_{\mathrm{RG},h}(v, v') := \begin{cases} \delta\big(\ell(v), \ell(v')\big) & \text{if } h = 0 \\ \lambda_v \lambda_{v'} \delta\big(\ell(v), \ell(v')\big) \sum_{R \in \mathcal{M}(v,v')} \prod_{(w,w') \in R} k_{\mathrm{RG},h-1}(w, w') & \text{if } h > 0 \end{cases}$$

where

$$\mathcal{M}(v, v') := \left\{ R \subseteq \mathcal{N}(v) \times \mathcal{N}(v') \middle| \begin{array}{ll} \forall (u,u'),(w,w') \in R: & u = w \iff u' = w' \\ \wedge \forall (u, u') \in R: & \ell(u) = \ell(u') \end{array} \right\}$$

(That is the set of exact matchings of subsets of the neighborhoods of v and v'.)
Now define the **Ramon-Gärtner subtree kernel** as

$$k_{\mathrm{RG}}^{(h)}(G, G') := \sum_{v \in V} \sum_{v' \in V'} k_{\mathrm{RG},h}(v, v')$$

The runtime complexity of this subtree kernel is $\mathcal{O}(n^2 h 4^d)$: h times a

- comparison of all pairs of vertices (n^2) and
- a pairwise comparison of all matchings in their neighborhoods in $\mathcal{O}(4^d)$

For a dataset of N graphs, the resulting runtime complexity is then $\mathcal{O}(N^2 n^2 h 4^d)$.

3.3.0.0.2 WL Edge Kernel In the case of graphs with unweighted edges, we consider the base kernel that counts matching pairs of edges with identically labeled endpoints (incident vertices) in two graphs. In other words, the base kernel is defined as

...

3.3.0.0.3 WL Shortest Path Kernel

Definition 3.9: Wasserstein Weisfeiler-Lehman kernel

Given a set of graphs $\mathcal{G} = \{G_1, \dots, G_N\}$ and the GWD defined for each pair of graphs on their WL embeddings, we define the **Wasserstein Weisfeiler-Lehman** (WWL) kernel as

$$K_{\mathrm{WWL}} = \exp\left(-\lambda D_W^{f_{WL}}\right)$$

Laplacian kernel > favourable conditions for positive definiteness with non-Euclidean distances: **Categorical WWL is positive definite** for all $\lambda > 0$
(Open problem for continuous WWL!!)

(Wasserstein distance is not isometric/ no metric-preserving mapping to an L^2-norm / thus not necessarily possible to derive a positive definite kernel. Laplacian kernel fortunately did this.)

For continuous WWL - treat as if indefinite kernel:

- (reproducing kernel) **Krein spaces** (RKKS)
  (Generalisation of reproducing kernel Hilbert spaces)

- Krein SVM (KSVM) as classifier

3.4 Algorithms

3.4.1 DFS

Algorithm 4 Depth First Search (DFS)

Input: A graph G,
a start vertex $s \in V$

Output: All vertices in DFS order, starting from s.
The algorithm is usually not used to give an output,
but rather to traverse edges and vertices in a structured way.

```
1:
```

Time complexity: $\mathcal{O}(n + m)$

Remarks:

- The augmented version of DFS can be used to perform other tasks. For example:
  - Count connected components
  - Compute a MST
  - Detect and find cycles
  - Check if the graph is bipartite
  - Find strongly connected components
  - Topologically sort the vertices
  - Find bridges and articulation points
  - Find augmenting paths in a flow network
  - Generate mazes
- **Count connected components**: Start a DFS run at every vertex, if it has not already been visited. For each DFS run, increase a counter and label the visited vertices of that run with the current counter value.
  The final counter indicates the number of connected components. All vertices of one connected component have the same counter value.

An implementation of the DFS algorithm using an adjacency list can look as follows:

```
1: $n = |V|$
2: Let $g$ be the adjacency list representing the graph
3: $v = [0]^n$                                          ▷ Visited array of size $n$
4: **procedure** DFS($i$)
5:     **if** $v[i] == 1$ **then**
6:         **return True**
7:     $v[i] = 1$
8:     $\mathcal{N} = g[i]$
9:     **for** $n \in \mathcal{N}$ **do**
10:        DFS($n$)
```

3.4.2 DFS

Algorithm 5 Breadth First Search (BFS)

Input: A graph G,
a start vertex $s \in V$

Output: All vertices in BFS order, starting from s.
The algorithm is usually not used to give an output,
but rather to traverse edges and vertices in a structured way.
It is particularly useful to find shortest paths.

BFS explores graphs in a layered fashion.

```
1:
```

Time complexity: $\mathcal{O}(n + m)$

Remarks:

- Using BFS or its augmented version can be used to perform many tasks. For example:
  -
-

3.4.3 Topological sort

Algorithm 6

Input: A digraph $G = (V, E)$.
Output: A topological ordering on V.

```
1: while Not all vertices have been visited do
2:     Pick an unvisited vertex $v$.
3:     Start DFS from $v$ (on unvisited vertices).
4:     On all recursive callback of the DFS, add the current vertex to the topo-
   logical ordering in reverse order.
```

Time complexity: $\mathcal{O}(n + m)$

The algorithm is useful for applications where vertices in directed graphs encode events which must occur before others. Examples are:

- School class prerequisites
- Program dependencies
- Event scheduling
- Assembly instructions

If G is a **tree**, the task becomes trivial. A topological order can be constructed by iteratively selecting all leaves (in arbitrary order), and then removing them from the graph.

3.4.4 EXAMPLE ALG

Algorithm 7

Input:
Output:

```
1:
```

Time complexity:

Remarks:

-

3.4.5 Graph isomorphism

The graph isomorphism problem is equivalent to computing the order of automorphism groups of graphs.**[1979_Mathon]**

3.4.5.1 Weisfeiler-Lehman test of isomorphism

1-dimensional variant ("′ naive vertex refinement"′). Algorithm 1. Augment the vertex labels by the sorted set of vertex labels of neighboring vertices, and compress these augmented labels into new, short labels. Repeat, until the set of vertex labels for the two compared graphs differ.

Algorithm 8 Weisfeiler-Lehman test of isomorphism - one iteration

```
Input:  two labeled graphs $G = (V, E, \ell)$, $G' = (V', E', \ell')$ (w.l.o.g. $\ell = \ell' : V \cup V' \to \Sigma$),
        the set of all possible labels $\Sigma$ (ordered and finite, ideally $|\Sigma| = 2|V|$).
        (sufficiently large, to allow $f$ to be injective)
Output: the two graphs re-labeled: $G = (V, E, \hat{\ell})$, $G' = (V', E', \hat{\ell}')$.

 1: Multiset-label determination:
      • For $i = 0$, set $M_i(v) := l_0(v) = \ell(v)$.
      • For $i > 0$, assign a multiset-label $M_i(v)$ to each vertex $v$ in $G$ and $G'$
        which consists of the multiset $\{l_{i-1}(u) | \; u \in \mathcal{N}(v)\}$.
 2: Sorting each multiset:
      • Sort elements in $M_i(v)$ in ascending order and concatenate them into
        a string $s_i(v)$.
      • $s_i(v) := l_{i-1}(v) \; s_i(v)$
 3: Label compression:
      • Sort all of the strings $s_i(v)$ for all $v$ from $G$ and $G'$ in ascending order.
      • Map each string $s_i(v)$ to a new compressed label, using any injective
        function $f : \Sigma^* \to \Sigma$, e.a.

            $f(s_i(v)) = f(s_i(w)) \iff s_i(v) = s_i(w)$

 4: Relabeling:
      • $\forall v \in G \cup G' : \; l_i(v) := f(s_i(v))$.
 5: if $\{l_i(v) | \; v \in V\} \neq \{l_i(v') | \; v' \in V'\}$ then
 6:     Terminate: "'$G$ is not identical to $G'$"'
 7: else if $i == n$ then
 8:     No answer. Or: "'$G$ and $G'$ are (most likely)
    isomorphic"'.[1992_Cai_CONF][1979_Babai_CONF]
 9: else
10:     Continue labeling ($i \to i + 1$).
```

Note that feasible definitions for both M_i and f arise naturally from the order in Σ. To construct f simply keep a counter variable that records the number of distinct strings, that f has compressed before. When the input for f as already been assigned at some time, f assigns it to the current state of the counter. If f gets a new string, the counter is incremented and f assigns its value to the new string.

The order on Σ guarantees, that all identical strings are mapped on the same number because they occur in a consecutive block.

Time complexity: $\mathcal{O}(mh)$ - where $m = |E|$ and h is the number of iterations.

Remarks:

- For unlabeled graphs, initialize $M_0(v) := l_0(v) = |\mathcal{N}(v)|$ (since two ver-

Instead of sorting the multiset of the WL-labels of the neighborhood, one may alternatively indicate the carnality of each such label in the multiset. The 1-dimensional WL-method also considers all other vertices that are not in the neighborhood of the current vertex v: $V(G)\backslash\mathcal{N}(v)$. The new color in iteration $r+1$ of v is thus defined as

$$\text{hash}\left(\ell^r(v), a_1, b_1, \ldots, a_n, b_n\right)$$

where a_i is the number of vertices of color i that v is adjacent to ($a_i = \left|\left\{w|\ w \in \mathcal{N}(v) \wedge \ell^r(w) = i\right\}\right|$) and b_i is the number of vertices of color i that v is not adjacent to ($b_i = \left|\left\{w|\ w \notin \mathcal{N}(v) \wedge \ell^r(w) = i\right\}\right|$) **[1992_Cai_IEEE]**.

If the next refinement step yields the same WL-label separation, a **stable refinement** is reached.

The 1-dimensional WL-method is also called **vertex refinement**.

The $(k-1)$-dimensional WL-method is equivalent to testing $\mathcal{C}_k$ equivalence (where $\mathcal{C}_k$ is the class of first-order graph theory languages with counting quantifiers) [**1992_Cai_IEEE**]. This is possible in time $n^k \log(n)$. The 1-dimensional WL-method identifies (is able to test for isomorphism) all trees and almost all graphs [**1992_Cai_IEEE**].

3.5 Distances

3.5.1 Wasserstein distances

Definition 5.1: Continuous Wasserstein distance

The (continuous) **Wasserstein distance** W_p (also called earth mover distance) is a distance function between probability distributions defined on a given metric space.
Let σ and μ be two probability distributions on a metric space M equipped with a ground distance d such as the Euclidean distance. For $p \in [1, \infty)$ it is:

$$W_p(\sigma, \mu) := \Big(\inf_{\gamma \in \Gamma(\sigma,\mu)} \int_{M \times M} d(x,y)^p d\gamma(x,y) \Big)^{\frac{1}{p}}$$

where $\Gamma(\sigma, \mu)$ is the set of all transportation plans $\gamma \in \Gamma(\sigma, \mu)$ over $M \times M$ with marginals σ and μ on the first and second factors respectively.

d is a metric $\implies$ W_p is a metric (Villani [44], chapter 6, for a proof)
Focus on $p = 1$ (L^1-Wasserstein distance).

Definition 5.2: Discrete Wasserstein distance

For two finite sets of vectors $X \in \mathbb{R}^{n \times m}$ and $X' \in \mathbb{R}^{n' \times m}$ (vertex embeddings) with $\forall x, x' : |x|_1 = |x'|_1$ the (discrete) **Wasserstein distance** is defined as:

$$W_M(X, X') := \min_{P \in \Gamma(X, X')} \langle P, M \rangle$$

M is a **cost** or **distance matrix** containing $d(x, x')$ for some metric d.
$P \in \Gamma$ - **transport matrix** (or joint probability). Contains fractions that indicate how to transport the values from X to X' with minimal total transport effort. Because we assume that the total mass to be transported equals 1 and is evenly distributed across the elements of X and X', the row and column values of P mus sum up to $\frac{1}{n}$ and $\frac{1}{n'}$ respectively.
$\langle \cdot, \cdot \rangle$ is the Frobenius inner product.
For a set of vectors $x_1, \ldots, x_n \in \mathbb{R}^n$ and a cost matrix $C^{n \times n}$ the **barycenter** of the vectors is defined as

$$\underset{c}{\operatorname{argmin}} \sum_{i \in [n]} W_C(x_i, c)$$

Chapter 4

Problems

This chapter contains definitions of difficult problems, that are concerned with the relations between defined objects and properties. Easier problems, that are concerned with constructive questions (for example, how to compute a spanning tree), may also be found tailing the related definition in chapter 3.

4.0.1 Solved

4.0.1.1 Shortest path problem

4.0.1.1.1 Problem definition

Input: A (weighted graph) $G = (V, E, \ell)$,
a start vertex $s \in V$,
a target vertex (or end vertex) $t \in V$.
Output: The shortest (meaning the cheapest) s-t-path.

4.0.1.1.2 Solutions - Overview

- BFS (for an unweighted graph)
- Dijkstra's algorithm
- Bellman-For algorithm
- Floyd-Warshall algorithm
- A^*

4.0.1.2 Connectivity problem

4.0.1.2.1 Problem definition

Input: A (weighted graph) $G = (V, E, \ell)$.
Output: The decision if G is connected. (That is there exists a (directed) path between any two vertices.

4.0.1.2.2 Solutions - Overview Typical solutions use the union find data structure or apply multiple runs of (path) search algorithms (like DFS or BFS).

4.0.1.3 Strongly connected components problem

4.0.1.3.1 Problem definition

Input: A digraph $G = (V, E, \ell)$.
Output: A vertex set labeling, indicating subsets of strongly connected components.

4.0.1.3.2 Solutions - Overview

- Tarjan's and Kosaraju's algorithm
-

4.0.1.4 Negative cycles problem

4.0.1.4.1 Problem definition

Input: A weighted graph $G = (V, E, \ell)$.
Output: The decision if there exists a cycle with negative edge weight sum in G

Within the task of finding a shortest path, negative cycles can be a trap in the sense that traversing it infinitely reduces the costs infinitely.

4.0.1.4.2 Solutions - Overview

- Bellman-Ford
- Floyd-Warshall

4.0.1.5 Bridge problem

4.0.1.5.1 Problem definition

Input: A graph $G = (V, E)$.
Output: The set of bridges $B \subset E$ in G.

4.0.1.6 Articulation point problem

4.0.1.6.1 Problem definition

Input: A graph $G = (V, E)$.
Output: The set of articulation points $A \subset V$ in G.

4.0.1.7 Minimum spanning tree (MST)

4.0.1.7.1 Problem definition

Input: A weighted graph $G = (V, E, \ell)$.
Output: A MST T of G.

4.0.1.7.2 Solutions - Overview

- Kruskal's algorithm
- Prim's algorithm
- Boruvka's algorithm

4.0.1.8 Network flow - max flow

4.0.1.8.1 Problem definition

Input: A weighted graph $G = (V, E, \ell)$,
a source $s \in V$,
a sink $t \in V$.
Output: The maximum s-t-flow.

4.0.1.8.2 Solutions - Overview

- Ford-Fulkerson algorithm
- Edmonds-Karp algorithm
- Dinic's algorithm

4.0.2 Partially solved

- Graph isomorphism problem - Subgraph isomorphism problem - largest common subgraph

4.0.2.1 Traveling Salesman problem (TSP)

4.0.2.1.1 Problem definition

Input: A (weighted) graph $G = (V, E)$.
Output: A shortest (cheapest) cycle of size n_G that does not contain sub-cycles.

The output is a route that visits every vertex exactly once and returns to the origin vertex.
This problem is NP-hard.

4.0.2.1.2 Solutions - Overview

4.0.2.1.3 Similarity measures based on exact matchings Due to the computational costs of many solutions, many approximation algorithms exist.

- Helf-Karp
- Branch and bound
- Ant colony optimization

4.0.2.2 Graph similarity measures

4.0.2.2.1 Problem definition Define a suitable similarity measure between graphs.

4.0.2.2.2 Solutions - Overview

4.0.2.2.3 Similarity measures based on exact matchings Examples of graph similarity measures based on **exact matchings** are based on:

- graph isomorphism (topological identical),
- subgraph isomorphism or
- largest common subgraph.

These measures are quite restrictive in the sense that graphs have to be exactly identical or contain large identical subgraphs in order to be deemed similar by these measures. **[2011_Shervashidze_JMLR]**

4.0.2.2.4 Similarity measures based on inexact matchings Examples of graph similarity measures based on **inexact matchings** are based on:

- graph edit distance [**1983_Bunke_ELSEVIER**, **2005_Neuhaus_IEEE**] (hard to parameterize; involves solving NP-complete problems as intermediate steps)
- optimal assignment kernels [**2005_Froehlich_ICML**] (best match between substructures; these kernels are in general not positive semidefinite [**2018_Vert_CONF**])
- skew spectrum [**2008_Kondor_ICML**] (unlabeled graphs; polynomial time)
- graphlet spectrum [**2009_Kondor_ICML**] (difficult to parameterize on general labeled graphs)

[**2011_Shervashidze_JMLR**]
Problems with graph kernels:

1. Simple aggregation of the similarities of the substructures might limit the ability to capture complex characteristics of the graph (Solutions: Fröhlich et a. [15], Kriege et al. [25])
2. Most proposed variants do not generalist to graphs with high-dimensional continuous vertex attributes

[**2011_Shervashidze_JMLR**]

4.0.3 Not solved yet

4.0.4 Not solvable

4.0.5 Theorems on connections between problems

Chapter 5

Theorems

5.1 Graphs

THEOREM 1.1 - Bound on SDM sequence lengths:

Let (M, T, T') be an SDM and $T_0 = T, T_1, \ldots, T_k$ be a sequence of trees obtained by the above atomic transformations such that **every** $v \in T$ and $v' \in T'$ has been considered **in exactly one transformation**. Then $T_k = T'$.**[2021_Schulz_CONF]** (Note that mapping a vertex in one tree to a vertex in another tree with the same label is considered as a *relabeling operation* with cost zero!)

5.1.0.0.1 Proof: Proof by contradiction. Assume that there is such a sequence, but $T_k \neq T'$. This implies that either

1. there exists a vertex v which has the wrong label than the corresponding vertex in T'.

2. there exists a vertex v in T_k but not in T',

3. there exists a vertex v in T' but not in T_k or

By assumption, this vertex v was involved in exactly one atomic operation. If the operation was

- **relabeling**, case 1 can not be the case, since the operation prevents it by definition. Case 2 can not be the case, since then there was no point in relabeling v - it rather would have been deleted. Case 3 also can not be the case, since then v did not exists in T_k and could not have been relabeled.

- **deletion**, case 2 can not be the case since then v does not exists in T' as desired. Case 1 and 2 also cannot be the case, since v does no longer exist in T' and there cannot be a corresponding vertex.
- **insertion**, case 3 can not be the case since then v exists in T' as desired. Recall that v was involved in exactly one operation, thus v could not have been deleted after the insertion (in a second operation). Case 2 can not be the case, since v was never deleted. And case 1 can also not be the case, since v is inserted with a suitable label. Thus there cannot be a need for relabeling.

This implies, that due to the initial assumption, non of the above cases can be true and thus $T_k = T'$ as claimed. □

THEOREM 1.2 - Reduction of optimal SDM to a optimal bipartite matching:

The task of finding an optimal SDM (that is minimal w.r.t. some cost function) can be reduced to the minimum cost perfect bipartite matching problem. **[2021_Schulz_CONF]**
(This states the correctness of algorithm 1.)

5.1.0.0.2 Proof: Let T and T' be two labeled rooted trees. Let the sets of trees below the roots of these trees be

$$\tilde{F} = \{T_1, \dots, T_k\} \qquad \text{and} \qquad \tilde{F}' = \{T'_1, \dots, T'_{k'}\}$$

Expand the sets by adding k' empty trees to $\tilde{F}$ and k to $\tilde{F}'$. Now both expanded sets, call them F and F' have $k + k'$ elements:

$$F = \{T_1, \dots, T_k, T_\perp, \dots, T_\perp\}, \qquad F' = \{T'_1, \dots, T'_{k'}, T_\perp, \dots, T_\perp\}$$

Define the distance between a tree and an empty graph as the cost of deleting, resp. inserting that tree. Furthermore, two empty graphs have distance 0.
Claim: The optimal set of SDMs directly corresponds to a perfect bipartite matching of minimum cost between tress in the expanded sets F and F' with

such a distance.
Reasoning: This follows from theorem **??** and the minimization constraint on both ends (triangle equality of the distance/metric).

Finally the SDTED between T and T' is the cumulative cost of the distance between their roots and the minimal cost perfect bipartite matching between the trees below them. □

5.1.1 Trees

THEOREM 1.3 - Three edit metric:
If γ is a metric, then the tree edit distance with cost function γ is a metric too.[**2021_Schulz_CONF**]

5.1.1.0.1 Proof: □

THEOREM 1.4 - Difficulty of the tree edit distance:
Calculating the tree edit distance is **NP-hard**. [**2005_Bille_TCS**]

5.1.1.0.2 Proof: □

THEOREM 1.5 - Bijektion between labels and unfolding trees:
There exists a bijektion between labels in Σ_i and the set of (pairwise non-isomorphic) i-unfolding trees.

5.1.1.0.3 Proof: □

Note that the i-unfolding trees of most vertices will be unique for very small values of i. Thus two structurally completely different unfolding trees are treated identically to two unfolding trees which differ by only very little.

5.2 Kernels

5.2.1 Weisfeiler-Lehman graph kernels

Weisfeiler-Lehman graph kernels [2011_Shervashidze_JMLR] is a family of efficient graph kernels (for graphs with discrete vertex labels). It can be understood as a meta graph kernel, since it is constructed by using any graph kernel on the same graph with multiple labelings, given by the Weisfeiler-Lehman labeling scheme. The positive definiteness of the base graph kernel is preserved.

THEOREM 2.1 - Positive semidefiniteness of the Weisfeiler-Lehman kernel:
Let the base kernel k be any positive semidefinite kernel on graphs. Then the corresponding WL-kernel $k_{\mathrm{WL}}^{(h)}$ is positive semidefinite.

5.2.1.0.1 Proof: Let ϕ be the feature mapping corresponding to the kernel k:

$$k(G_i, G_i') = \langle \phi(G_i), \phi(G_i') \rangle$$

It is:

$$\begin{aligned} k(G_i, G_i') &= k(r^i(G), r^i(G')) \\ &= \langle \phi\big(r^i(G)\big), \phi\big(r^i(G')\big) \rangle \\ &= \langle \psi(G), \psi(G') \rangle\rangle && \text{defining: } \psi(G) := \phi\big(r^i(G)\big) \end{aligned}$$

Hence k is a kernel on G and G' and $k_{\mathrm{WL}}^{(h)}$ is positive semidefinite as a sum of positive semidefinite kernels.

The positive semidefiniteness for the more general case (with weights α_i) holds, since the property is invariant under positive linear combinations. □
Thus the definition of the WL-kernel (definition 3.4) provides a framework for applying all positive semidefinite graph kernels that take into account discrete vertex labels to different levels of the vertex-labeling of graphs.

Side-effects of the WL-graphs for the used kernel k:

- The graph structures (V, E) remain constant.

Thus features like shortest paths or MSTs remain constant with h.

- The alphabet of the labels increasingly grows with h.

Note that one can now easily implement a shortest path kernel, that counts pairs of shortest paths with distance d between identically labeled source and sink vertices on the original paths by running a shortest path kernel with shortest paths length 1 on the WL-graphs with $h = d - 1$. TODO: Check this - the WL-graphs should be even more powerful since they GROW in all directions - compare with statement after proof on page 8.

TODO: I do not understand the learning idea in 3.1.1.
Other source: **[2009_Shervashideze_NIPS]**

THEOREM 2.2 - Equivalency of the Weisfeiler-Lehman kernel and the Ramon-Gärtner kernel:
The Weisfeiler-Lehman kernel and the Ramon-Gärtner kernel are equivalent.

5.2.1.0.2 Proof: The WL kernel can be defined in a recursive fashion which elucidates its relation to the Ramon-Gärtner kernel. **[2009_Shervashidze_NIPS]**

□

THEOREM 2.3 - Similarity of the WL-kernel and -subtree kernel:

Let the base kernel k be a function counting pairs of matching vertex labels in two graphs

$$k(G, G') := \sum_{v \in V} \sum_{v' \in V'} \delta\big(\ell(v), \ell(v')\big)$$

then

$$\forall G, G' : \qquad k_{\text{WL}}^{(h)}(G, G') = k_{\text{WLsubtree}}^{(h)}(G, G')$$

5.3 Algorithms

THEOREM 3.1 - Runtime of the Weisfeiler-Lehman test of isomorphism:

he Weisfeiler-Lehman test of isomorphism (algorithm 8) has runtime $\mathcal{O}(mh)$, where $m = \max\{|E|, |E'|\}$ is the bigger number of edges in both given graphs and h is the number of iterations.

5.3.0.0.1 Proof: Lets consider the runtime for every of the sequential steps one-by-one:

1. Defining the multiset-labels $M_i(v)$ can be done in $\mathcal{O}(m)$ by sorting all multiset in $\mathcal{O}(m)$. For one iteration, this can be done as follows:
   - Use **counting sort**, since there is a given order on the finite set Σ (which can be extended to a lexicographical order).
   - $\{l_{i-1}(u) | \ u \in \mathcal{N}(v)\} \subseteq \{f(s_i(v)) | \ v \in V\}$
   - $\left|\{f(s_i(v)) | \ v \in V\}\right| \leq n$
   - Assign the elements of all multisets to their corresponding buckets and record which multiset they came from.
   - Read through all buckets in ascending order to extract the sorted multisets for all vertices.
   - As there are $\mathcal{O}(m)$ elements in the multisets of a graph in iteration i, this bounds the runtime.
   - Sorting the resulting strings has time $\mathcal{O}(m)$ via **radix sort**
   - The label compression requires one pass over all strings and their characters (again in $\mathcal{O}(m)$)

   Thus for all h iterations the runtime is $\mathcal{O}(hm)$).

$\square$

THEOREM 3.2 - Weisfeiler-Lehman subtree kernel algorithm runtime:
For N graphs, the WL-subtree kernel with h iterations on all pairs of these graphs can be computed in

$$\mathcal{O}(Nhm + N^2hn)$$

5.3.0.0.2 Proof: Naive application of the kernel from definition 3.6 for computing $N \times N$ kernel matrix would require a runtime of $\mathcal{O}(N^2hm)$. One can improve upon this runtime complexity by computing $\phi^{(h)}_{\text{WLsubtree}}$ explicitly for each graph and only then taking pairwise inner products.

1. The multiset-label determination in step one still requires $\mathcal{O}(Nm)$.

2. The sorting of the multisets in step two can be done vie bucket sort (counting sort) of all strings, requiring $\mathcal{O}(Nn + Nm)$ time.

The effort of computing $\phi^{(h)}_{\mathrm{WLsubtree}}$ on all N graphs in h iterations is then $\mathcal{O}(Nhm)$, assuming that $m > n$. To get all pairwise kernel values, we have to multiply all feature vectors, which requires a runtime of $\mathcal{O}(N^2hn)$, as each graph G has at most hn non-zero entries in $\phi^{(h)}_{\mathrm{WLsubtree}}(G)$. □

We will later see that the first factor (Nhm) dominates the overall runtime in practice.

5.4 Problems

- Graph isomorphism problem is NP. Neither proven NP-complete, nor found to be solved by a polynomial-time algorithm [**1979_Garey_BOOK**].

- Subgraph isomorphism problem is NP-complete [**1979_Garey_BOOK**].
- Largest common subgraph is NP-complete [**1979_Garey_BOOK**].

Other strategies of comparing vertex neighborhoods (other than matching the embedding scheme):

- [**2009_Hido_IEEE**] - using hash functions and logical operations on bit-representations of vertex labels (scales linearly in the number of edges)

- [**2004_Mahe_CONF**] - Morgan index

5.5 Distances

5.5.1 Wasserstein distances

THEOREM 5.1 - Discrete Wasserstein metric:
If the cost matrix is defined by a metric, then the Wasserstein distance is a metric.

5.5.1.0.1 Proof: … □

THEOREM 5.2 - Structure and depth preserving distance in terms of Wasserstein distance:

For two depth-i unfolding trees T and T', the distance between their roots is equal to $\mathcal{W}^{M_r}\left(\mathbb{V}_r(T), \mathbb{V}_r(T')\right)$.
Furthermore the calculation of the minimum cost perfect bipartite matching between the sets of child trees below these roots can be reduced to computing the Wasserstein distance $\mathcal{W}^{M_c}\left(\mathbb{V}_c(T), \mathbb{V}_c(T')\right)$. Thus it is

$$\textsc{SdTed}(T, T') = \mathcal{W}^{M_r}\left(\mathbb{V}_r(T), \mathbb{V}_r(T')\right) + \mathcal{W}^{M_c}\left(\mathbb{V}_c(T), \mathbb{V}_c(T')\right)$$

(Compare with algorithm 1.) **[2021_Schulz_CONF]**

Chapter 6

Further results and experiments of research papers

Chapter 7

LAB Notes - Weisfeiler-Lehman Graph Kernels