Graph Comparison and Classification

M. Vazirgiannis[1] G. Nikolentzos[1,2]

[1]LIX
École Polytechnique

[2]Department of Informatics
Athens University of Economics and Business

January 8, 2017

- Have become extremely popular for modelling real-world data in recent years
- Examples:
  - social networks
  - protein or gene regulation networks
  - textual documents
- Compared to vectors, graphs:
  - offer enhanced representation capabilities
  - are allowed to vary in the number of nodes and/or in the number of edges

Definition (Graph Comparison Problem)

Given two graphs G_1 and G_2 from the space of graphs $\mathcal{G}$, the problem of graph comparison is to find a mapping

$$s : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{R}$$

such that $s(G_1, G_2)$ quantifies the similarity of G_1 and G_2.

Graph comparison is a topic of high significance

- Data is often represented as graphs
- It is the central problem for all learning tasks on graphs such as clustering and classification
- Most machine learning algorithms make decisions based on the similarities or distances between pairs of instances (e.g. k-nn)

Not an Easy Problem

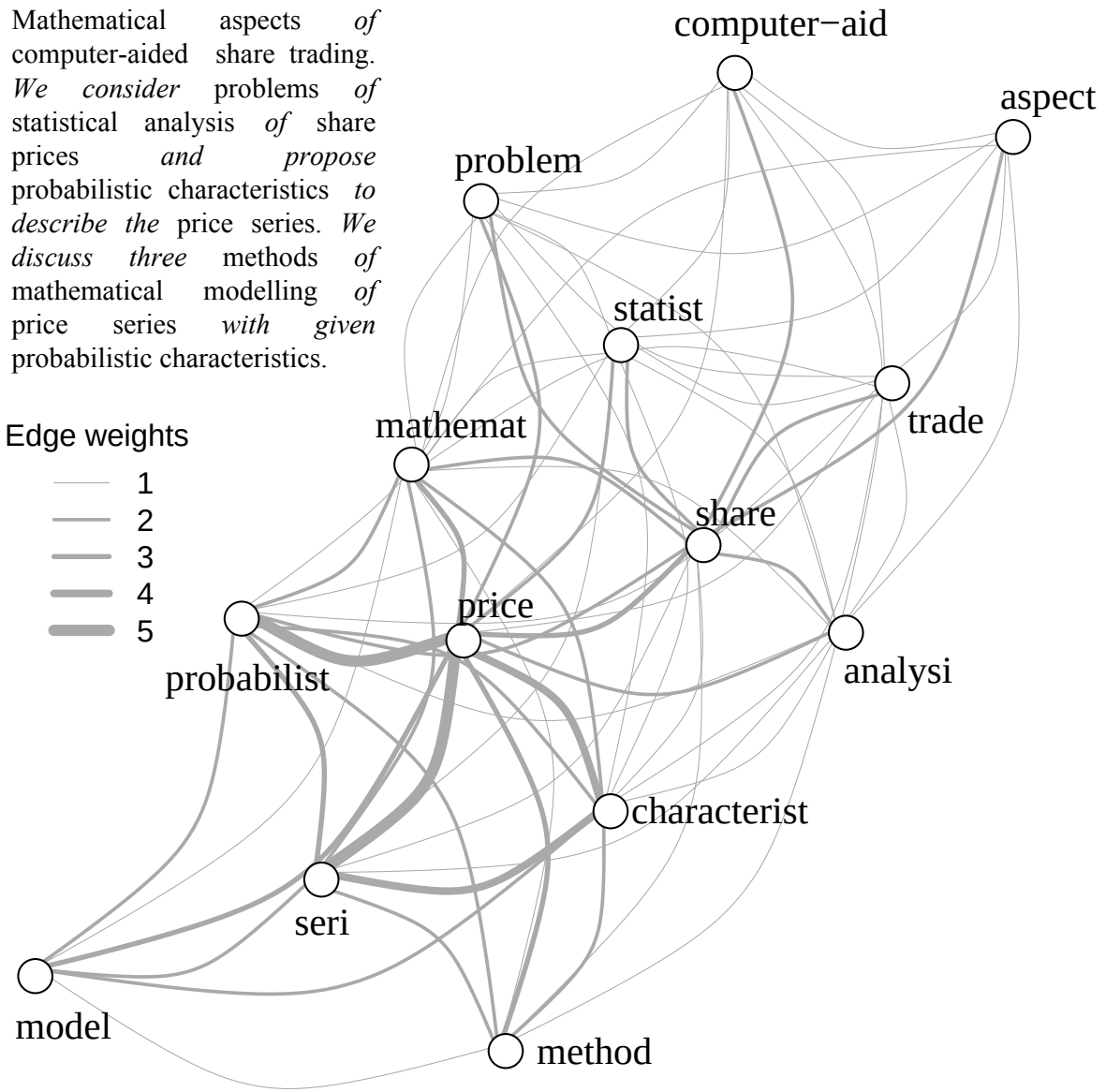
Although graph comparison seems a tractable problem, it is very complex

Many problems related to it are **NP-complete**

- subgraph isomorphism
- finding largest common subgraph

We are interested in algorithms capable of measuring the similarity between two graphs in **polynomial** time

Motivation - Text Categorization [RKV15]



Given a text, create a graph where

- vertices correpond to terms
- two terms are linked to each other if they co-occur within a fixed-size sliding window

Intuition: documents sharing same subgraphs belong to the same class

Given a set of documents and their graph representations:

Extract frequent subgraphs

- from the set of graphs

  or

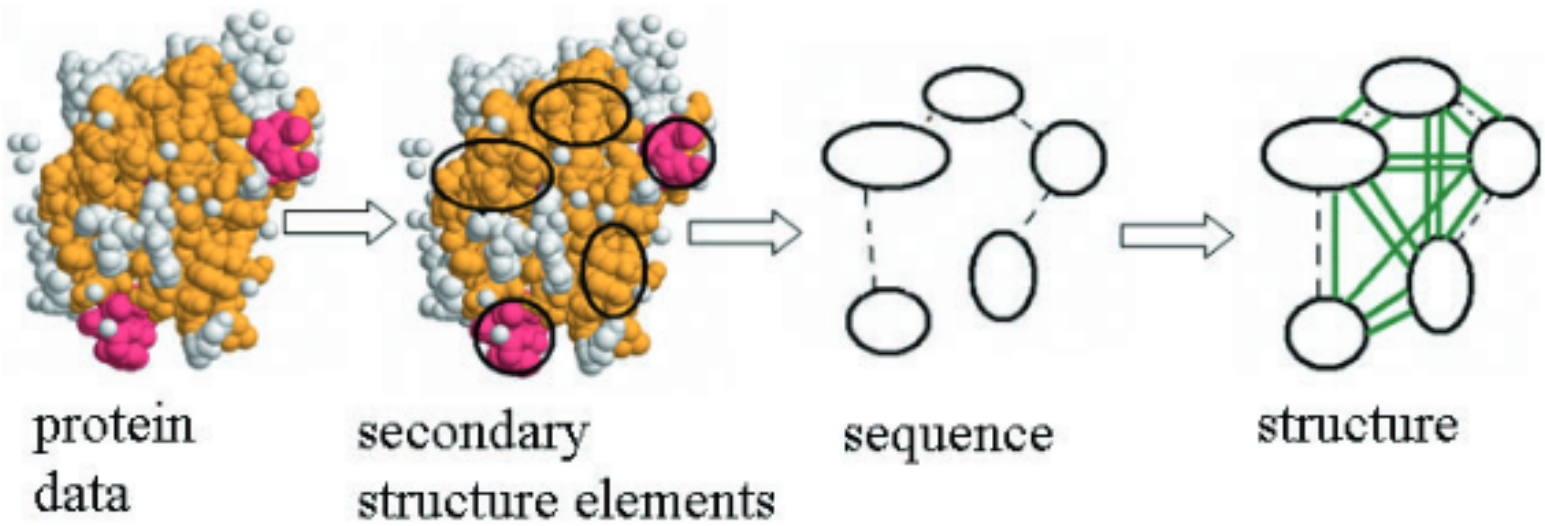
- from the set of the main cores of the graphs

Then, use frequent subgraphs as features for classification

Motivation - Protein Function Prediction [BOS+05]

For each protein, create a graph that contains information about its

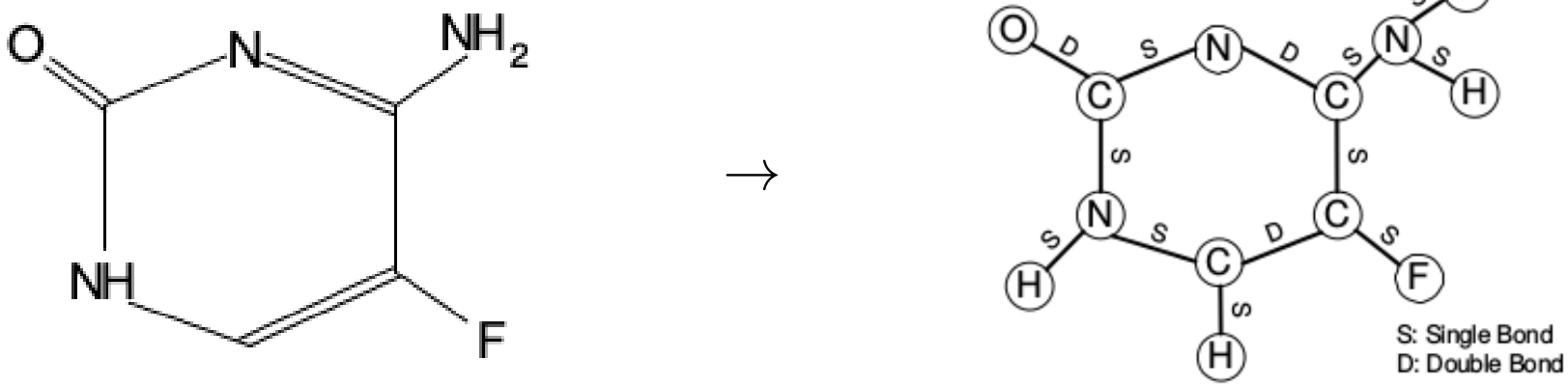
- structure
- sequence
- chemical properties



Use graph kernels to

- measure structural similarity between proteins
- predict the function of proteins

Represent each chemical compound as a graph



Use a frequent subgraph discovery algorithm to discover the substructures that occur above a certain support constraint

Perform feature selection

Use the remaining substructures as features for classification

Motivation - Anomaly Detection for the Web Graph [PDGM10]

Search engines create snapshots of the web $\rightarrow$ web graphs

These are necessary for

- monitoring the evolution of the web
- computing global properties such as PageRank

Identify anomalies in a single snapshot by comparing it with previous snapshots

Employed similarity mesaures:

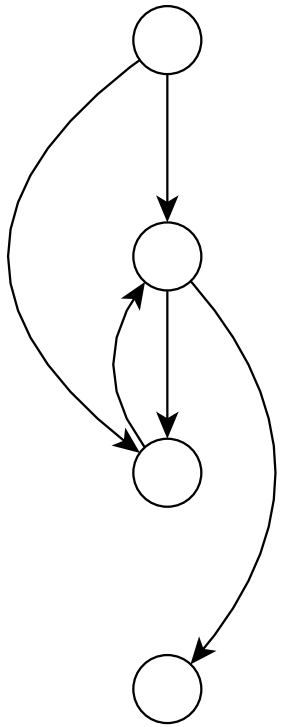
- vertex/edge overlap
- vertex ranking
- vertex/edge vector similarity
- etc

Motivation - Malware Detection [BMM06]

Given a computer program, create its control flow graph

```
            processed_pages.append(processed_page)
            visited += 1
            links = extract_links(html_code)
            for link in links:
                if link not in visited_links:
                    links_to_visit.append(link)

        return create_vocabulary(processed_pages)


def parse_page(html_code):
    punct = re.compile(r'([^A-Za-z0-9])')
    soup = BeautifulSoup(html_code, 'html.parser')
    text = soup.get_text()
    processed_text = punct.sub(" ", text)
    tokens = processed_text.split()
    tokens = [token.lower() for token in tokens]
    return tokens


def create_vocabulary(processed_pages):
    vocabulary = {}
    for processed_page in processed_pages:
        for token in processed_page:
            if token in vocabulary:
                vocabulary[token] += 1
            else:
                vocabulary[token] = 1

    return vocabulary
```

→



Compare the control flow graph of the problem against the set of control flow graphs of known malware

If it contains a subgraph isomporphic to these graphs → malicious code inside the program

Motivation – More Applications

- Bioinformatics [CHLN06, YHC06, DRO+02]
- Chemoinformatics [RSSB05, RW02]
- Comparing electrical circuits [TIK+88]
- Relational entity disambiguation [HKJ+13]
- Entity classification and link prediction [LBR12]
- Face recognition [WFKVDM97]
- Object tracking [CLL01]
- Handwriting recognition [FRB10]
- Fingerprint identification [IZ86]
- Malware Detection [GYAR13]
- Optimal task assignment in distributed systems [ST85]
- Business process model search [DDGB09]

⋮

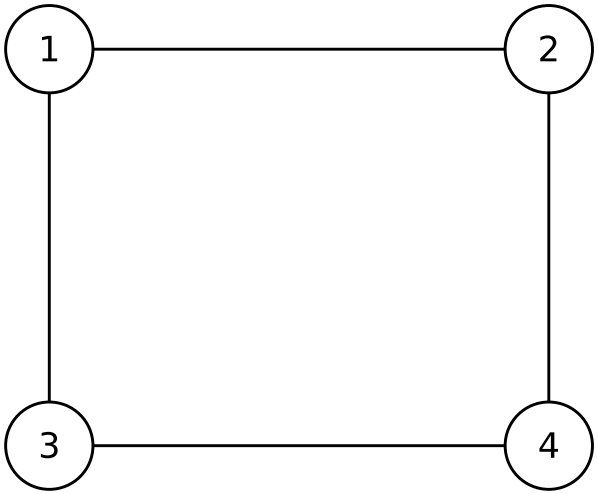
Definition (Graph Isomorphism (GI))

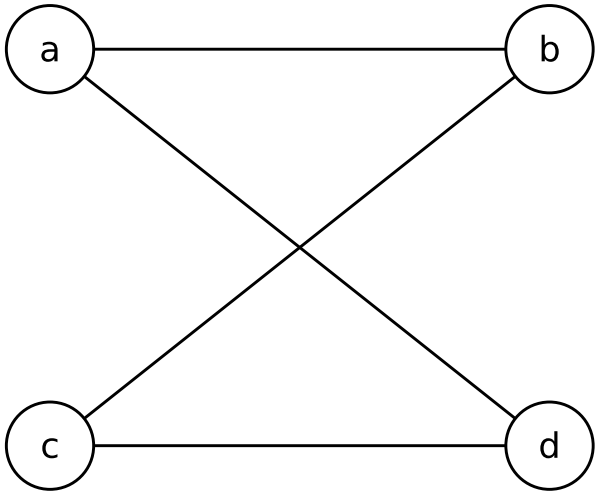
Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a bijective mapping $f : V_1 \rightarrow V_2$ such that $(v_i, v_j) \in E_1$ iff $(f(v_1), f(v_2)) \in E_2$. If the two graphs are isomorphic, we write $G_1 \cong G_2$

- Graph Isomorphism is an extreme case of the graph similarity problem
- It asks if two graphs are structurally equivalent to each other
- The problem became known during the 1960's as a method of comparing two chemical structures

Example

The following two graphs are isomorphic





The function f with

- $f(1) = a$
- $f(2) = b$
- $f(3) = d$
- $f(4) = c$

is a one-to-one correspondence between the vertices of the two graphs

Not a Trivial Problem

The following table lists the number of:

- all possible graphs with n vertices (# graphs)
- all non-isomorphic graphs with n vertices (# unique)

n	1	2	3	4	5	6	7	8
# graphs	1	2	8	64	1,024	32,768	2,097,152	268,435,456
# unique	1	2	4	11	34	156	1,044	12,346

Also, given two graphs with n vertices, not practical to solve the problem by brute force:

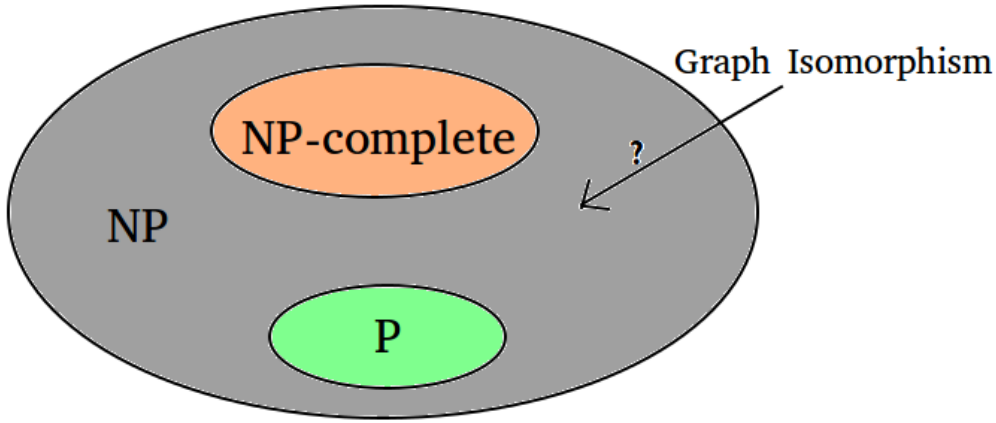
- there are $n!$ bijections between the vertices of the graphs

Hence, the need for an efficient algorithm for the GI problem is of great significance

Complexity of GI

Unfortunately, even though GI has been extensively studied:

- no *polynomial-time* algorithm is known
- neither is it known to be *NP-complete*



Schoning showed that it is located in the low hierarchy of NP [Sch88]

GI has long been a favorite target of algorithm designers. Due to its intractable nature, it was already described as a "*disease*" in 1977 [RC77].

- The best algorithms known for determining whether two graphs are isomorphic have exponential worst-case time complexity:
  - The best known algorithm for the problem is due to Babai and Luks and runs in $2^{\mathcal{O}(\sqrt{nlogn})}$ time [BL83]

- Recently, Babai presented an algorithm that runs in quasipolynomial time $2^{(\log n)^{\mathcal{O}(1)}}$ [Bab15]. Babai's algorithm improves the previous bound, however, it has not been peer reviewed yet.

- There are also available algorithms with good time complexity in many practical cases:
  - `nauty`
  - `saucy`
  - `Traces`

  based on graph invariants - also very fast..

We saw that proving that two graphs are isomorphic is not a simple task

It is much simpler to show that two graphs are not isomorphic by finding a property that only one of the two graphs has. Such a property is called a *graph invariant*

Definition (Graph Invariant)

A graph invariant is a numerical property of graphs for which any two isomorphic graphs must have the same value

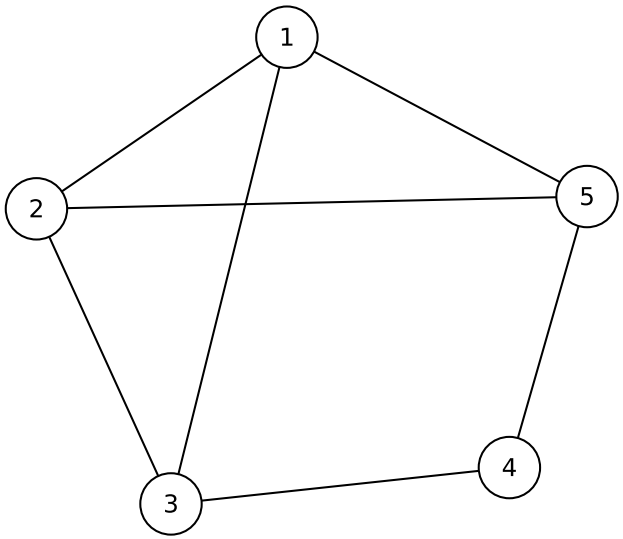
Some examples of graph invariants include:

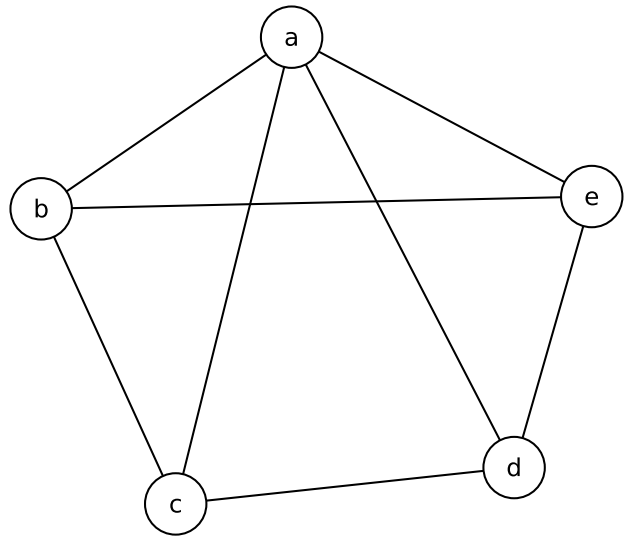
1. number of vertices
2. number of edges
3. number of spanning trees
4. degree sequence
5. spectrum

Example (1/2)

Two graphs with different graph invariants **cannot** be isomorphic

The following two graphs have different number of edges, hence we can say with confidence that they are not isomorphic



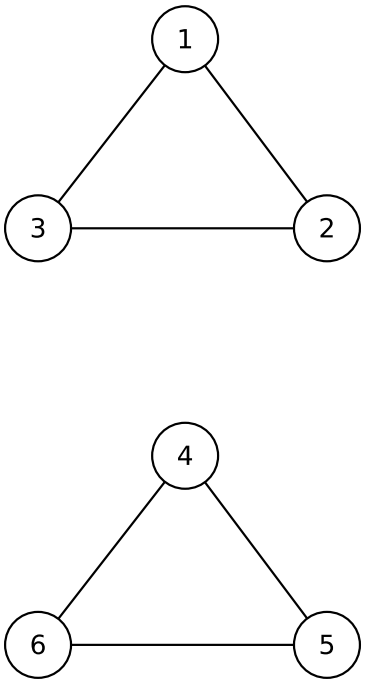


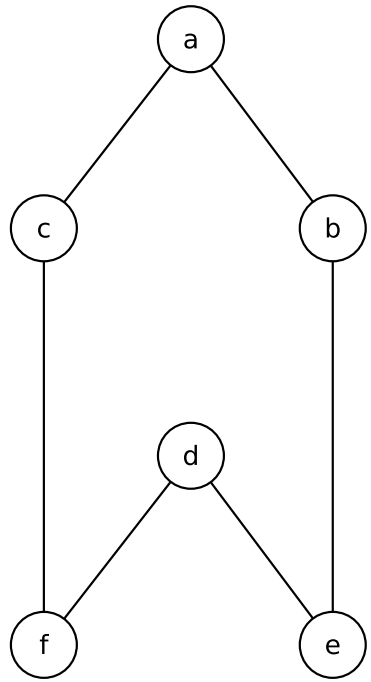
$G_1 \not\cong G_2$

Example (2/2)

Two graphs with the same invariants may or may not be isomorphic

The following two graphs have the same degree sequence, but are **not** isomorphic





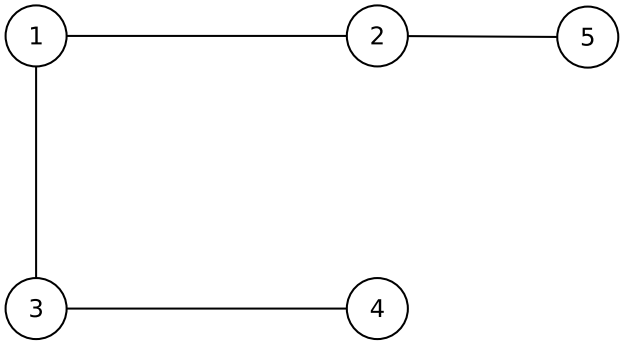
$G_1 \not\cong G_2$

Definition (Subgraph Isomorphism (SI))

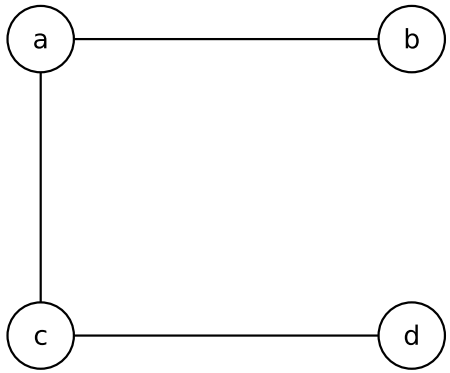
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. A subgraph isomorphism from G_1 to G_2 is a function $f : V_1 \rightarrow V_2$ such that if $(v_i, v_j) \in E_1$, then $(f(v_1), f(v_2)) \in E_2$, and if $(v_i, v_j) \notin E_1$, then $(f(v_1), g(v_2)) \notin E_2$

- The Subgraph Isomorphism problem asks if a graph contains a subgraph that is topologically identical (isomorphic) to a second graph
- Subgraph isomorphism generalizes problems such as Clique and Hamiltonian Path, and is therefore NP-complete [GJ79]
  - Exponential runtime in worst case
  - Not practical for larger graphs with many nodes

Example



G_1



G_2

The subgraph of G_1 induced by vertices $\{1, 2, 3, 4\}$ is isomorphic to G_2 since

- $f(1) = a$
- $f(2) = b$
- $f(3) = c$
- $f(4) = d$

is a one-to-one correspondence between the vertices of the induced subgraph and G_2

Definition (Maximum Common Subgraph (MCS))

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. A common subgraph G of G_1 and G_2 is a graph such that there exist subgraph isomorphisms from G to G_1 and from G to G_2. Graph G is a maximum common subgraph of G_1 and G_2 if there exists no other common subgraph of of G_1 and G_2 that has more nodes than G

- Maximum Common Subgraph finds the largest subgraph of a graph that is isomorphic to a subgraph of a second graph
- It is known that the Maximum Common Subgraph problem is NP-complete [GJ79]
  - Same problems as in the case of Subgraph Isomorphism

Graph Edit Distance [GXTL10]

The graph edit distance is a dissimilarity measure between graphs:

- takes as input two graphs
- outputs their distance

Computes the dissimilarity between two graphs by taking into account the *number* and the *strength* of the distortions that are needed to transform one graph to the other

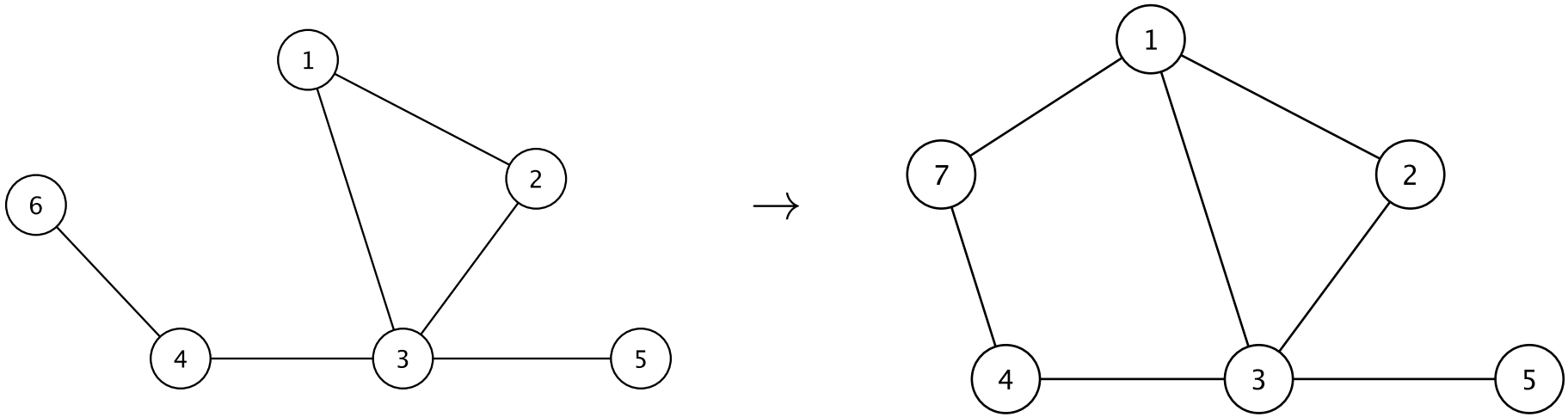
Given two graphs, G_1 and G_2, a graph edit distance method transforms G_1 into G_2 using some edit operations

- insertions
- deletions
- substitutions

of both nodes and edges

It then computes their dissimilarity based on these operations

Example



G_1 G_2

One of the edit operation sequences includes

- insertion and edge insertion (vertex 7 and its relative edges)
- node deletion and edge deletion (vertex 6 and its relative edge)

A cost function is defined for each operation

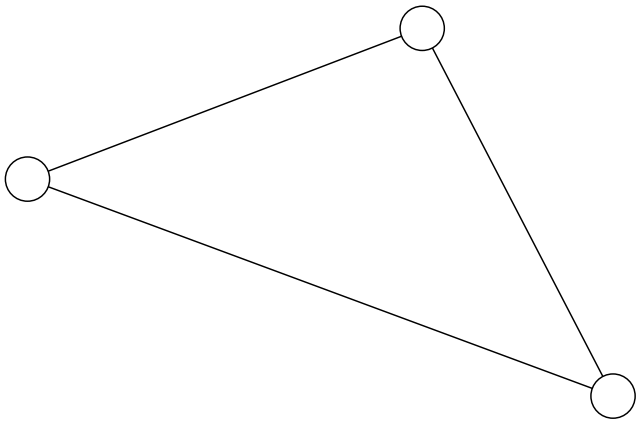
Total cost of sequence $\rightarrow$ sum of costs for all operations in the sequence

- Let $G = (V, E)$ be a simple unweighted, undirected graph where V is the set of vertices and E the set of edges
- The neighbourhood $\mathcal{N}(v)$ of vertex v is the set of all vertices adjacent to v, $\mathcal{N}(v) = \{u : (v, u) \in E\}$ where (v, u) is an edge between v and u
- A labeled graph is a graph with labels on vertices and/or edges
- Given a set of labels $\mathcal{L}$, $\ell : V \to \mathcal{L}$ is a function that assigns labels to the vertices of the graph
- A walk in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_{k+1}$ where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k$
- A walk in which $v_i \neq v_j \Leftrightarrow i \neq j$ is called a path

- A cycle is a path with $(v_{k+1}, v_1) \in E$
- An acyclic graph is a graph having no cycles
- A subtree is an acyclic subgraph in which there is a path between any two vertices
- The height of a subtree is the maximum distance between the root and any other node in the graph

Learning on Graphs

- To analyze and extract knowledge from graphs, one needs to perform machine learning tasks
- Most machine learning algorithms require the input to be represented as a fixed-length feature vector
- There is no straightforward way to transform graphs to such a representation



?

Definition (Kernel Function)

The function $k : \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ is a kernel if it is:

1. symetric: $k(x, y) = k(y, x)$
2. positive semi-definite: $\forall x_1, x_2, \ldots, x_n \in \mathcal{X}$, the Gram Matrix $\mathbf{K}$ defined by $\mathbf{K}_{ij} = k(x_i, x_j)$ is positive semi-definite

- If a function satisfies the above two conditions on a set $\mathcal{X}$, it is known that there exists a map $\phi : \mathcal{X} \to \mathbb{H}$ into a Hilbert space $\mathbb{H}$, such that:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

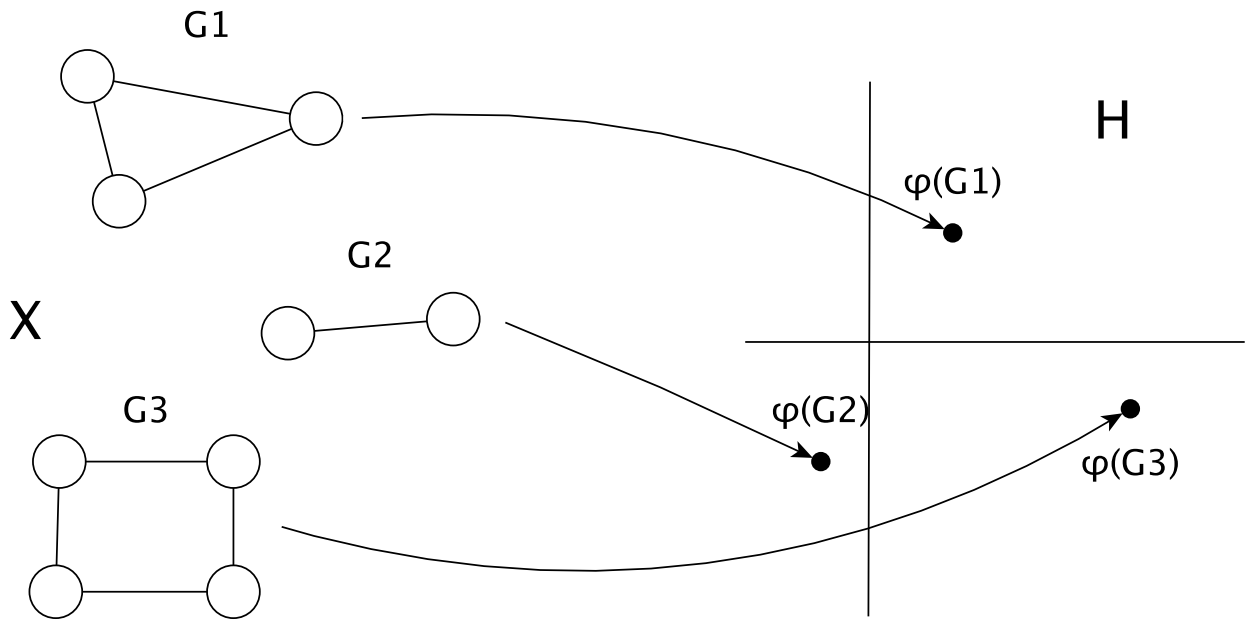
for all $(x, y) \in \mathcal{X}^2$ where $\langle \cdot, \cdot \rangle$ is the inner product in $\mathbb{H}$

- Informally, $k(x, y)$ is a measure of similarity between x and y

Definition (Graph Kernel)

A graph kernel $k : \mathcal{G} \times \mathcal{G} \rightarrow \mathcal{R}$ is a kernel function over a set of graphs $\mathcal{G}$

- Instance of R-convolution kernels [HAU99]
- It is equivalent to an inner product of the embeddings $\phi : \mathcal{X} \rightarrow \mathbb{H}$ of a pair of graphs into a Hilbert space
- Makes the whole family of kernel methods applicable to graphs.



- Many machine learning algorithms can be expressed only in terms of inner products between vectors
- Let $\phi(G_1), \phi(G_2)$ be vector representations of graphs G_1, G_2 in a very high (possibly infinite) dimensional feature space
- Computing the explicit mappings $\phi(G_1), \phi(G_2)$ and their inner product $\langle \phi(x), \phi(y) \rangle$ for the pair of graphs can be computationally demanding
- The kernel trick avoids the explicit mapping by directly computing the inner product $\langle \phi(x), \phi(y) \rangle$ via the kernel function

Let $\mathcal{X} = \mathcal{R}^2$ and $x = (x_1, x_2), y = (y_1, y_2) \in \mathcal{X}$

For any $x = (x_1, x_2)$ let ϕ be a map $\phi : \mathcal{R}^2 \to \mathcal{R}^3$ defined as:

$$\phi(x) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

Let also $k : \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ a kernel defined as $k(x, y) = \langle x, y \rangle^2$. Then

$$\begin{aligned} k(x, y) &= \langle x, y \rangle^2 = (x_1y_1 + x_2y_2)^2 = x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 \\ &= \langle \phi(x), \phi(y) \rangle \end{aligned}$$

Hence, using the kernel we can compute the inner product $\langle \phi(x), \phi(y) \rangle$ without computing $\phi(x)$ and $\phi(y)$

Definition (Complete Graph Kernel)

A graph kernel $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$ is complete if ϕ is injective

Hence, for complete graph kernels, $\phi(G_1) = \phi(G_2)$ iff G_1 and G_2 are isomorphic

How **hard** is to compute a complete graph kernel?

Proposition ([GFW03])

Computing any complete graph kernel is at least as hard as the graph isomorphism problem

Definition (Complete Graph Kernel)

A graph kernel $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$ is complete if ϕ is injective

Hence, for complete graph kernels, $\phi(G_1) = \phi(G_2)$ iff G_1 and G_2 are isomorphic

How **hard** is to compute a complete graph kernel?

Proposition ([GFW03])

Computing any complete graph kernel is at least as hard as the graph isomorphism problem

We are interested in kernels that can be computed in polynomial time (with small degree)

If the kernel is complete:

- Computation is at least as hard as the graph isomorphism problem
  - No polynomial algorithm for the graph isomorphism problem is known

If the kernel is not complete:

- It can be computed efficiently
- We can have $\phi(G_1) = \phi(G_2)$ even if $G_1 \ncong G_2$
  - The kernel is not expressive enough

Subgraph Kernel

Let $\mathcal{G}$ denote the set of all graphs

For any graph $G \in \mathcal{G}$, each feature of its subgraph space is defined as:

$$\forall H \in \mathcal{G}, \quad \phi_H(G) = |\{G' \text{ is a subgraph of } G : G' \cong H\}|$$

The subgraph kernel between two graphs G_1 and G_2 is defined as:

$$k(G_1, G_2) = \sum_{H \in \mathcal{G}} \lambda_H \phi_H(G_1) \phi_H(G_2)$$

where λ_H is a positive weight

How **hard** is to compute the subgraph kernel?

The subgraph kernel is a **complete** graph kernel

- it is at least as hard as solving the graph isomorphism problem

Subgraph Kernel

Let $\mathcal{G}$ denote the set of all graphs

For any graph $G \in \mathcal{G}$, each feature of its subgraph space is defined as:

$$\forall H \in \mathcal{G}, \quad \phi_H(G) = |\{G' \text{ is a subgraph of } G : G' \cong H\}|$$

The subgraph kernel between two graphs G_1 and G_2 is defined as:

$$k(G_1, G_2) = \sum_{H \in \mathcal{G}} \lambda_H \phi_H(G_1) \phi_H(G_2)$$

where λ_H is a positive weight

How **hard** is to compute the subgraph kernel?

Proposition ([GFW03])

Computing the subgraph kernel is NP-hard

A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks
- shortest paths
- cyclic patterns
- subtree patterns
- graphlets

$$\vdots$$

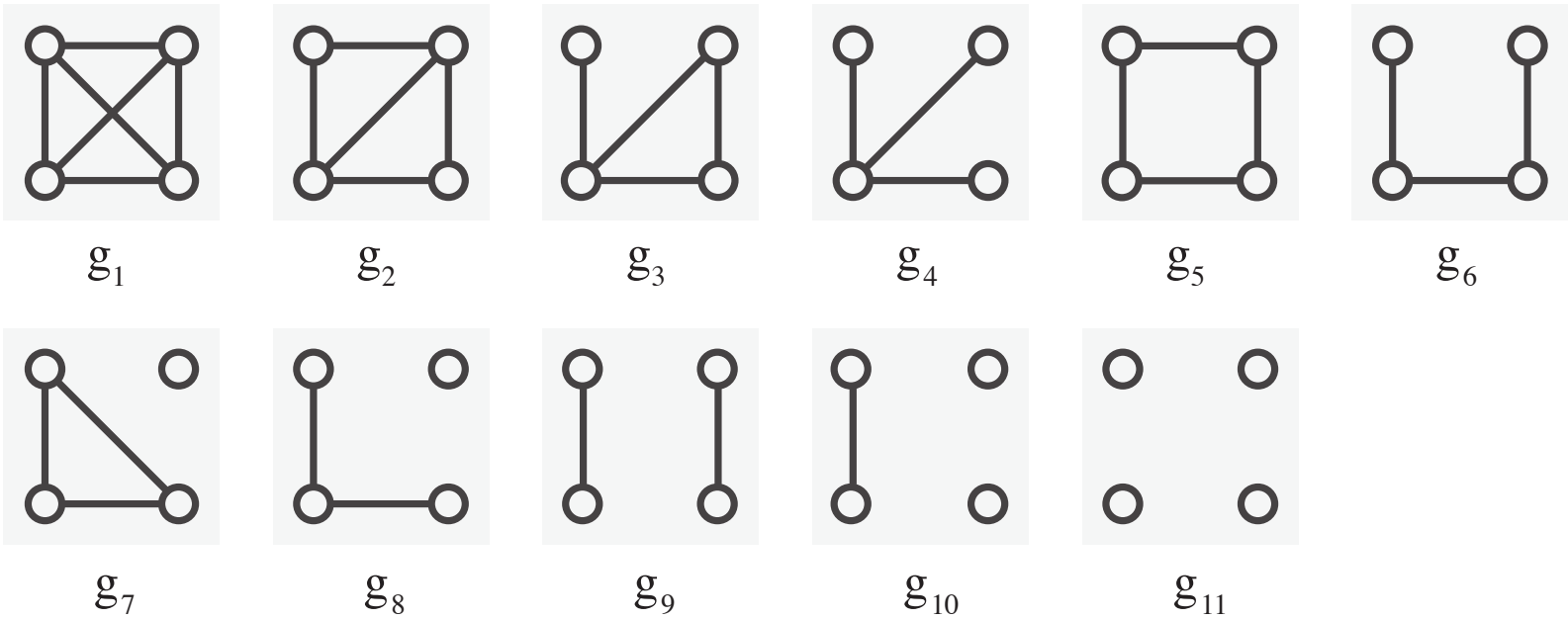
Graphlet Kernel [SPM+09]

The graphlet kernel compares graphs by counting *graphlets*

A graphlet corresponds to a small subgraph [Prž07]

- typically of 3,4 or 5 vertices

Below is the set of graphlets of size 4



Let $\mathcal{G} = \{graphlet_1, graphlet_2, \ldots, graphlet_r\}$ be the set of size-k graphlets

Let also $f_G \in \mathcal{N}^r$ be a vector such that its i-th entry is $f_{G,i} = \#(graphlet_i \sqsubseteq G)$

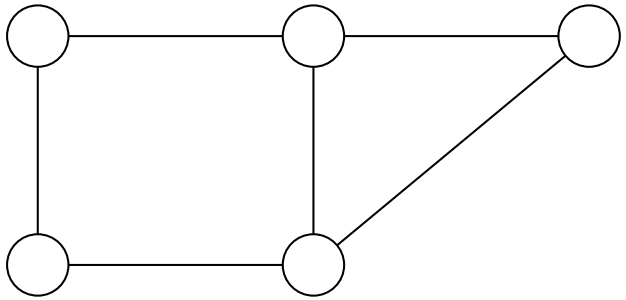
The graphlet kernel is defined as:

$$k(G_1, G_2) = f_{G_1}^T f_{G_2}$$

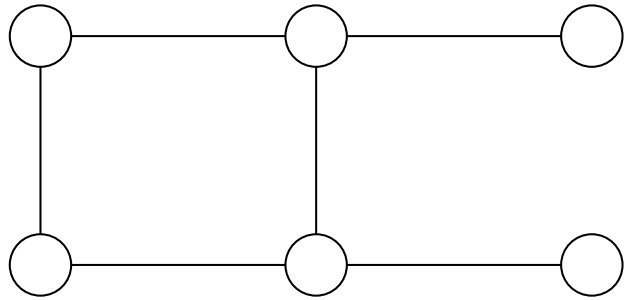
Problems:

- There are $\binom{n}{k}$ size-k subgraphs in a graph
- Exaustive enumeration of graphlets is very expensive

  Requires $O(n^k)$ time
- For labeled graphs, the number of graphlets increases further



G_1



G_2

The vector representations of the graphs above according to the set of graphlets of size 4 is:

$$f_{G_1} = (0, 0, 2, 0, 1, 2, 0, 0, 0, 0, 0)^T$$
$$f_{G_2} = (0, 0, 0, 2, 1, 5, 0, 4, 0, 3, 0)^T$$

Hence, the value of the kernel is:

$$k(G_1, G_2) = f_{G_1}^T f_{G_2} = 11 \quad (1)$$

Subtree Kernel [RG03]

Compares subtree patterns in two graphs

A subtree pattern is a subgraph of a graph which has

- a root vertex
- no cycles

The height of a subtree is the maximum distance between the root and any other node in the subtree

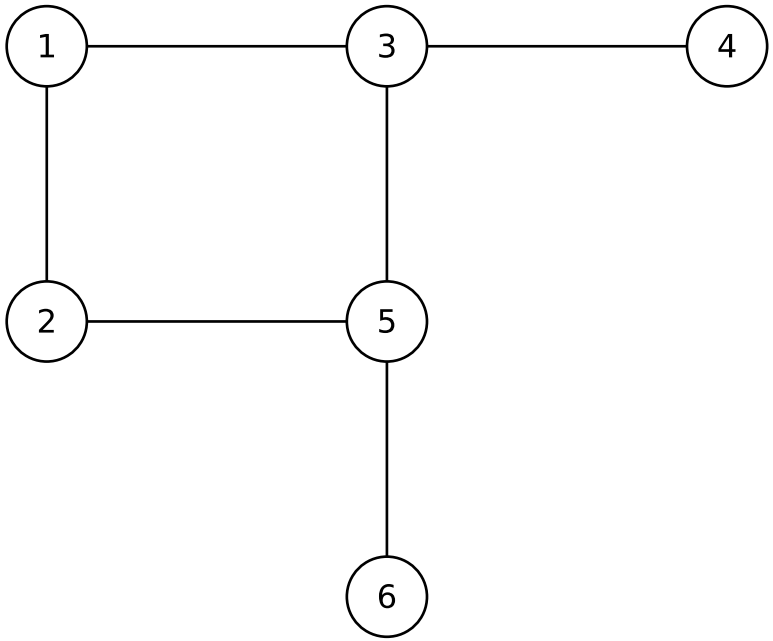
If there are cycles in the graph, a vertex can appear more than once in a subtree pattern

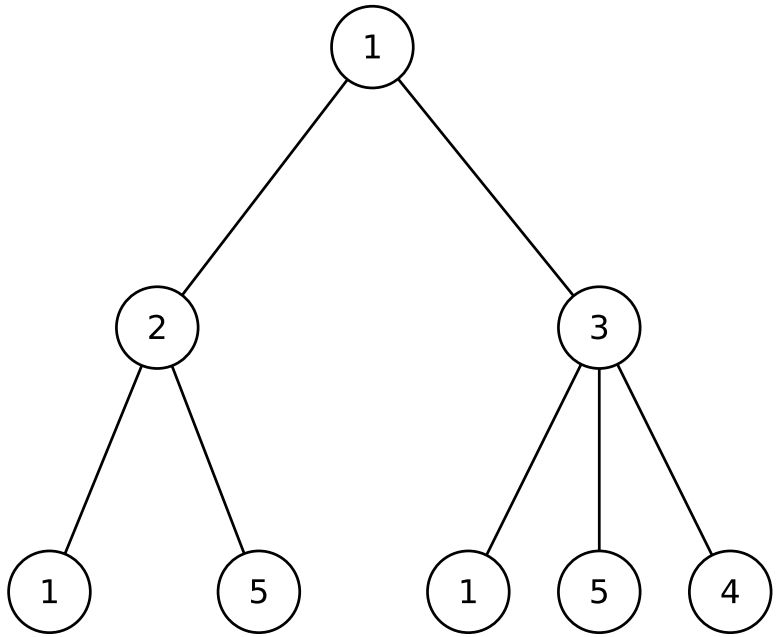
- it is treated as a distinct vertex such that the pattern is still a cycle-free tree

For all pairs of nodes v from G_1 and u from G_2:

- Create the subtree patterns of height h rooted at v, u
- Compare v and u via a kernel function
- Recursively compare all vertices of the subtree patterns of v and u via a kernel function

Example





Subtree of height 2
rooted at vertex 1

Given a pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the subtree kernel of height h is defined as:

$$k(G_1, G_2) = \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_h(v_1, v_2)$$

where

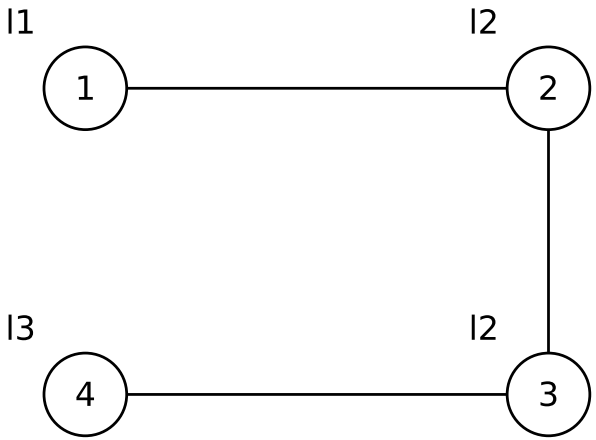
$$k_h(v_1, v_2) = \begin{cases} \delta(\ell(v_1) = \ell(v_2)) & \text{if } h = 0 \\ \lambda_{v_1} \lambda_{v_2} \delta(\ell(v_1) = \ell(v_2)) \sum_{R \in \mathcal{M}(v_1, v_2)} \prod_{(w_1, w_2) \in R} k_{h-1}(w_1, w_2) & \text{if } h > 0 \end{cases}$$

where $\delta(\cdot, \cdot)$ is the Kronecker delta function that equals 1 if its arguments are equal, 0 otherwise, λ_{v_1} and λ_{v_2} are weights associated with nodes v_1 and v_2, and

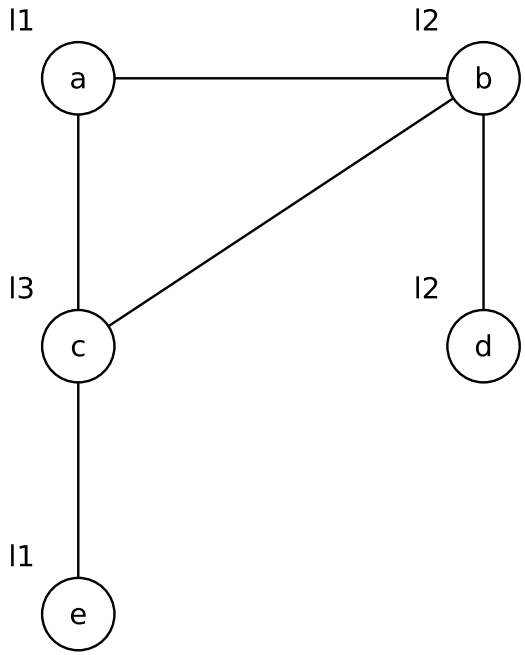
$$\mathcal{M}(v_1, v_2) = \Big\{ R \subseteq \mathcal{N}(v_1) \times \mathcal{N}(v_2) | (\forall (u_1, u_2), (w_1, w_2) \in R : u_1 = w_1 \Leftrightarrow u_2 = w_2) \wedge (\forall (u_1, u_2) \in R : \ell(u_1) = \ell(u_2)) \Big\}$$

Example

We are given the following graphs



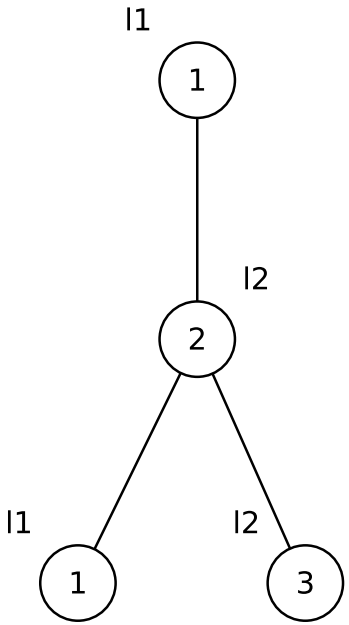
G_1

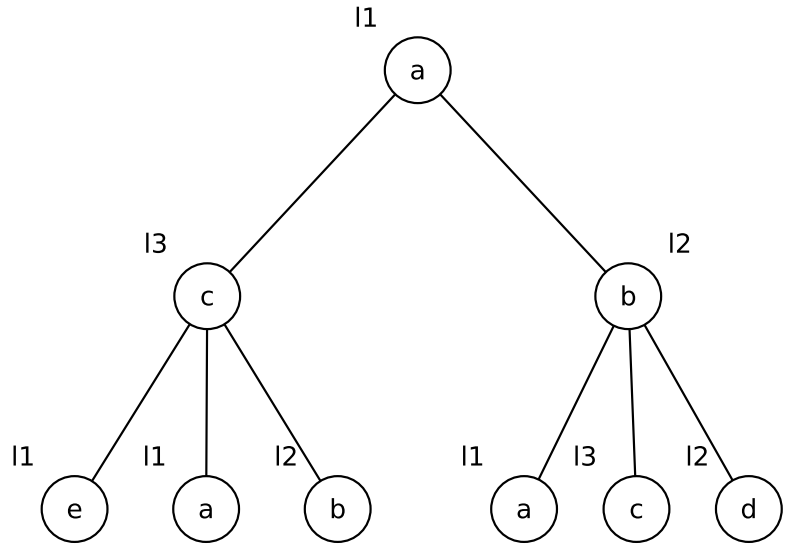


G_2

Example

Below are given the subtrees of G_1 and G_2 with height 2 rooted at 1 and a respectively





We are gonna compute $k_2(1, a)$

We set $\lambda_v = 1$ for all v and we have:

$$k_2(1,a) = \delta(\ell(1) = \ell(a)) \sum_{R \in \mathcal{M}(1,a)} \prod_{(v_1,v_2) \in R} k_1(v_1, v_2)$$

- $\delta(\ell(1) = \ell(a)) = 1$ since $\ell(1) = \ell(a) = l1$
- $\mathcal{M}(1,a) = \{(2,b)\}$ since $\ell(2) = \ell(b) = l2$

Hence, we have now to compute $k_1(2,b)$

$$k_1(2,b) = \delta(\ell(2) = \ell(b)) \sum_{R \in \mathcal{M}(2,b)} \prod_{(v_1,v_2) \in R} k_0(v_1, v_2)$$

- $\delta(\ell(2) = \ell(b)) = 1$ since $\ell(2) = \ell(b) = l2$
- $\mathcal{M}(2,b) = \{(1,a),(3,d)\}$ since $\ell(1) = \ell(a) = l1$ and $\ell(3) = \ell(d) = l2$

At height 0, we have:

$$k_0(1, a) = k_0(3, d) = 1$$

Therefore,

$$k_1(2, b) = k_0(1, a)k_0(3, d) = 1$$

And finally,

$$k_2(1, a) = k_1(2, b) = 1$$

Subtree kernel

Pros: Richer representation of graph structure

Cons: Very high complexity

- $\mathcal{O}(n^2 h 4^d)$ where d is the maximum degree of the pair of graphs

Compares the length of shortest-paths of two graphs

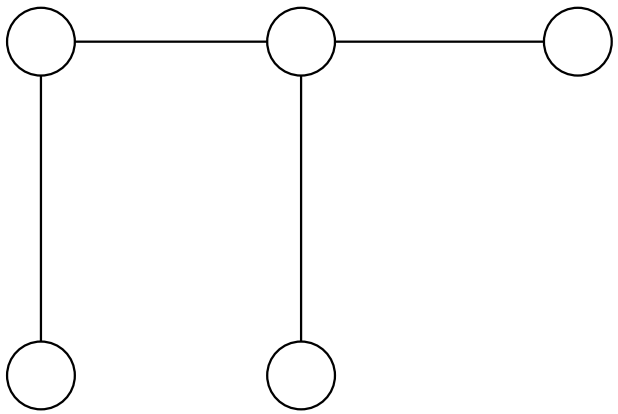
- and their endpoints in labeled graphs

Floyd-transformation

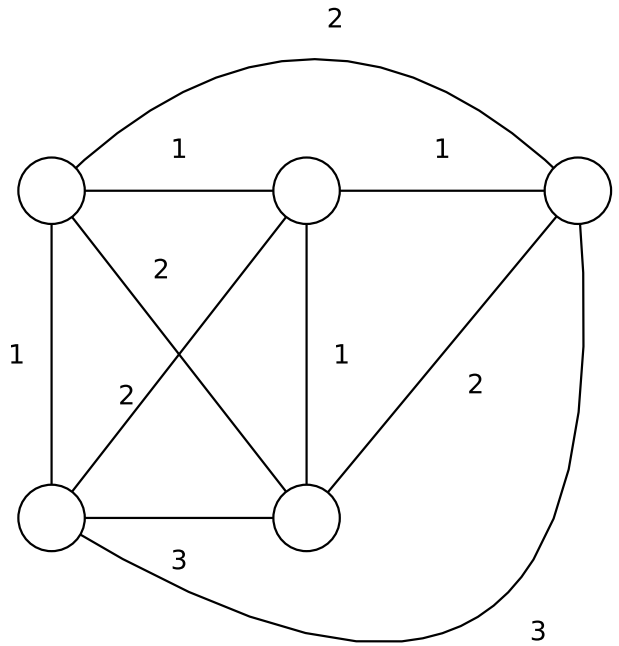
Transforms the original graphs into shortest-paths graphs

- Compute the shortest-paths between all pairs of vertices of the input graph G using some algorithm (i. e. Floyd-Warshall)
- Create a shortest-path graph S which contains the same set of nodes as the input graph G
- All nodes which are connected by a walk in G are linked with an edge in S
- Each edge in S is labeled by the shortest distance between its endpoints in G

Floyd-transformation



G



S

Given the Floyd-transformed graphs $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ of G_1 and G_2, the shortest path kernel is defined as:

$$k(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2)$$

where $k^{(1)}_{walk}$ is a kernel on edge walks of length 1

- For unlabeled graphs, it can be:

$$k^{(1)}_{walk}(e_1, e_2) = \delta(\ell(e_1), \ell(e_2)) = \begin{cases} 1 & \text{if } \ell(e_1) = \ell(e_2), \\ 0 & \text{otherwise} \end{cases}$$

  where $\ell(e)$ gives the label of edge e

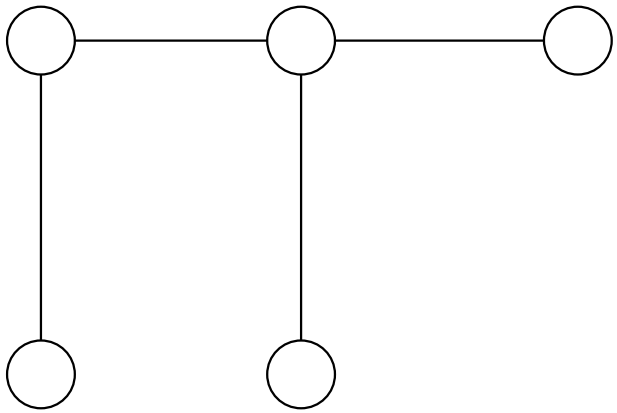
- For labeled graphs, it can be:

$$k^{(1)}_{walk}(e_1, e_2) = \begin{cases} 1 & \text{if } \ell(e_1) = \ell(e_2) \wedge \ell(e_1^1) = \ell(e_2^1) \wedge \ell(e_1^2) = \ell(e_2^2), \\ 0 & \text{otherwise} \end{cases}$$

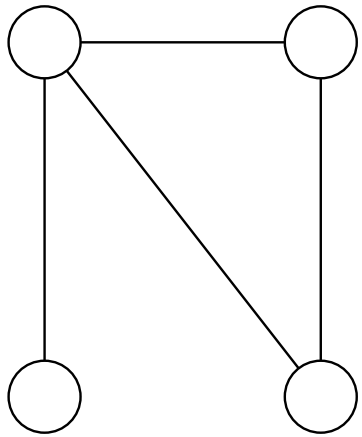
  where e^1, e^2 are the two endpoints of e

Example

Compute the shortest path kernel for the following pair of unlabeled graphs

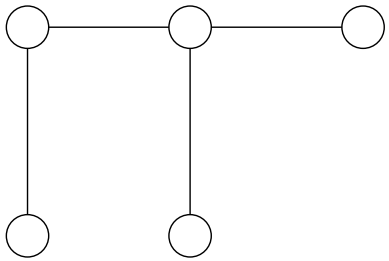


G_1

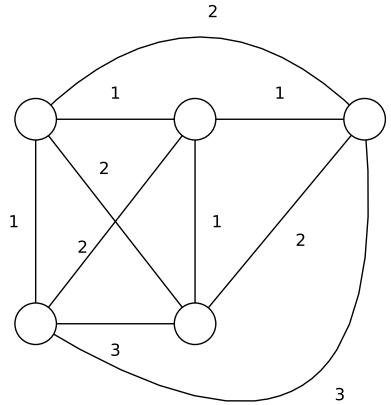


G_2

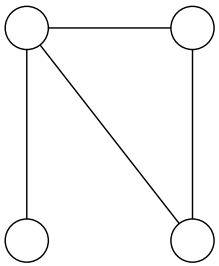
Floyd-transformations



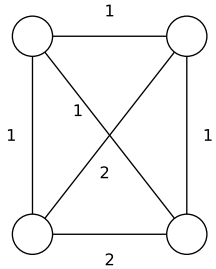
$\rightarrow$



G_1 S_1



$\rightarrow$



G_2 S_2

In S_1 we have:

- 4 edges with label 1
- 4 edges with label 2
- 2 edges with label 3

In S_2 we have:

- 4 edges with label 1
- 2 edges with label 2

Hence, the value of the kernel is:

$$k(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2) = 4 * 4 + 4 * 2 = 24$$

Computing the shortest path kernel includes:

- Computing shortest paths for all pairs of vertices in the two graphs: $\mathcal{O}(n^3)$
- Comparing all pairs of shortest paths from the two graphs: $\mathcal{O}(n^4)$

Hence, runtime is $\mathcal{O}(n^4)$

Problems:

- Very high complexity for large graphs
- Shortest-path graphs may lead to memory problems on large graphs

Cyclic Pattern Kernel [HGW04]

Compares simple cycles and tree patterns in two graphs

- extracts the set of all simple cycles from the two graphs

Problems:

- Number of simple cycles is exponential in the number of vertices n in the worst case
- Computing the Cyclic pattern kernel on general graphs is NP-hard

Solution:

- Consider graphs whose number of simple cycles is bounded by a constant (graphs with up to k simple cycles)

However, can only be applied to the families of graphs where the number of simple cycles is polynomially bounded

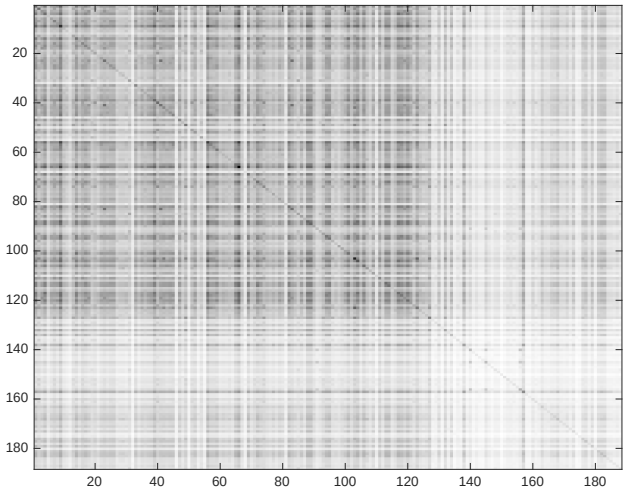
- its practical use is limited

A Structural Smoothing Framework [YV15]

Diagonal dominance problem of kernels that compare specific substructures of graphs:

1. Very large feature space, hence, unlikely that two graphs will contain similar substructures
2. Kernel value between pairs of graphs $\ll$ kernel value between a graph and itself

This leads to the diagonal dominance problem



The resulting kernel matrix is close to the identity matrix

However, the substructures used to define a graph kernel are often **related** to each other

Solution: apply smoothing to alleviate the problem

First construct a Directed Acyclic Graph (DAG):

- each vertex corresponds to a substructure
- for each substructure s of size k determine all possible substructures of size $k-1$ that s can be reduced into
- these correspond to the parents of s
- draw a weighted directed edge from each parent to its children vertices

DAG provides a topological ordering of the vertices

- all descendants of a given substructure at depth $k-1$ are at depth k

The structural smoothing for a substructure s at level k is defined as:

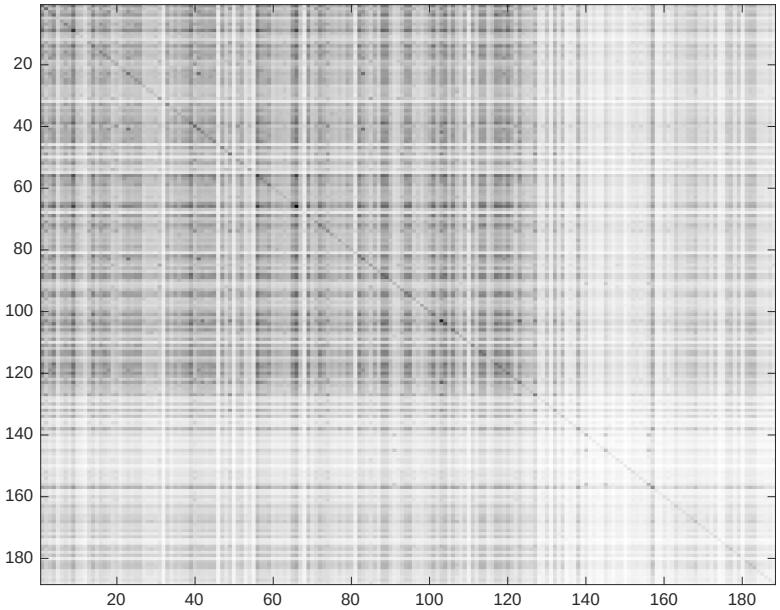
$$P_{SS}^{k}(s) = \frac{\max(c_s - d, 0)}{m} + \frac{m_d d}{m} \sum_{p \in \mathcal{P}_s} P_{SS}^{k-1}(p) \frac{w_{ps}}{\sum_{c \in \mathcal{C}_p} w_{pc}}$$

where

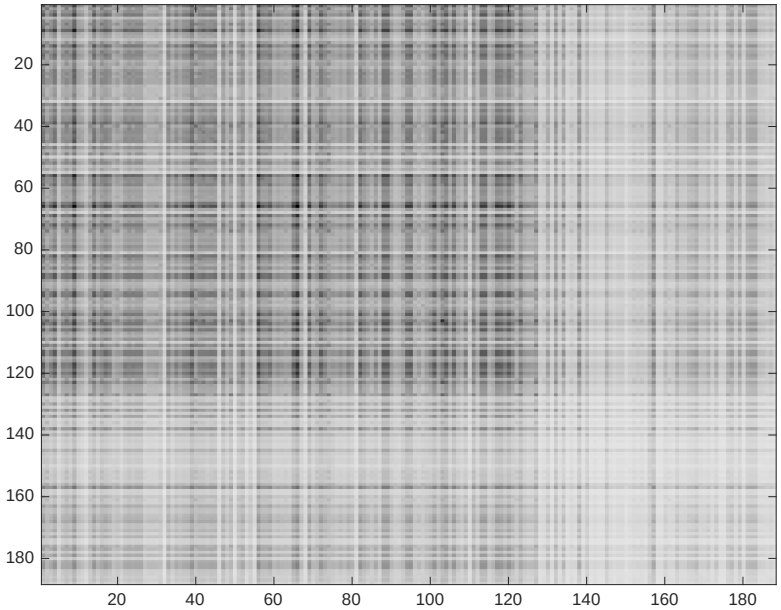
- c_a denotes the number of times substructure a appears in the graph
- $m = \sum_i c_i$ denotes the total number of substructures present in the graph
- $d > 0$ is a discount factor
- $m_d := |\{i : c_i > d\}|$ is the number of substructures whose counts are larger than d
- w_{ij} denotes the weight of the edge connecting vertex i to vertex j
- $\mathcal{P}_s$ denotes the parents of vertex s
- $\mathcal{C}_p$ the children of vertex p

Hence, even if the graph does not contain a substructure s ($c_s = 0$), its value in the feature vector can be greater than 0 ($P_{SS}(s) \geq 0$)

Example



Kernel matrix before smoothing



Kernel matrix after smoothing

- Probably the most well-studied family of graph kernels
- Counts matching walks in two graphs

Product graph

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their direct product $G_\times$ is a graph with vertex set:

$$V_\times = \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\} \text{ for unlabeled graphs}$$

or

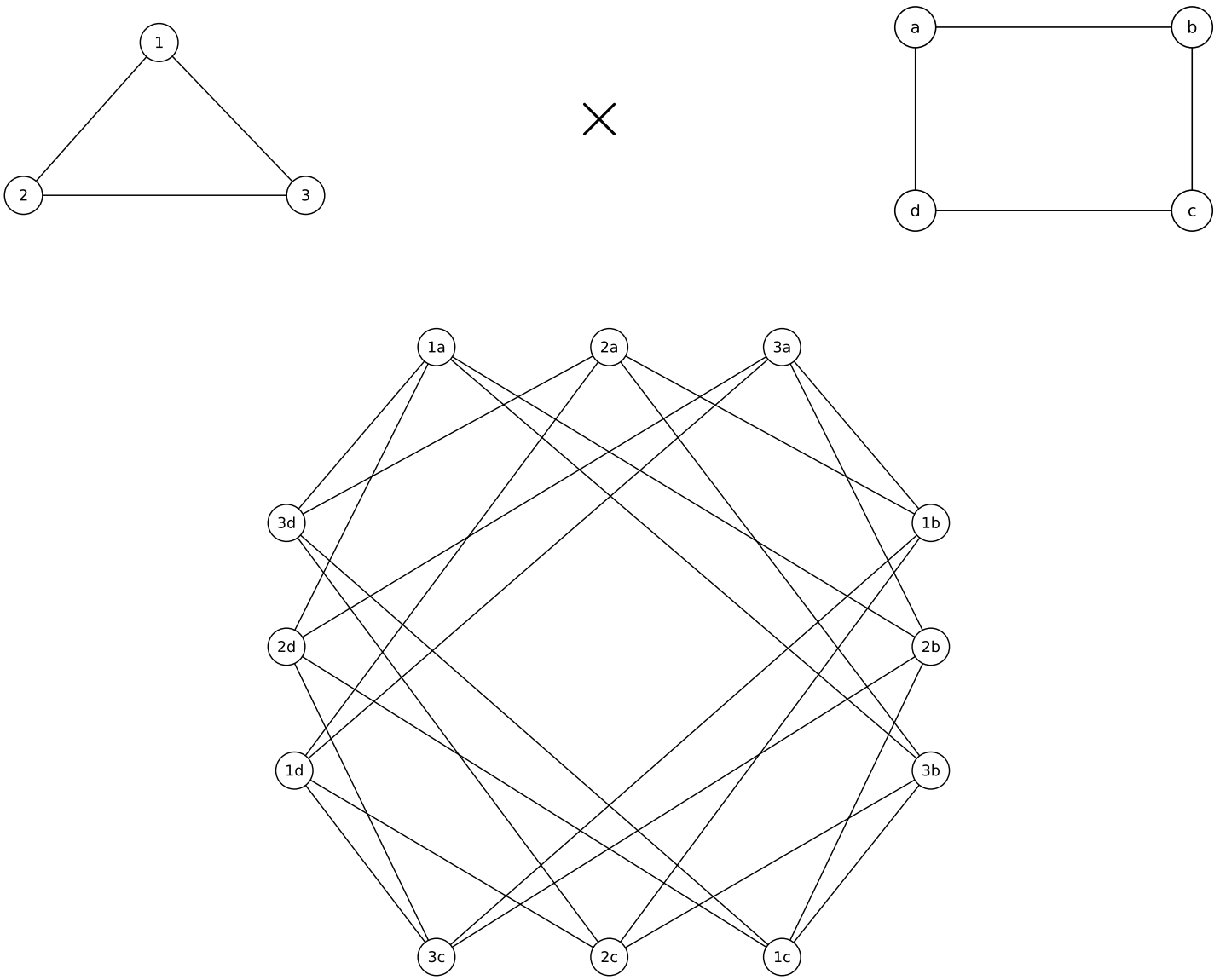
$$V_\times = \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2, \ell(v1) = \ell(v2)\} \text{ for labeled graphs}$$

and edge set:

$$E_\times = \{((v_1, v_2), (u_1, u_2)) : (v_1, u_1) \in E_1, (v_2, u_2) \in E_2\}$$

- vertices: pairs of vertices from G_1 and G_2
- draw edge if corresponding vertices of G_1 and G_2 are adjacent in G_1 and G_2

Example



The k-th power of the adjacency matrix A of a graph G computes walks of length k

- A^k_{ij} = number of walks of length k from vertex i to vertex j

Performing a random walk on $G_\times$ is equivalent to performing a simultaneous random walk on G_1 and G_2

- Common walks of length k can be computed using $A^k_\times$

For $k \in \mathcal{N}$, the k-step random walk kernel is defined as:

$$K^k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \Big[\sum_{l=0}^{k} \lambda_l A^l_\times \Big]_{ij}$$

where $\lambda_0, \lambda_1, \ldots, \lambda_k$ positive weights and $A^0_\times = \mathbf{I}$

For $k \to \infty$, we get the random walk kernel $K_{\times}^{\infty}(G_1, G_2)$

If $\lambda_l = \lambda^l$, $K_{\times}^{\infty}(G_1, G_2)$ can be directly computed as follows:

$$K_{\times}^{\infty}(G_1, G_2) = \sum_{i,j=1}^{|V_{\times}|} \left[\sum_{l=0}^{\infty} \lambda^l A_{\times}^l \right]_{ij} = \mathbf{e}^T (\mathbf{I} - \lambda A_{\times})^{-1} \mathbf{e}$$

where $\mathbf{e}$ the all-ones vector

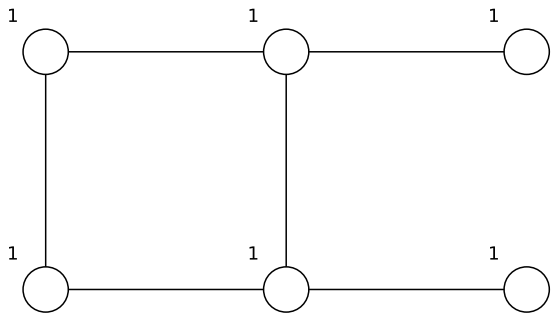
Problem: compitational complexity is $\mathcal{O}(n^6)$

Solution: Efficent computation using

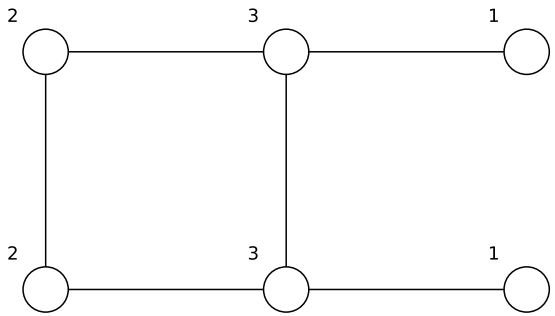
- Sylvester equation
- Conjugate gradient solver
- Fixed-point iterations
- Spectral decompositions

Label Enrichment: Morgan Index

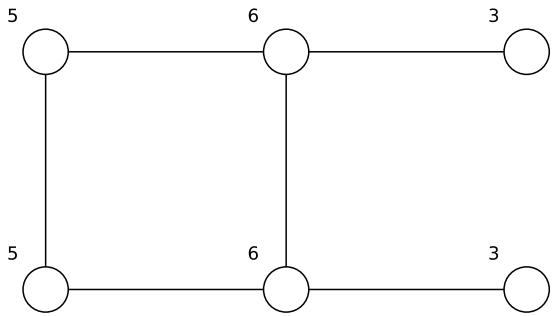
- Introduce new artificial node labels
- Initially all vertices are labeled with the number 1
- At each iteration, the label of a vertex is equal to the sum of the labels of its neighbors



No Morgan indices



Order 1 indices



Order 2 indices

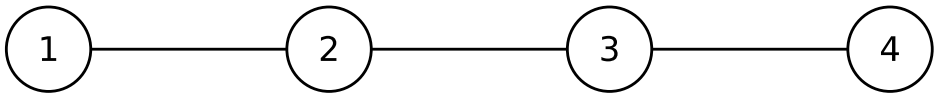
Label enrichment:

- number of labels ↑
- size of product graph ↓
- number of common label paths between graphs ↓
- computation time ↓

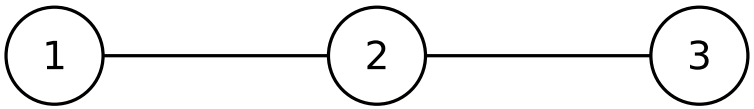
Random walk kernels suffer from *tottering*

A tottering walk is a walk $w = v_1 \dots v_n$ where $v_i = v_{i+2}$ for some i

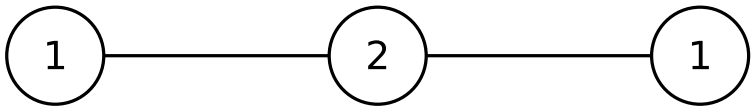
- after a move to a new vertex comes back to the original vertex
- results in redundant paths



Example graph



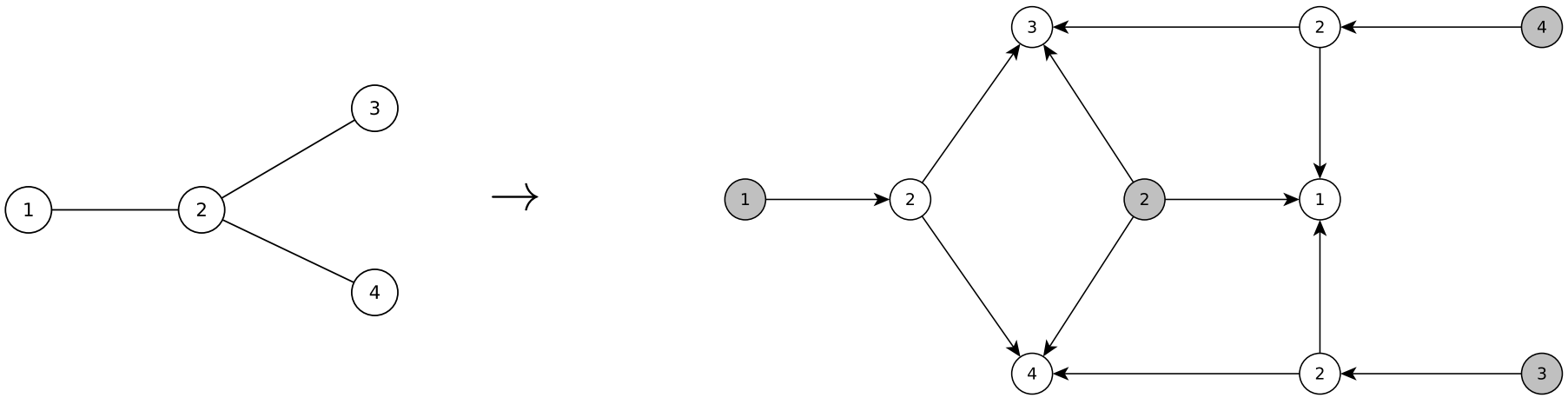
Non-tottering



Tottering

Second-order Markov random walk that forbids walks of the form $v \rightarrow u \rightarrow v$

Transform each graph G into a new directed graph G' and perform a normal random walk on G'



Original graph

Transformed graph

Weisfeiler-Lehman Framework [SSVL+11]

Uses the Weisfeiler-Lehman isomorphism test to improve the performance of existing kernels

- subtree kernel
- shortest path kernel

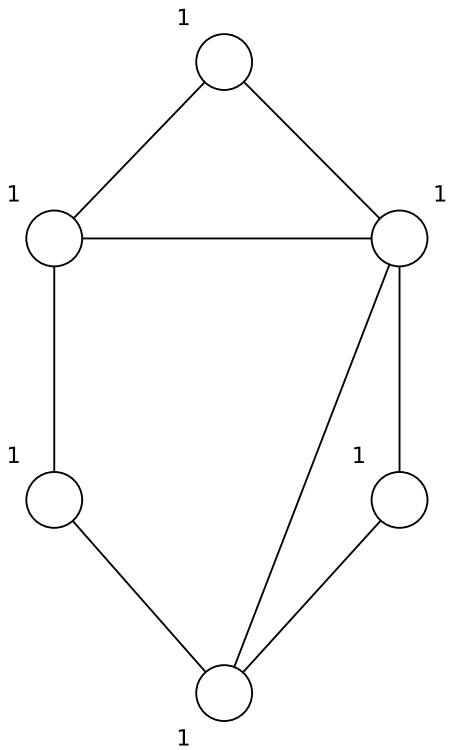
⋮

Weisfeiler-Lehman kernels achieve state-of-the-art results

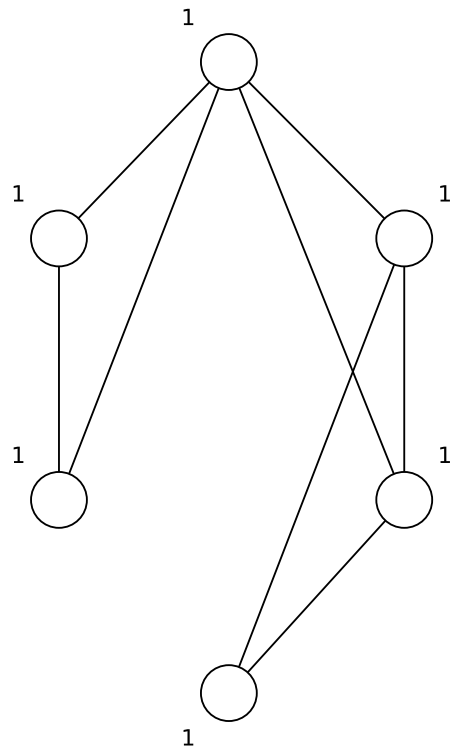
Based on the Weisfeiler-Lehman algorithm [WL68]: may answer if two graphs are not isomorphic

Example

Run the Weisfeiler-Lehman algorithm for the following pair of graphs

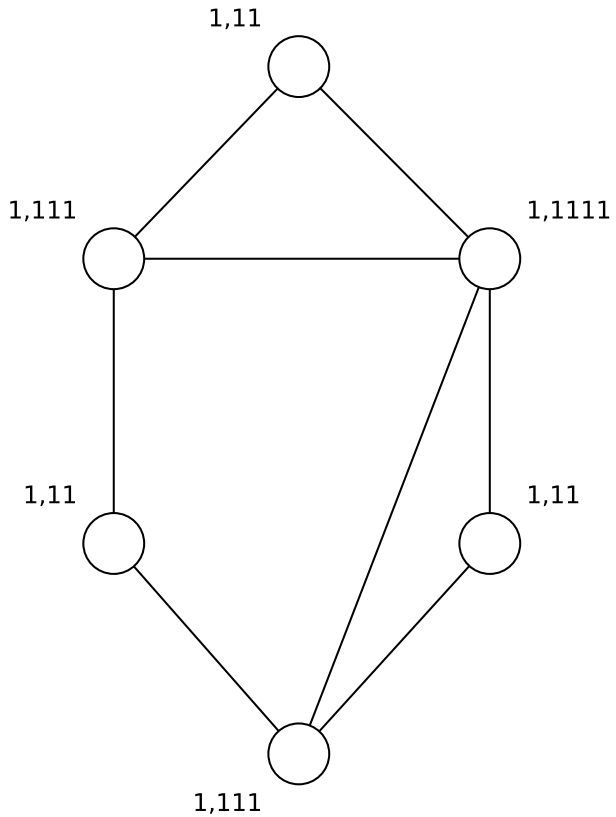


G_1

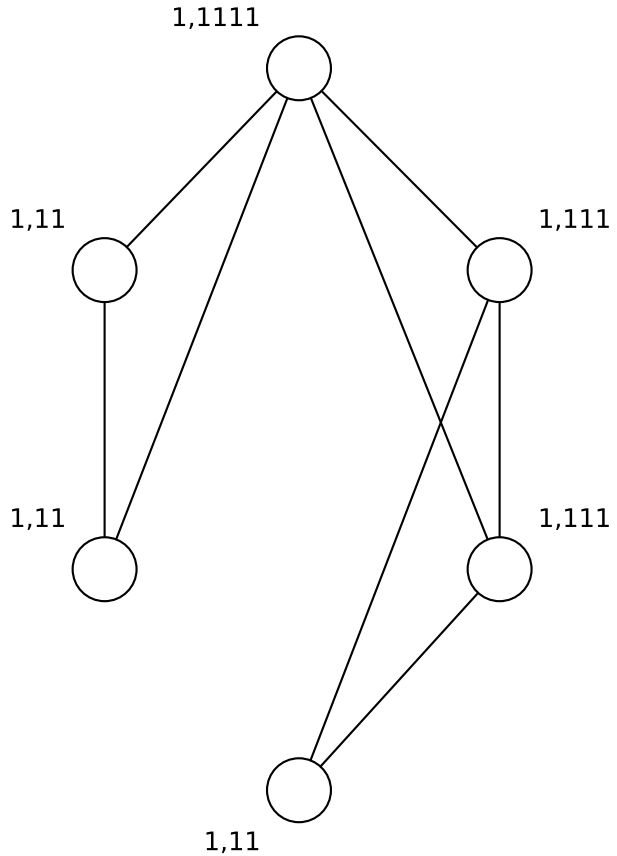


G_2

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices



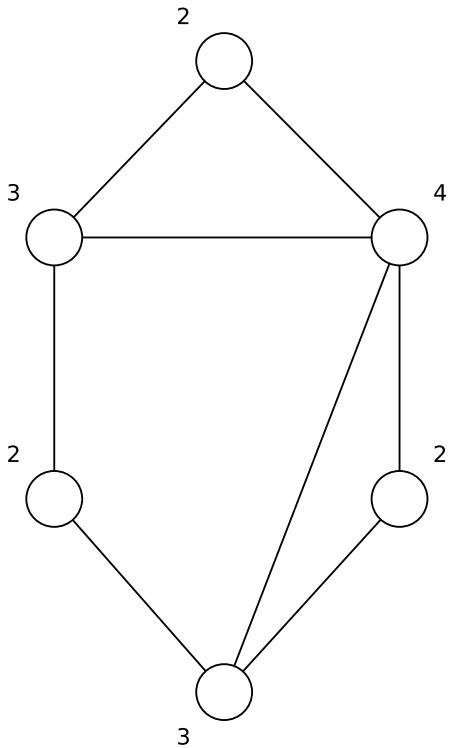
G_1



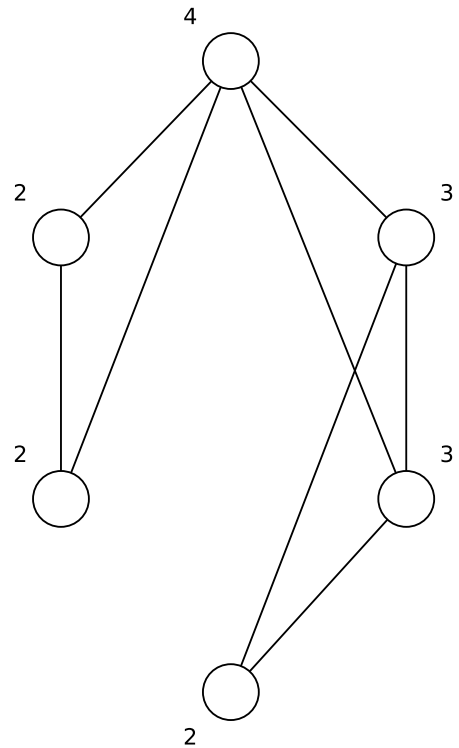
G_2

Second step: Compress the augmented labels into new, short labels:

- $1, 11 \rightarrow 2$
- $1, 111 \rightarrow 3$
- $1, 1111 \rightarrow 4$

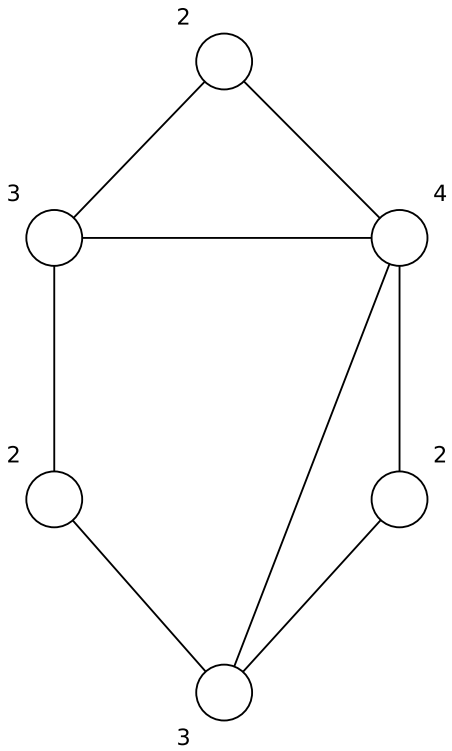


G_1

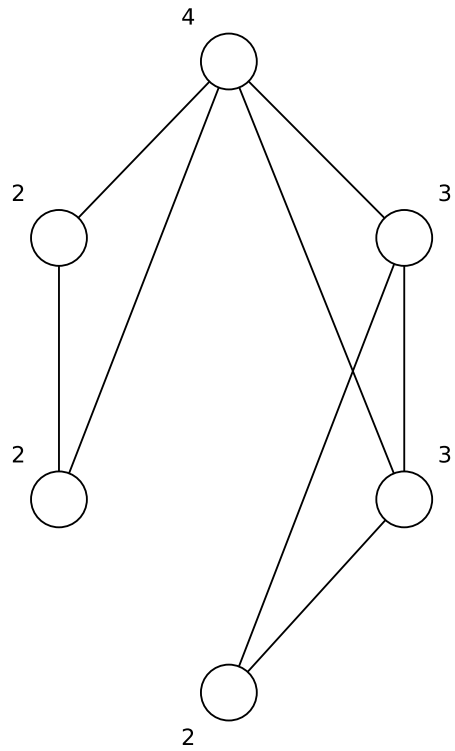


G_2

Are the label sets of G_1 and G_2 identical?



G_1

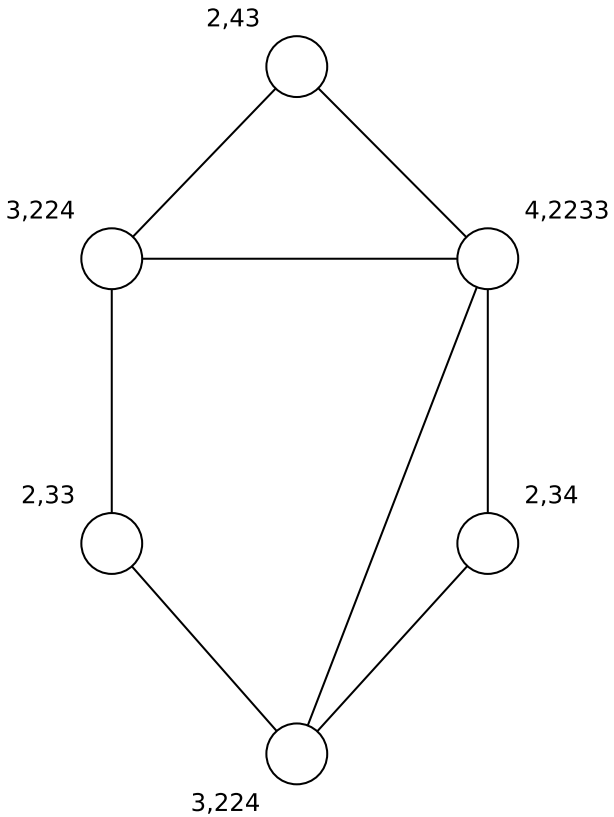


G_2

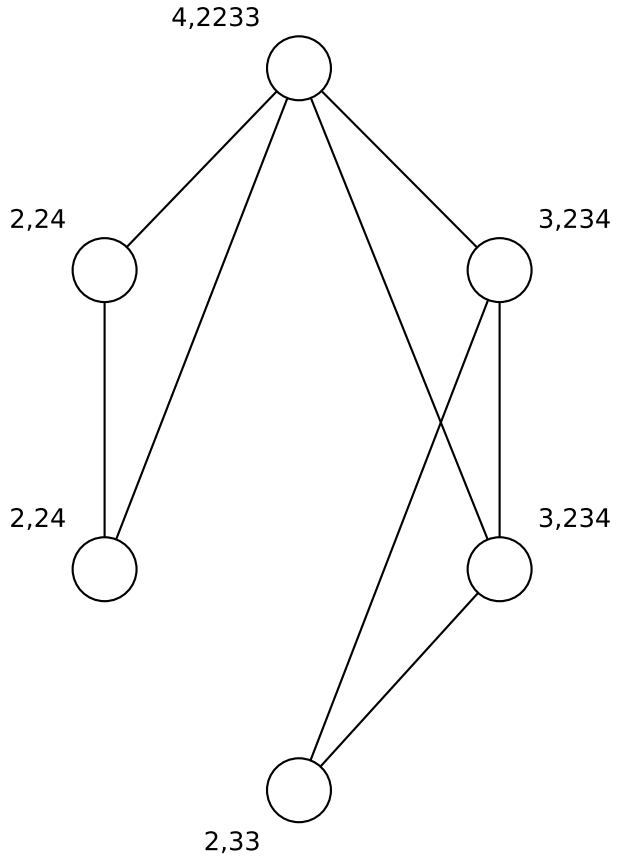
Yes!!!

Continue to the next iteration

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices



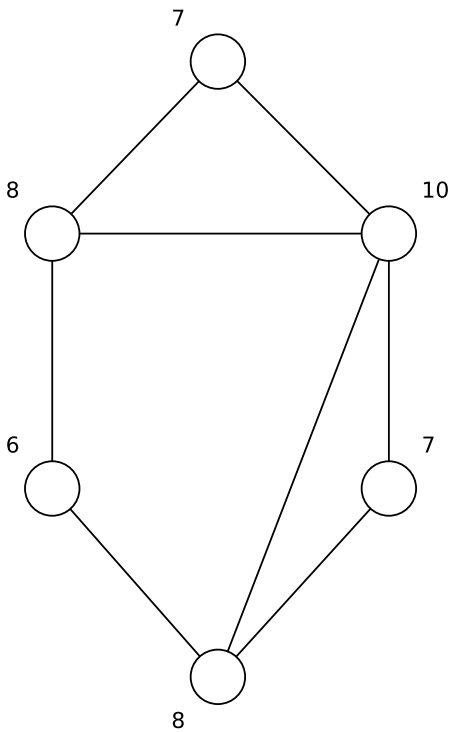
G_1



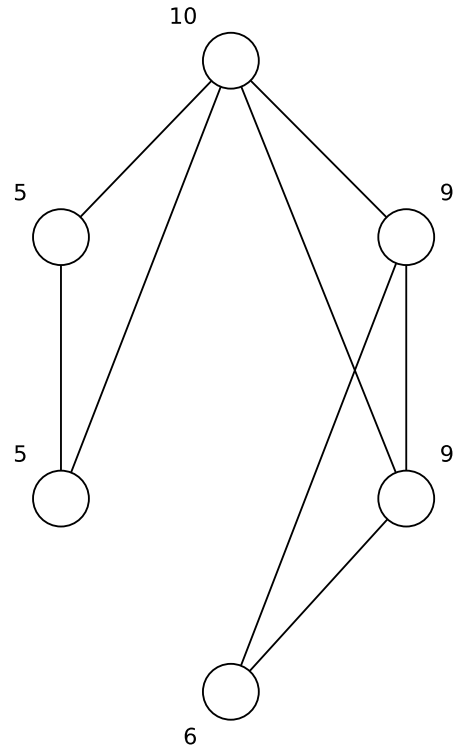
G_2

Second step: Compress the augmented labels into new, short labels:

- $2, 24 \rightarrow 5$
- $2, 33 \rightarrow 6$
- $2, 34 \rightarrow 7$
- $3, 234 \rightarrow 9$
- $4, 2233 \rightarrow 10$

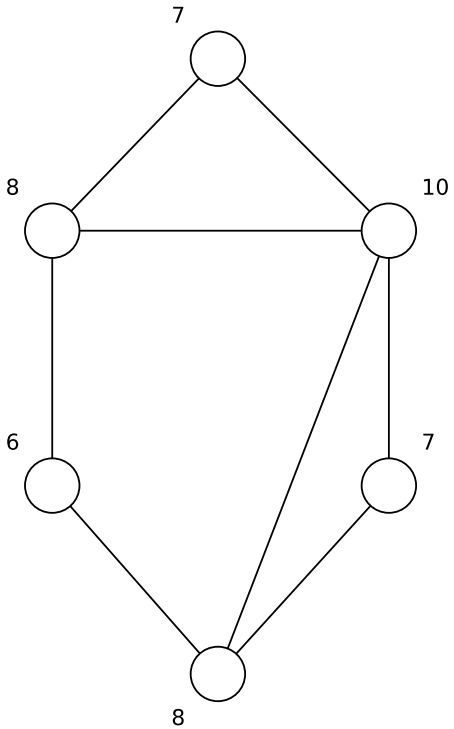


G_1

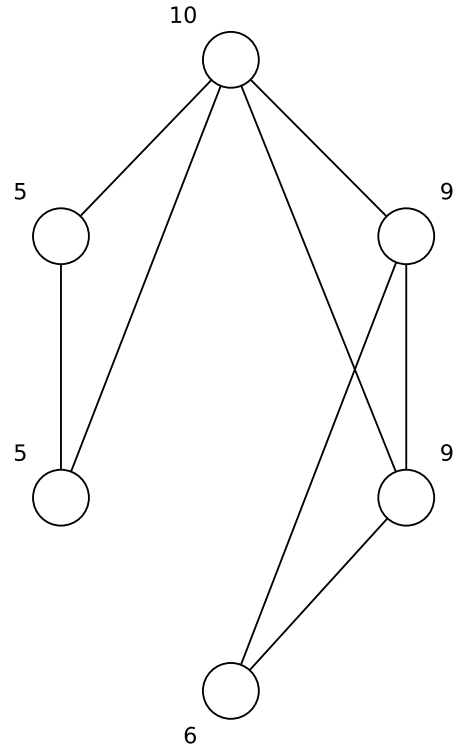


G_2

Are the label sets of G_1 and G_2 identical?



G_1



G_2

No!!!

Graphs are not isomorphic

Let $G^1, G^2, \ldots, G^h$ be the graphs emerging from graph G at the iteration $1, 2, \ldots, h$ of the Weisfeiler-Lehman algorithm

Then, the Weisfeiler-Lehman kernel is defined as:

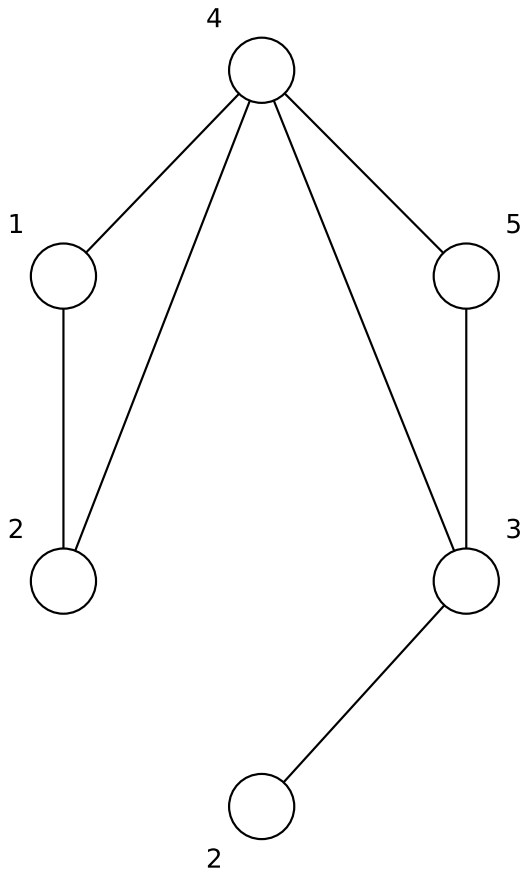
$$k^h_{WL}(G_1, G_2) = k(G_1, G_2) + k(G^1_1, G^1_2) + k(G^2_1, G^2_2) + \ldots + k(G^h_1, G^h_2)$$

where $k(\cdot, \cdot)$ is a base kernel (e.g. subtree kernel, shortest path kernel, ...)

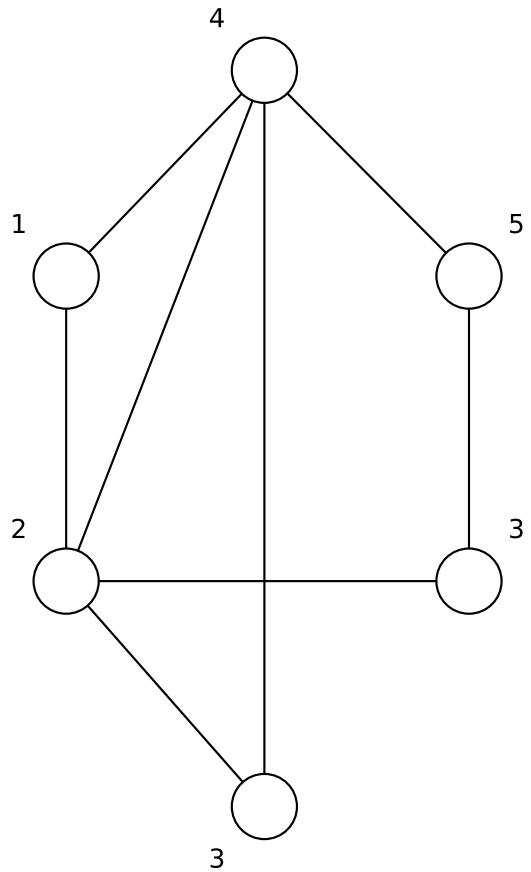
At each iteration of the Weisfeiler-Lehman algorithm:

- runs a graph kernel for labeled graphs
- the new kernel values are added to the ones of the previous iteration

Counts matching pairs of labels in two graphs after each iteration



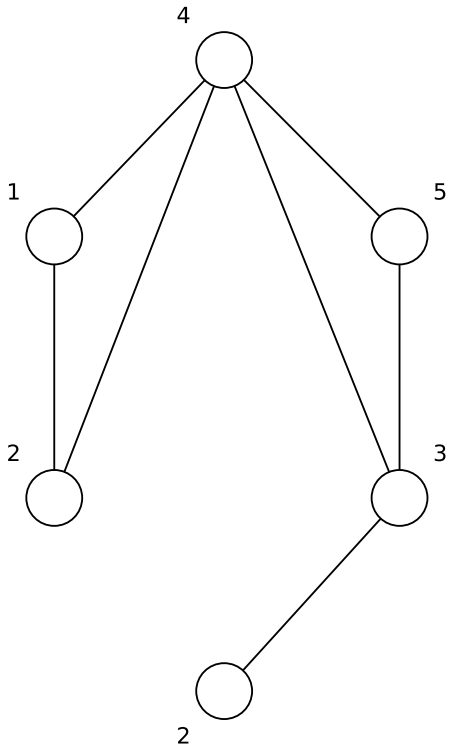
G_1



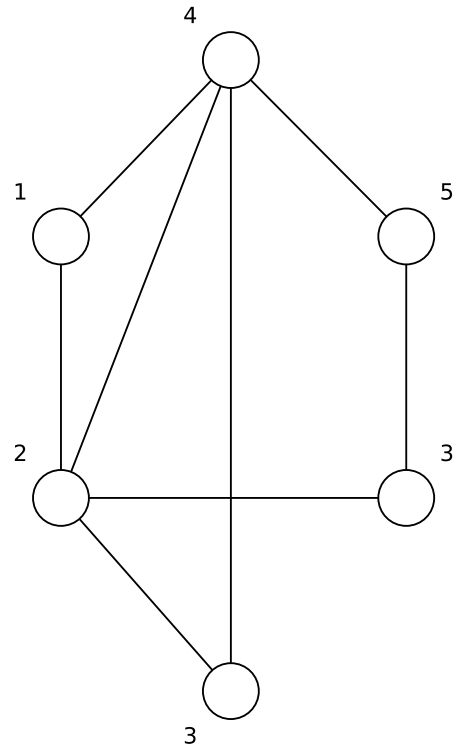
G_2

Feature vector for a graph G:

$\phi(G) = \{\#\text{nodes with label 1}, \#\text{nodes with label 2}, \ldots, \#\text{nodes with label } l\}$



G_1

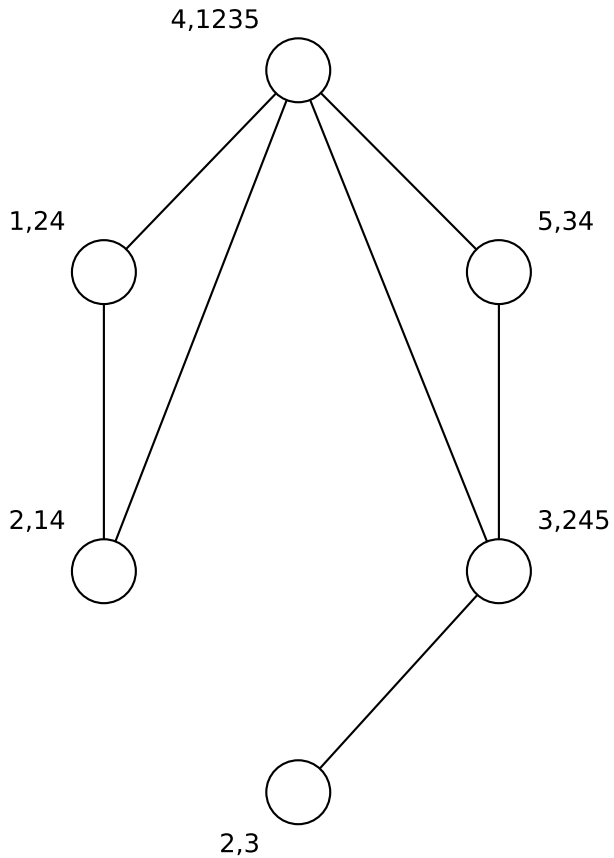


G_2

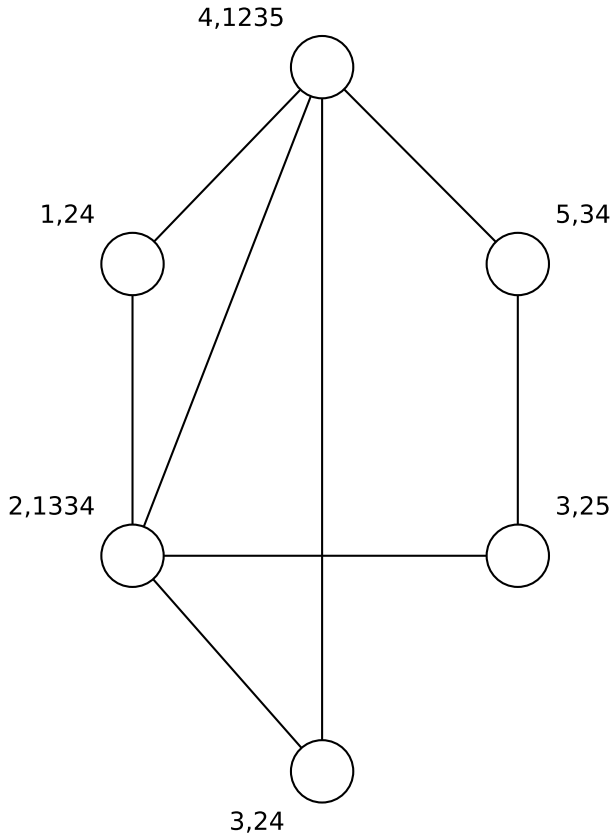
$$\phi(G_1) = \{1, 2, 1, 1, 1\}^T \qquad \phi(G_2) = \{1, 1, 2, 1, 1\}^T$$

$$k(G_1, G_2) = \phi(G_1)^T \phi(G_2) = 7$$

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices



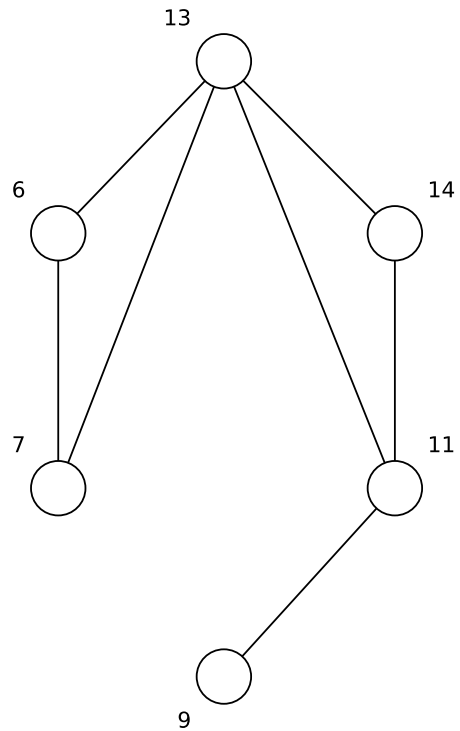
G_1



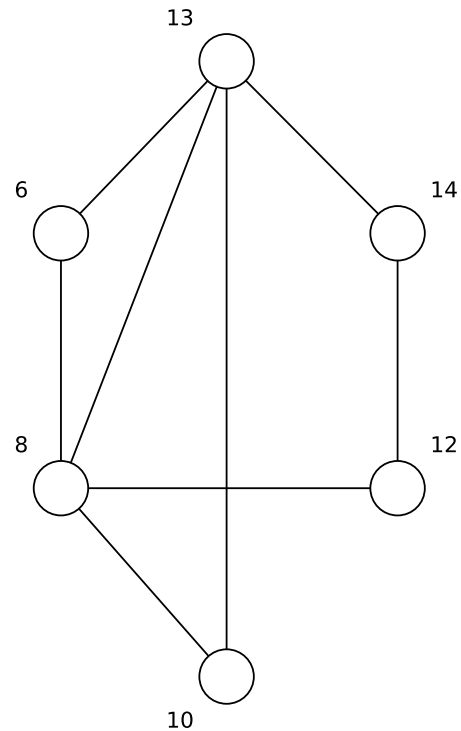
G_2

Second step: Compress the augmented labels into new, short labels:

- $1, 24 \rightarrow 6$
- $2, 14 \rightarrow 7$
- $2, 1334 \rightarrow 8$
- $2, 3 \rightarrow 9$
- $3, 24 \rightarrow 10$
- $3, 245 \rightarrow 11$
- $3, 25 \rightarrow 12$
- $4, 1235 \rightarrow 13$
- $5, 34 \rightarrow 14$

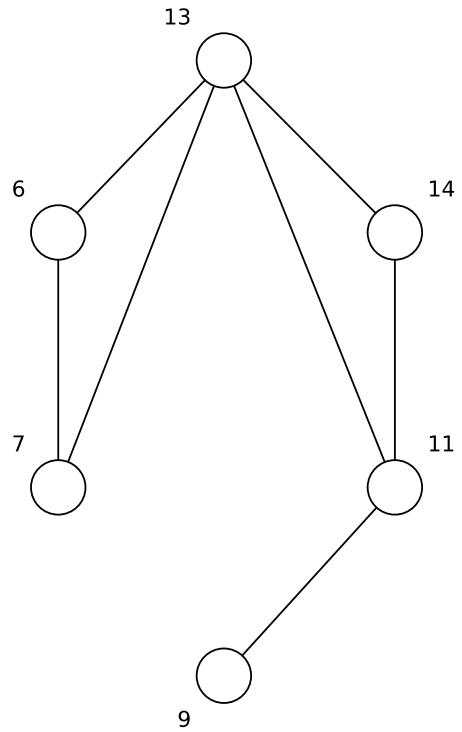


G_1

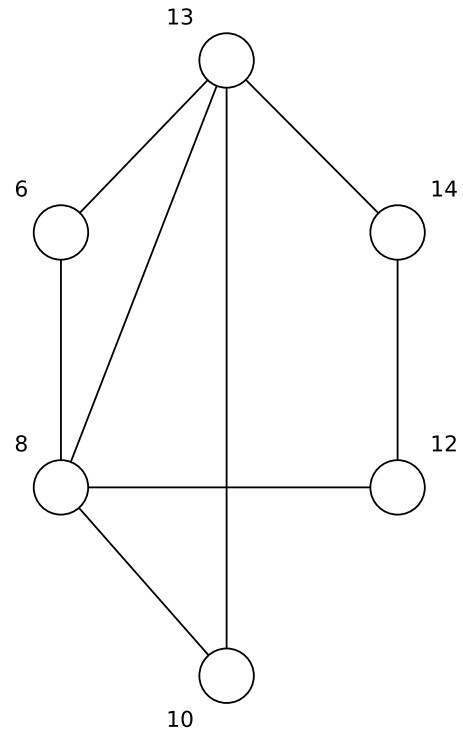


G_2

Third step: Compute kernel value for iteration $h = 1$ and add it to previous kernel value



G_1



G_2

$$\phi(G_1^1) = \{1,1,0,1,0,1,0,1,1\}^T \quad \phi(G_2^1) = \{1,0,1,0,1,0,1,1,1\}^T$$

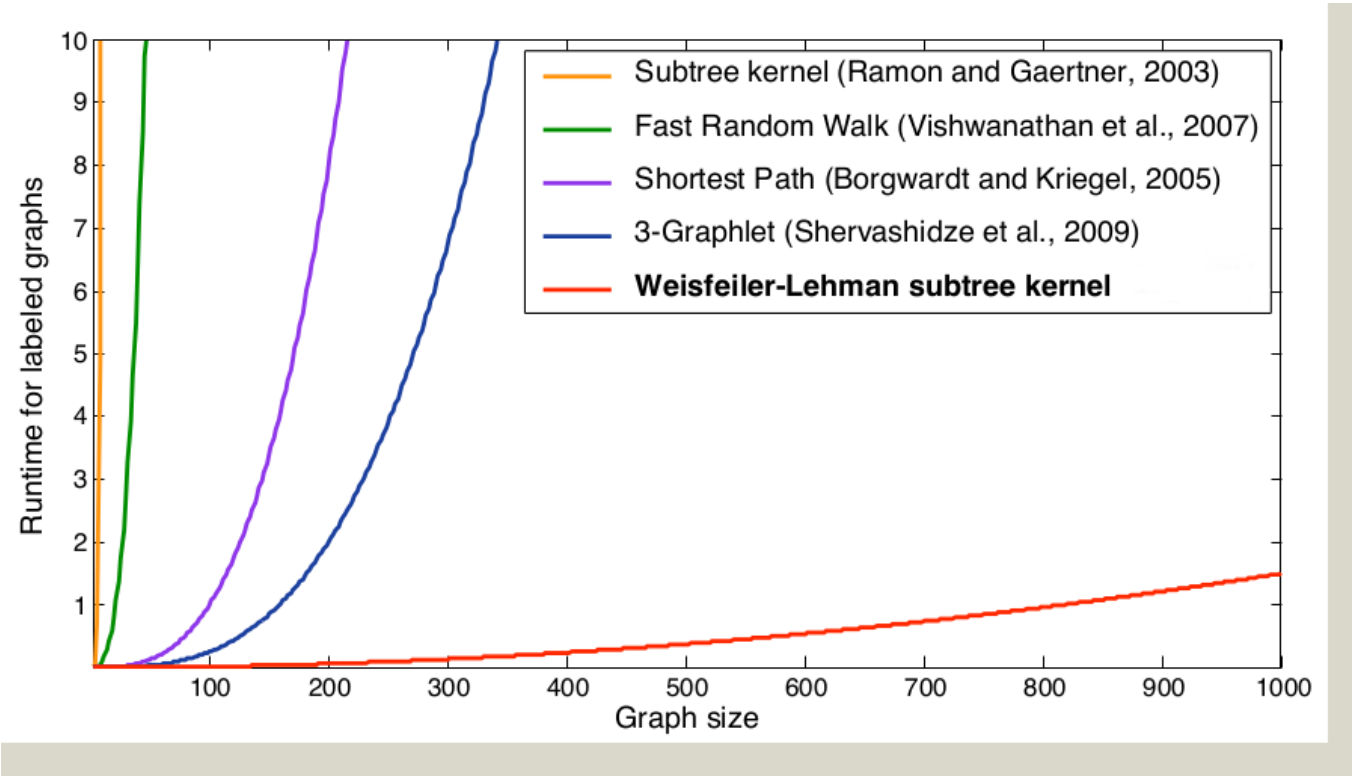
$$k(G_1^1, G_2^1) = \phi(G_1^1)^T \phi(G_2^1) = 3$$

$$k_{WL}^1(G_1, G_2) = k(G_1, G_2) + k(G_1^1, G_2^1) = 10$$

Computing the Weisfeiler-Lehman Subtree Kernel takes $\mathcal{O}(hm)$ time

- very **efficient**

Comparison to other well-known kernels



Local vs Global Graph Kernels

The previously presented graph kernels compare graphs in terms of features defined on small subgraphs:

- walks
- graphlets
- paths

Such kernels are inherently **local** since they

- operate only on a small set of vertices
- ignore the rest of the graph

However, several interesting properties are not captured in local substructures

- need graph kernels that capture **global** properties

Lovász ϑ kernel

Compares graphs based on the orthonormal representation associated with the Lovász number

- the orthonormal representation captures **global** graph properties

Orthonormal representation of a graph $G = (V, E)$:

- each vertex $i \in V$ is assigned a unit vector $\mathbf{u}_i$, $||\mathbf{u_i}|| = 1$
- let $U_G = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ be the set of all vectors
- for $i, j \in V$, if $(i, j) \notin E$, then $\mathbf{u}_i^T \mathbf{u}_j = 0$

An interesting orthonormal representation is associated with the Lovász number $\vartheta(G)$

Definition (Lovász number [Lov79])

For a graph $G = (V, E)$,

$$\vartheta(G) = \min_{\mathbf{c}, U_G} \max_{i \in V} \frac{1}{(\mathbf{c}^T \mathbf{u}_i)^2}$$

where the minimization is taken over all orthonormal representations U_G

Given a subset of vertices $S \subseteq V$, the Lovász value of the subgraph induced by S is:

$$\vartheta_S(G) = \min_{\mathbf{c}} \max_{\mathbf{u}_i \in U_{G|S}} \frac{1}{(\mathbf{c}^T \mathbf{u}_i)^2}$$

where $U_{G|S} = \{\mathbf{u}_i \in U_G : i \in S\}$ and U_G is the set of orthonormal representations of the vertices of G

The Lovász kernel is then defined as:

$$k_\vartheta(G_1, G_2) = \sum_{\substack{S_1 \subseteq V_1 \\ |S_1|=|S_2|}} \sum_{S_2 \subseteq V_2} \frac{1}{Z} k(\vartheta_{S_1}(G_1), \vartheta_{S_2}(G_2))$$

where $Z = \binom{n_1}{d}\binom{n_2}{d}$, $d = |S_1| = |S_2|$ and $k(\cdot, \cdot)$ is a base kernel (e.g. linear, gaussian)

Kernel captures global properties since

- $\vartheta_{S_1}(G_1), \vartheta_{S_2}(G_2)$ utilize representations of U_{G_1} and U_{G_2} respectively
- U_{G_1} and U_{G_2} are sets of representations generated for the whole graphs

Lovász ϑ kernel

Computing the Lovász ϑ kernel for a pair of graphs G_1, G_2 is very expensive since it computes:

- the Lovász numbers of the two graphs
- the Lovász value for all subgraphs of the two graphs

Solution: sampling

- Evaluate the Lovász value for a smaller number of subgraphs of size d

Pyramid Match Kernel

Embed all vertices in the d-dimensional vector space $\mathcal{R}^d$ as follows

- compute the eigendecomposition of the adjacency matrix
- use the eigenvectors of the d largest in magnitude eigenvalues

Such embeddings capture **global** properties of graphs

Example: eigenvector corresponding to greatest eigenvalue contains eigenvector centrality scores of vertices $\rightarrow$ global property

After embedding: each vertex is a point in the d-dimensional unit hypercube

Then, use pyramid match kernel [GD05], a kernel function over unordered feature sets:

- Each feature set is mapped to a multiresolution histogram
- The histogram pyramids are then compared using a weighted histogram intersection computation

Pyramid Match Kernel [NMV17]

Given a sequence of levels from 0 to L, then at level l the d-dimensional unit hypercube has

- 2^l cells along each dimension
- $D = 2^{dl}$ cells in total

Given a pair of graphs G_1, G_2,

- $H^l_{G_1}$ and $H^l_{G_2}$ denote the histograms of the two graphs at level l
- $H^l_{G_1}(i)$, $H^l_{G_2}(i)$ denote the number of vertices of the two graphs that lie in the i^{th} cell

The number of points in two sets which match at level l is then computed using the histogram intersection function

$$\mathcal{I}(H^l_{G_1}, H^l_{G_2}) = \sum_{i=1}^{D} \min\left(H^l_{G_1}(i), H^l_{G_2}(i)\right)$$

The matches that occur at level l also occur at levels $l+1, \ldots, L$.

We are interested in the number of new matches found at each level

$$\mathcal{I}(H^l_{G_1}, H^l_{G_2}) - \mathcal{I}(H^{l+1}_{G_1}, H^{l+1}_{G_2}) \text{ for } l = 0, \ldots, L-1$$

- These matches are weighted according to the size of that level's cells
- Matches found within smaller cells are weighted less than those made in larger cells
- The weight for level l is set equal to $\frac{1}{2^{L-l}}$

The pyramid match kernel is then defined as follows:

$$k_\Delta(G_1, G_2) = \mathcal{I}(H^L_{G_1}, H^L_{G_2}) + \sum_{l=0}^{L-1} \frac{1}{2^{L-l}} \left(\mathcal{I}(H^l_{G_1}, H^l_{G_2}) - \mathcal{I}(H^{l+1}_{G_1}, H^{l+1}_{G_2})\right)$$

Selected Publications I

László Babai, *Graph Isomorphism in Quasipolynomial Time*, arXiv preprint arXiv:1512.03547 (2015).

K. M. Borgwardt and H. Kriegel, *Shortest-path kernels on graphs*, Proceedings of the 5th International Conference on Data Mining, 2005, pp. 74–81.

László Babai and Eugene M Luks, *Canonical Labeling of Graphs*, Proceedings of the 15th Annual Symposium on Theory of Computing, 1983, pp. 171–183.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga, *Detecting self-mutating malware using control-flow graph matching*, International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2006, pp. 129–143.

Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel, *Protein function prediction via graph kernels*, Bioinformatics 21 (2005), no. suppl 1, i47–i56.

Jin Chen, Wynne Hsu, Mong Li Lee, and See-Kiong Ng, *NeMoFinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs*, Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining, 2006, pp. 106–115.

Hwann-Tzong Chen, Horng-Horng Lin, and Tyng-Luh Liu, *Multi-Object Tracking Using Dynamical Graph Matching*, Proceedings of the 2001 Conference on Computer Vision and Pattern Recognition, vol. 2, 2001, pp. II–210–II–217.

Remco Dijkman, Marlon Dumas, and Luciano García-Bañuelos, *Graph Matching Algorithms for Business Process Model Similarity Search*, International Conference on Business Process Management, 2009, pp. 48–63.

Selected Publications II


Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis, *Frequent sub-structure-based approaches for classifying chemical compounds*, IEEE Transactions on Knowledge and Data Engineering **17** (2005), no. 8, 1036–1050.


Eric H Davidson, Jonathan P Rast, Paola Oliveri, Andrew Ransick, Cristina Calestani, Chiou-Hwa Yuh, Takuya Minokawa, Gabriele Amore, Veronica Hinman, Cesar Arenas-Mena, et al., *A Genomic Regulatory Network for Development*, Science **295** (2002), no. 5560, 1669–1678.


Andreas Fischer, Kaspar Riesen, and Horst Bunke, *Graph Similarity Features for HMM-Based Handwriting Recognition in Historical Documents*, Proceedings of the 2010 International Conference on Frontiers in Handwriting Recognition, 2010, pp. 253–258.


Kristen Grauman and Trevor Darrell, *The pyramid match kernel: Discriminative classification with sets of image features*, Proceedings of the 10th International Conference on Computer Vision, 2005, pp. 1458–1465.


T. Gärtner, P. Flach, and S. Wrobel, *On Graph Kernels: Hardness Results and Efficient Alternatives*, Learning Theory and Kernel Machines, 2003, pp. 129–143.


M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W.H. Freeman & Co, 1979.


David Gitchell and Nicholas Tran, *Sim: A Utility for Detecting Similarity in Computer Programs*, Proceedings of the 30th Technical Symposium on Computer Science Education, vol. 31, 1999, pp. 266–270.


Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li, *A survey of graph edit distance*, Pattern Analysis and Applications **13** (2010), no. 1, 113–129.

Selected Publications III


Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck, *Structural Detection of Android Malware using Embedded Call Graphs*, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, 2013, pp. 45–54.


D HAUSSLER, *Convolution kernels on discrete structures*, Technical Report (1999).


T. Horváth, T. Gärtner, and S. Wrobel, *Cyclic Pattern Kernels for Predictive Graph Mining*, Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004, pp. 158–167.


Linus Hermansson, Tommi Kerola, Fredrik Johansson, Vinay Jethava, and Devdatt Dubhashi, *Entity Disambiguation in Anonymized Graphs using Graph Kernels*, Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, 2013, pp. 1037–1046.


DK Isenor and Safwat G Zaky, *FINGERPRINT IDENTIFICATION USING GRAPH MATCHING*, Pattern Recognition **19** (1986), no. 2, 113–122.


Fredrik Johansson, Vinay Jethava, Devdatt Dubhashi, and Chiranjib Bhattacharyya, *Global graph kernels using geometric embeddings*, Proceedings of the 31st International Conference on Machine Learning, 2014, pp. 694–702.


H. Kashima, K. Tsuda, and A. Inokuchi, *Marginalized Kernels Between Labeled Graphs*, Proceedings of the 20th Conference in Machine Learning, 2003, pp. 321–328.


Uta Lösch, Stephan Bloehdorn, and Achim Rettinger, *Graph Kernels for RDF Data*, Extended Semantic Web Conference, 2012, pp. 134–148.


László Lovász, *On the Shannon Capacity of a Graph*, IEEE Transactions on Information Theory **25** (1979), no. 1, 1–7.

Selected Publications IV


Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert, *Extensions of Marginalized Graph Kernels*, Proceedings of the 21st International Conference on Machine Learning, 2004, pp. 552–559.


Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis, *Matching Node Embeddings for Graph Similarity*, Submitted to the 31st AAAI Conference on Artificial Intelligence, 2017.


Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina, *Web graph similarity for anomaly detection*, Journal of Internet Services and Applications 1 (2010), no. 1, 19–30.


Nataša Pržulj, *Biological network comparison using graphlet degree distribution*, Bioinformatics 23 (2007), no. 2, e177–e183.


Ronald C Read and Derek G Corneil, *The Graph Isomorphism Disease*, Journal of Graph Theory 1 (1977), no. 4, 339–363.


Jan Ramon and Thomas Gärtner, *Expressivity versus Efficiency of Graph Kernels*, First International Workshop on Mining Graphs, Trees and Sequences, 2003, pp. 65–74.


John W Raymond, Eleanor J Gardiner, and Peter Willett, *RASCAL: Calculation of Graph Similarity Using Maximum Common Edge Subgraphs*, The Computer Journal 45 (2002), no. 6, 631–644.


François Rousseau, Emmanouil Kiagias, and Michalis Vazirgiannis, *Text Categorization as a Graph Classification Problem*, Proceedings of the 52th Annual Meeting of the Association for Computational Linguistics and the 6th International Joint Conference on Natural Language Processing, 2015.

Selected Publications V


Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi, *Graph Kernels for Chemical Informatics*, Neural Networks **18** (2005), no. 8, 1093–1110.


John W Raymond and Peter Willett, *Maximum common subgraph isomorphism algorithms for the matching of chemical structures*, Journal of Computer-Aided Molecular Design **16** (2002), no. 7, 521–533.


Uwe Schöning, *Graph Isomorphism Is in the Low Hierarchy*, Journal of Computer and System Sciences **37** (1988), no. 3, 312–323.


N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S. Vishwanathan, *Efficient Graphlet Kernels for Large Graph Comparison*, Proceedings of the International Conference on Artificial Intelligence and Statistics, 2009, pp. 488–495.


N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, *Weisfeiler-Lehman Graph Kernels*, The Journal of Machine Learning Research **12** (2011), 2539–2561.


Chien-Chung Shen and Wen-Hsiang Tsai, *A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion*, IEEE Transactions on Computers **100** (1985), no. 3, 197–203.


Shanhu Shang, Ning Zheng, Jian Xu, Ming Xu, and Haiping Zhang, *Detecting malware variants via function-call graph similarity*, Proceedings of the 5th International Conference on Malicious and Unwanted Software, 2010, pp. 113–120.


Makoto Takashima, Atsuhiko Ikeuchi, Shoichi Kojima, Toshikazu Tanaka, Tamaki Saitou, and Jun-ichi Sakata, *A Circuit Comparison System with Rule-based Functional Isomorphism Checking*, Proceedings of the 25th Conference in Design Automation, 1988, pp. 512–516.

Selected Publications VI


S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt, *Graph Kernels*, The Journal of Machine Learning Research 11 (2010), 1201–1242.


Laurenz Wiskott, Jean-Marc Fellous, N Kuiger, and Christoph Von Der Malsburg, *Face Recognition by Elastic Bunch Graph Matching*, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (1997), no. 7, 775–779.


Boris Weisfeiler and AA Lehman, *A reduction of a graph to a canonical form and an algebra arising during this reduction*, Nauchno-Technicheskaya Informatsia 2 (1968), no. 9, 12–16.


Nikil Wale, Xia Ning, and George Karypis, *Trends in Chemical Graph Data Mining*, Managing and Mining Graph Data, 2010, pp. 581–606.


Nikil Wale, Ian A Watson, and George Karypis, *Comparison of Descriptor Spaces for Chemical Compound Retrieval and Classification*, Knowledge and Information Systems 14 (2008), no. 3, 347–375.


Chang Hun You, Lawrence B Holder, and Diane J Cook, *Application of Graph-based Data Mining to Metabolic Pathways*, Proceedings of the 6th International Conference on Data Mining, 2006, pp. 169–173.


P. Yanardag and S. Vishwanathan, *A Structural Smoothing Framework For Robust Graph Comparison*, Advances in Neural Information Processing Systems, 2015, pp. 2125–2133.