Graph Kernels

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA

Contents

- Overview of Graph Kernels
- Graph Kernels methods:
  - Graphlet kernel
  - Shortest path kernel
  - Random walk kernel
  - Weisfeiler-Lehman (WL) graph kernel: WL relabelling process
- Subgraph Matching Kernel
- Sample code of Graph Kernels

Motivation

- From an algorithmic perspective, graphs are the most general data structures, as all common data types are simple instances of graphs
  - E.g. 1, A time series of vectors can be represented as a graph that contains one node per time step, and consecutive steps are linked by an edge
  - E.g. 2, A string is a graph in which each node represents one character, and consecutive characters are connected by an edge

- Given their generality, the natural question to ask is:

  Why have graphs not been the common data structure in computer science for decades?

- The answer is simple:

  Their comparison is computationally expensive

  Graphs are prisoners of their own their flexibility

Why Graph Kernels?

- Graph kernels are one of the most recent approaches to graph comparison
- Interestingly, graph kernels employ concepts from all three traditional branches of graph comparison:
  - Measure similarity in terms of isomorphic substructures of graphs
  - Allow for inexact matching of nodes, edges, and labels
  - Treat graphs as vectors in a Hilbert space of graph features

Graph Comparison

- Given two graphs G and G'. The problem of graph comparison is to find a mapping:

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

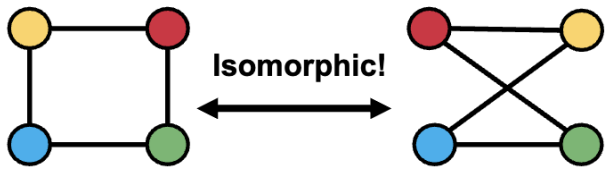
- Such that $s(G, G')$ quantifies the similarity (dissimilarity) of G and G'

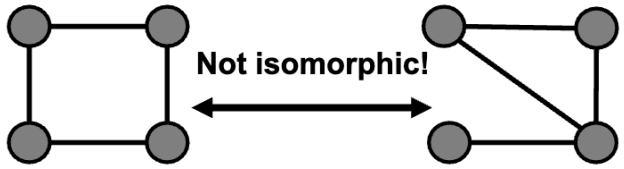
Applications of Graph Comparison

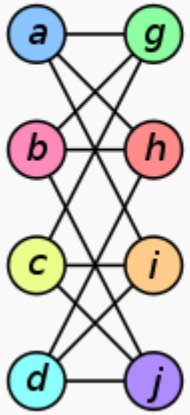
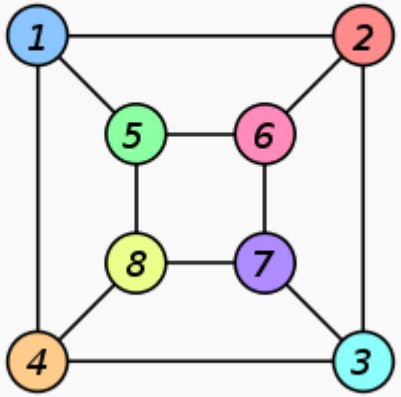
- Function prediction of chemical compounds
- Structural comparison and function prediction of protein structures
- Comparison of social networks
- Analysis of semantic structures in NLP
- Comparison of UML diagrams

Graph Isomorphism

- Graph isomorphism:
  - Find a mapping f of the vertices of G_1 to the vertices of G_2 such that G_1 and G_2 are identical
  - i.e. (x, y) is an edge of G_1 if $(f(x), f(y))$ is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are called isomorphic
  - No polynomial-time algorithm is known for graph isomorphism
  - Neither is it known to be NP-hard

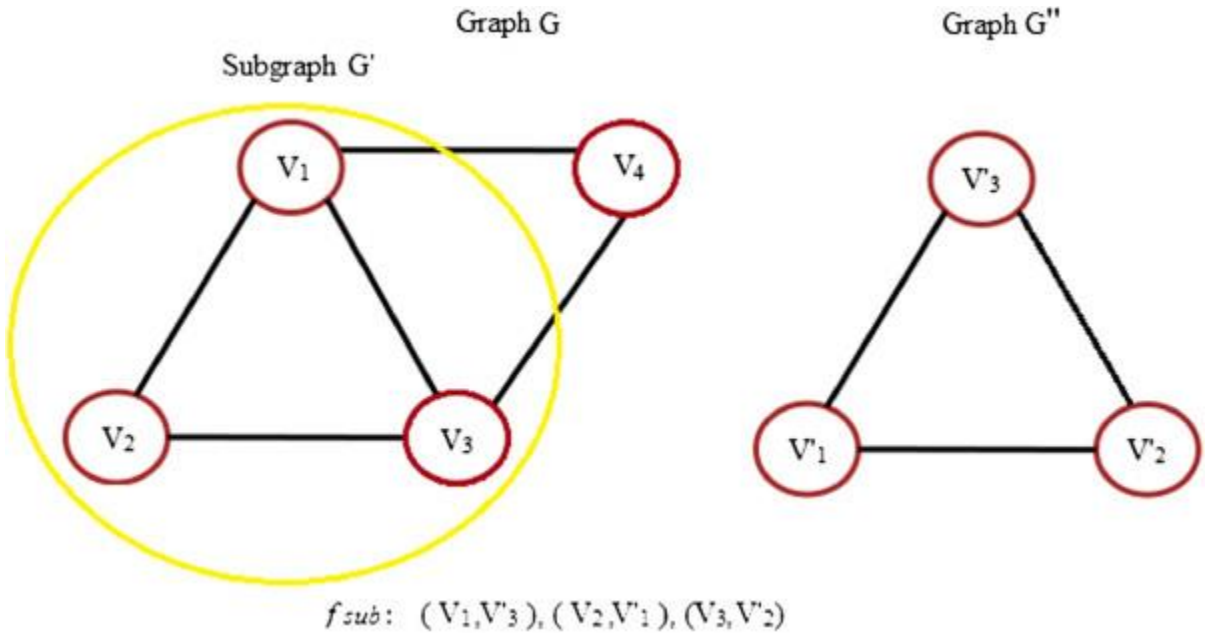




Graph G	Graph H	An isomorphism between G and H
		f(a) = 1 f(b) = 6 f(c) = 8 f(d) = 3 f(g) = 5 f(h) = 2 f(i) = 4 f(j) = 7

Graph Isomorphism

- Subgraph isomorphism:
  - Subgraph isomorphism asks if there is a subset of edges and vertices of G_1 that is isomorphic to a smaller graph G_2
  - Subgraph isomorphism is **NP-complete**



Subgraph Isomorphism Challenges

- **NP-completeness:**
  - A decision problem C is NP-complete if
    - C is in NP
    - C is NP-hard, i.e. every other problem in NP is reducible to it

- **Problems for the practitioner:**
  - Excessive runtime in worst case
  - Runtime may grow exponentially with the number of nodes
  - For larger graphs with many nodes and for large datasets of graphs, this is an enormous problem

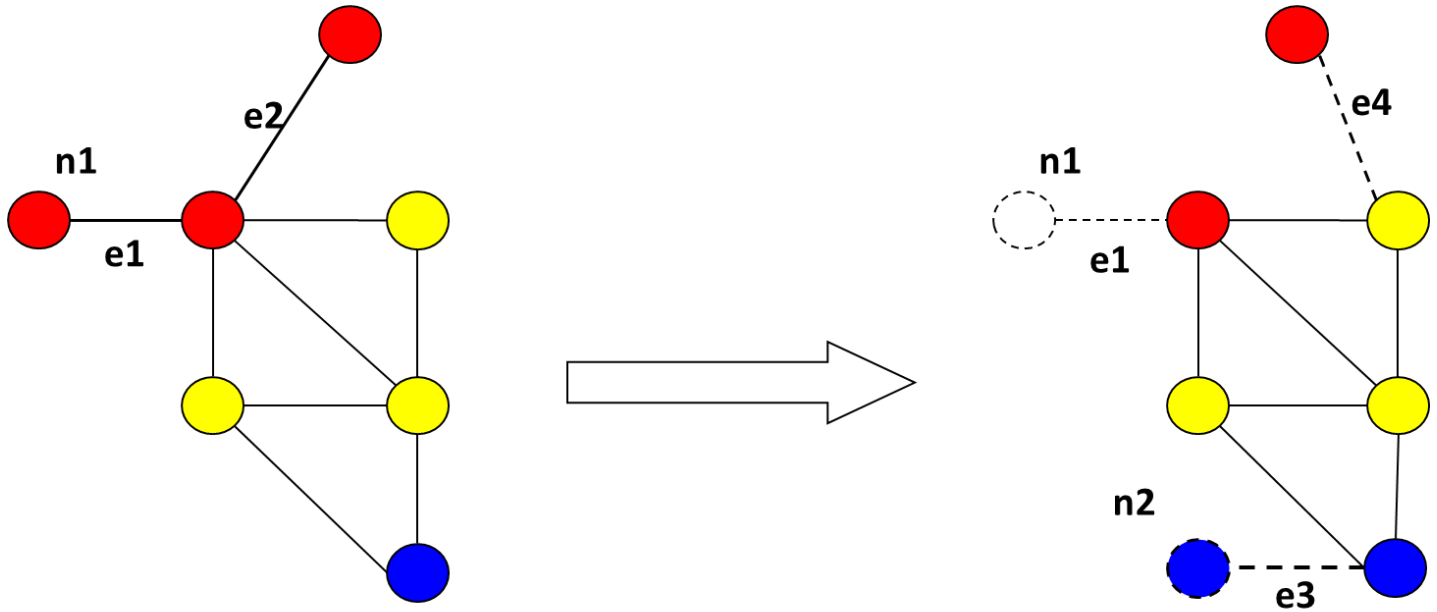
⇒ Wanted: Polynomial-time similarity measure for graphs

Graph Edit Distances

- Principle:
  - Count operations that are necessary to transform g_1 into g_2
  - Assign costs to different types of operations (edge/node insertion/deletion, modification of labels)

$$GED(g_1, g_2) = \min_{(e_1,\ldots,e_k)\in\mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e_i)$$

Where $P(g_1, g_2)$ denotes the set of edit paths transforming $g_1$into g_2 and $c(e) \geq 0$ is the cost of each graph edit operation z



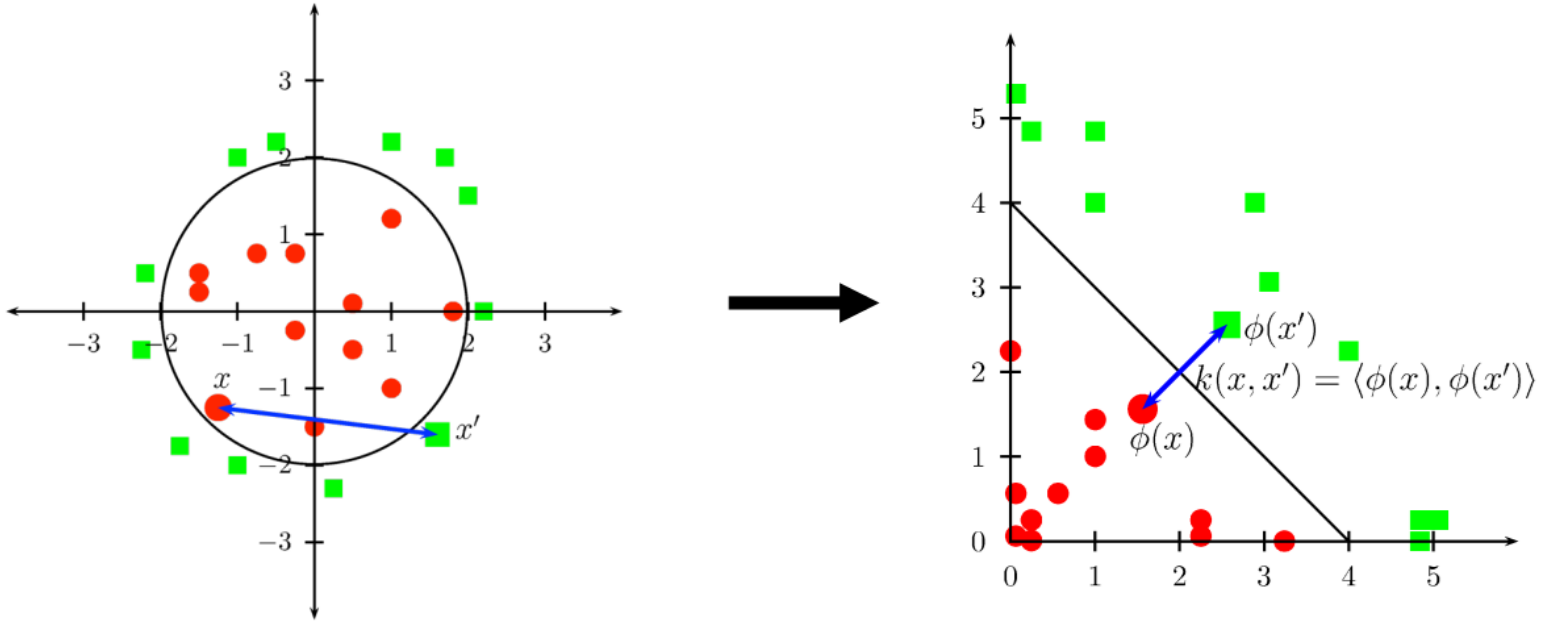
Graph Edit Distances Pros and Cons

- Advantages:
  - Captures partial similarities between graphs
  - Allows for noise in the nodes, edges and their labels
  - Flexible way of assigning costs to different operations
- Disadvantages:
  - Contains subgraph isomorphism check as one intermediate step
  - Choosing cost function for different operations is difficult

What is a Kernel? (Schölkopf ,1997)

- Kernel is a type of measures of similarity
- Mapping two objects x and x' via mapping ϕ into feature space *H*
- Measure their similarity in *H* as $\langle \phi(x), \phi(x') \rangle$.
- **Kernel Trick**: Compute inner product in *H* as kernel in input space

$$k(x, x') = \langle \phi(x), \phi(x') \rangle.$$

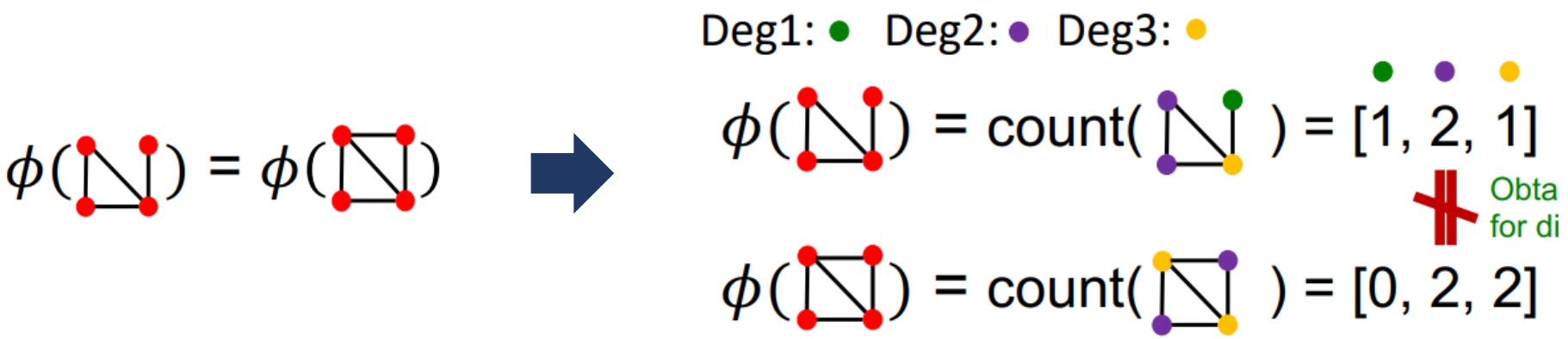


https://link.springer.com/chapter/10.1007/BFb0020217

What is a Kernel? (Schölkopf ,1997)

- Kernel is a type of measures of similarity
- Mapping two objects x and x' via mapping ϕ into feature space *H*
- Measure their similarity in *H* as $\langle \phi(x), \phi(x') \rangle$.
- **Kernel Trick**: Compute inner product in *H* as kernel in input space

$$k(x, x') = \langle \phi(x), \phi(x') \rangle.$$



What is a Graph Kernel?

- Instance of R-convolution kernels by Haussler (1999):
  - R-convolution kernels compare decompositions of two structured objects

$$k_{convolution}(x, x') = \sum_{(x_d, x) \in \mathrm{R}} \sum_{(x'_d, x') \in \mathrm{R}} k_{parts}(x_d, x'_d)$$

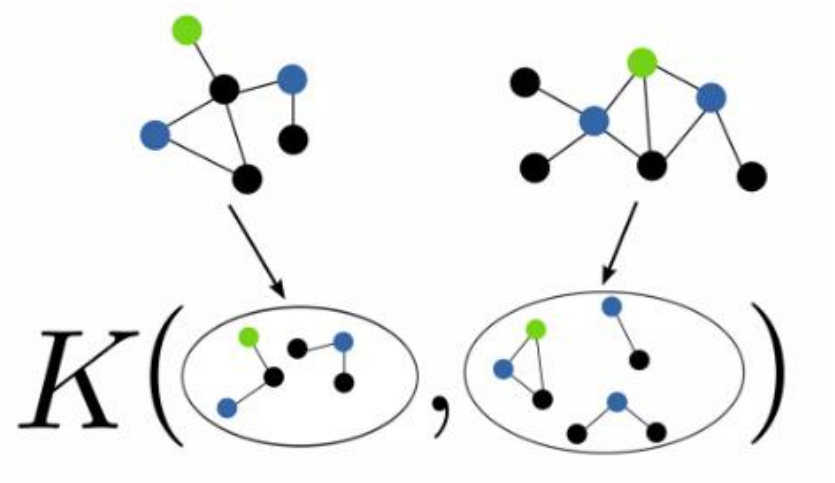
  - Decompose graphs into their substructures and add up the pairwise similarities between these substructures

- Concept:
  - Kernel function to measure the similarity of pairs of graphs by computing an inner product on graphs
  - Compare substructures of graphs that are computable in polynomial time

https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5ee0d8aeb2cb01ef4d8a858d234e72a7400c03ac

Kenels based on bags of patterns

- Graph kernels based on bags of patterns:
  - Extraction of a set of patterns from graphs
  - Comparison between patterns
  - Comparison between bags of patterns



Hardness Results (Gaertner, Flach, Wrobel, COLT 2003)

- Link to graph isomorphism
  - Let $k(G, G') = < \phi(G), \phi(G') >$ be a graph kernel
  - If ϕ is injective (one-to-one), *k* is called a complete graph kernel

Proposition 1

- Computing any complete graph kernel is at least as hard as deciding whether two graphs are isomorphic.

- Criteria for a good graph kernel:
  - Expressive
  - Efficient to compute
  - Positive definite
  - Applicable to wide range of graphs

https://link.springer.com/chapter/10.1007/978-3-540-45167-9_11

Example

- Let X represent the alphabet where each element of X is a set
- Define the intersection kernel to be $K(X_1, X_2) = |X_1 \cap X_2|$
- Given a graph G, obtain the set of all its subgraphs, denoted by $X(G)$
- Then the following is a graph kernel

$$K(G_1, G_2) = K(X(G_1), X(G_2)) = |X(G_1) \cap X(G_2)|$$

- This kernel looks for all overlapping substructures between two graphs
- High complexity

Graph Kernel Applications: Chemoinformatics (Ralaivola et al., 2005)

- Graph kernels inspired by concepts from chemoinformatics:
  - Define three new kernels (Tanimoto, MinMax, Hybrid) for function prediction of chemical compounds
  - Based on the idea of molecular fingerprints and
  - Counting labeled paths of depth up to d using depth-first search from each possible node

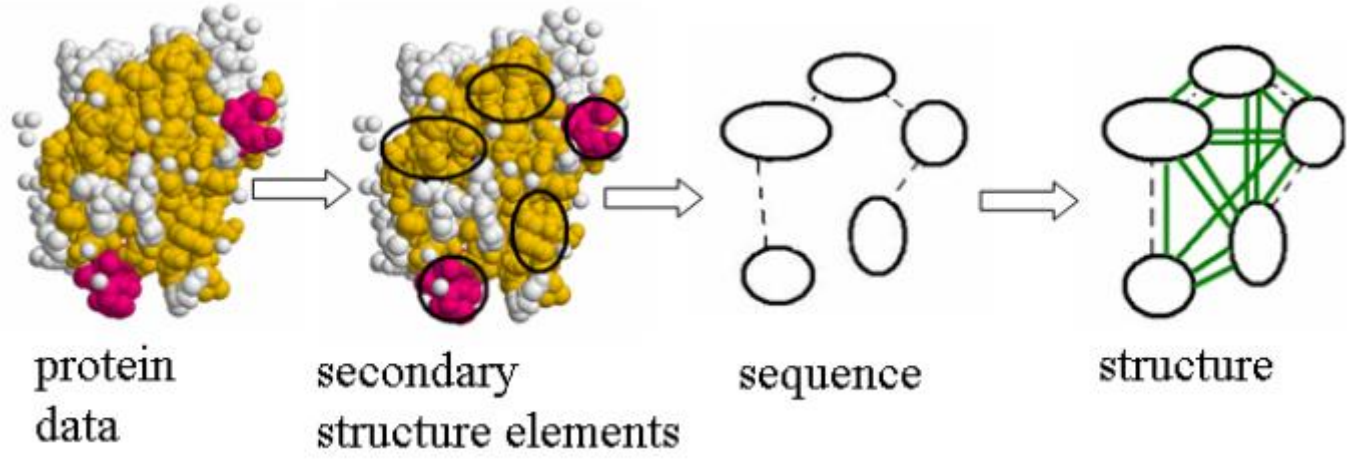
- **Properties**:
  - Tailored for applications in chemical informatics
  - Exploit the small size and low average degree of these molecular graphs.

Chemical Compound Classification (Wale et al, ICDM 2006)

- New kernels and experimental comparison of existing techniques:
  - Define a kernel that considers *graph fragments*: Subgraphs with a maximum of l edges
  - Fragment-based kernels outperform kernels using frequent subgraphs and walk-based kernels.
- **Four choices in kernel design for chemical compound**:
  - Generation of patterns (learnt from dataset versus defined by expert).
  - 'Preciseness' of the patterns (whether subgraph features map to the same dimension in feature space)
  - Complete coverage (whether the patterns occur in all of the instances of the dataset)
  - Complexity of patterns (walks and cycles versus frequent subgraphs)

Applications: Protein Function Prediction (B. et al, ISMB 2005)

- Predict the function of a protein from its structure
- Model protein structure as graph
- Use graph kernels to measure structural similarity and SVM to predict functional class
- Reaches competitive results on benchmark datasets

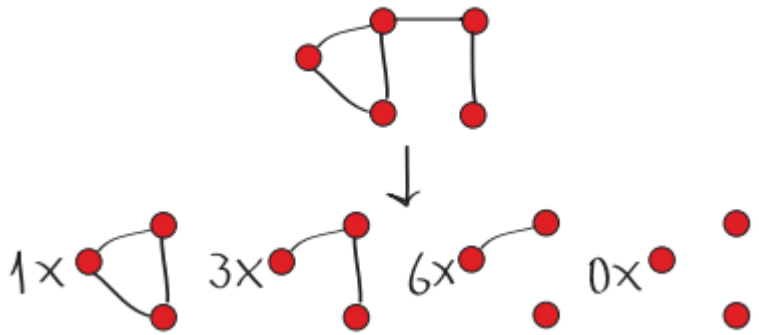


Base Kernels

- **Graphlet kernel** [Shervashidze et al., 2009]:
  - Counts identical pairs of graphlets.
    (i.e., subgraphs with k nodes where k = 3, 4, 5) in two graphs
- **Shortest path kernel** [Borgwardt and Kriegel, 2005]:
  - Counts pairs of shortest paths in two graphs having the same source and sink labels and identical length
- **Random walk kernel** [Kashima et al.,2003; Gärtner et al., 2003]:
  - Counts pairs of random walks and compare walks in two input graphs
- **Weisfeiler-Lehman** [Weisfeiler-Lehman isomorphic testing, 1968]:
  - WL relabelling process
- **Pyramid match graph kernel** [Nikolentzos et al.,2017b]:
  - Embeds the vertices of the input graphs in a vector space
  - It then partitions the feature space into regions of increasingly larger size and takes a weighted sum of the matches that occur at each level

Graphlet Kernel (B., Petri, et al., MLG 2007)

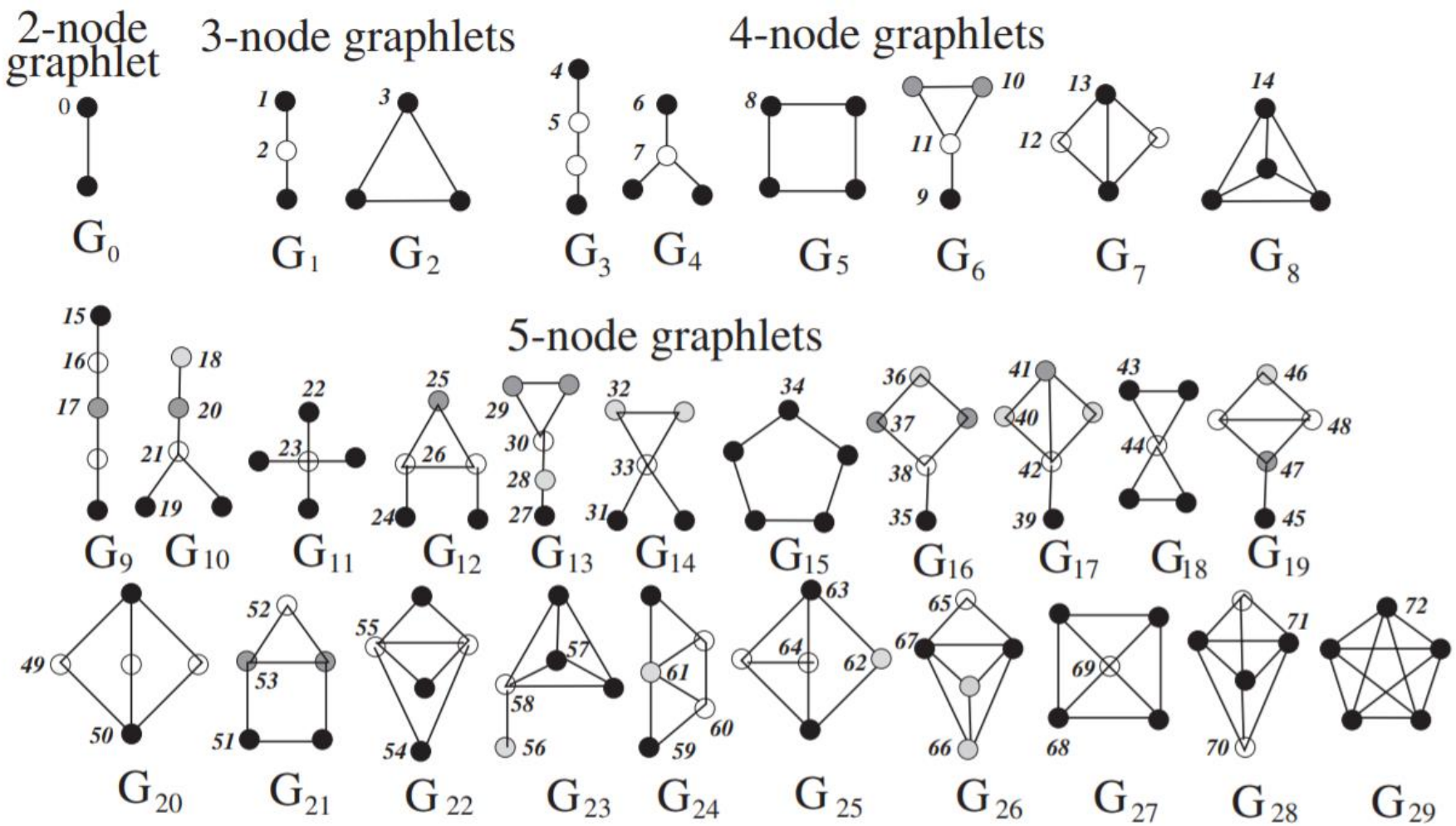
- Principle:
  - Count subgraphs of limited size k in G and G'.
  - These subgraphs are referred to as graphlets (Przulj, Bioinformatics 2007)
  - Define graph kernel that counts isomorphic graphlets in two graphs
- Runtime problems:
  - Pairwise test of isomorphism is expensive
  - Number of graphlets scales as $O(nk)$
- Two solutions on unlabelled graphs:
  - Precompute isomorphisms
  - Sample graphlets
- Disadvantage:
  - Same solutions not feasible on labelled graphs



Size k-graphlet = 3, number of node n = 5

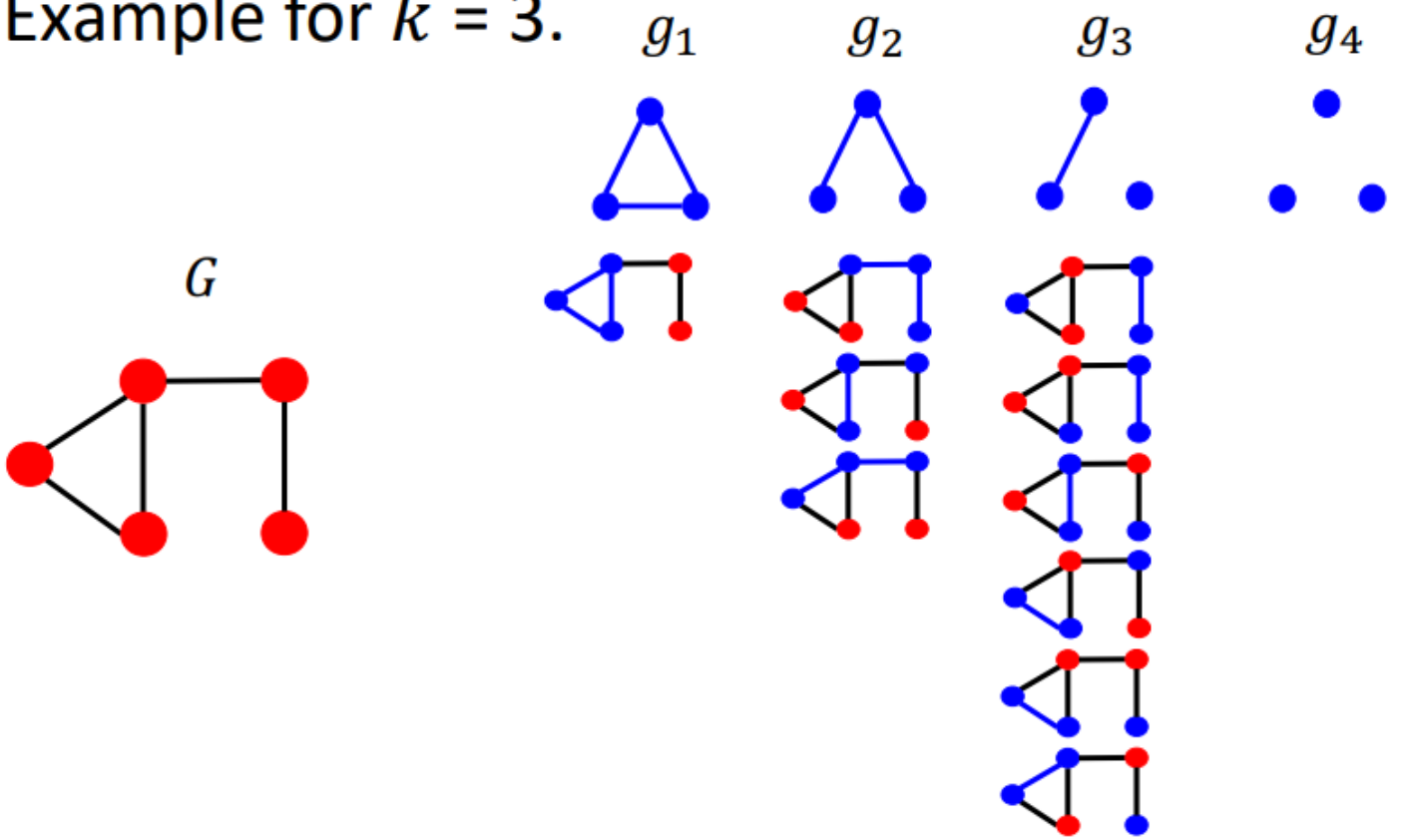
Graphlet Kernel (B., Petri, et al., MLG 2007)

- Variety of subgraphs according to size *k*:



Graphlet Kernel (B., Petri, et al., MLG 2007)

- Count subgraphs of limited size 3:
  - Example for $k = 3$.



$$f_G = (1, \quad 3, \quad 6, \quad 0)^T$$

Shortest Path Kernel: More General Form

Floyd-transformation:

- Given an input graph G, outputs a shortest-path graph S
  - S contains the same set of nodes as the input graph G
  - There exists an edge between all nodes in S which are connected by a path in G
  - Every edge in S between two nodes is labelled by the shortest distance between these two nodes
  - This transformation can be done in $O(n^3)$ time

Shortest Path Kernel: More General Form

Definition (Shortest-path graph kernel):

- Let G_1 and G_2 be two graphs that are Floyd-transformed into S_1 and S_2
- We can then define shortest-path graph kernel on $S_1 = (G_1, E_1)$ and $S_2 = (G_2, E_2)$ as:

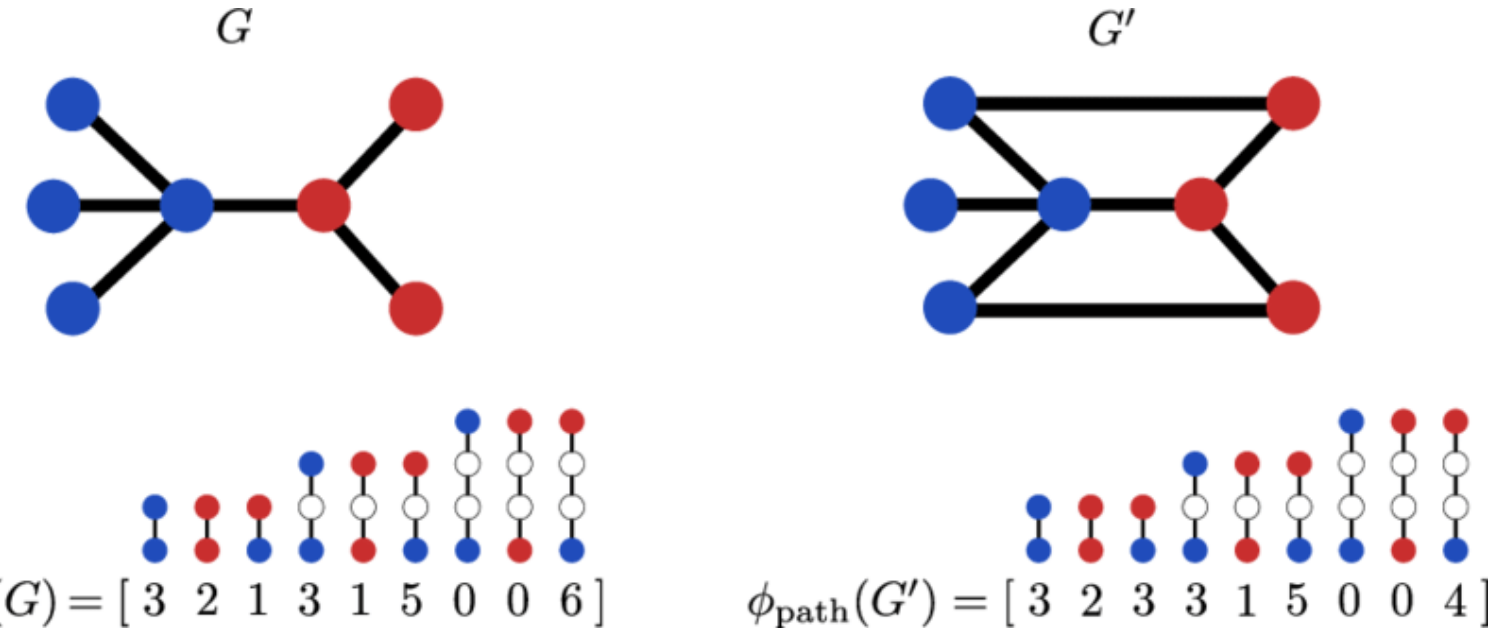
$$k_{shortest\ paths}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}^{(1)}(e_1, e_2),$$

where K_{walk} is a positive definite kernel on edge walks of length 1

Shortest Path Kernel: More General Form

Shortest-path graph kernel:

$$k_{shortest\ paths}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2),$$



Random Walk Kernel

- Random walk-kernels are based on the idea to count the number of matching walks in two input graphs. All pairs of matching walks in two input graphs G_1 and G_2 via a direct product graph G_X:

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda_n A_\times^n]_{ij},$$

where A_X is the adjacency matrix of G_X, defined from

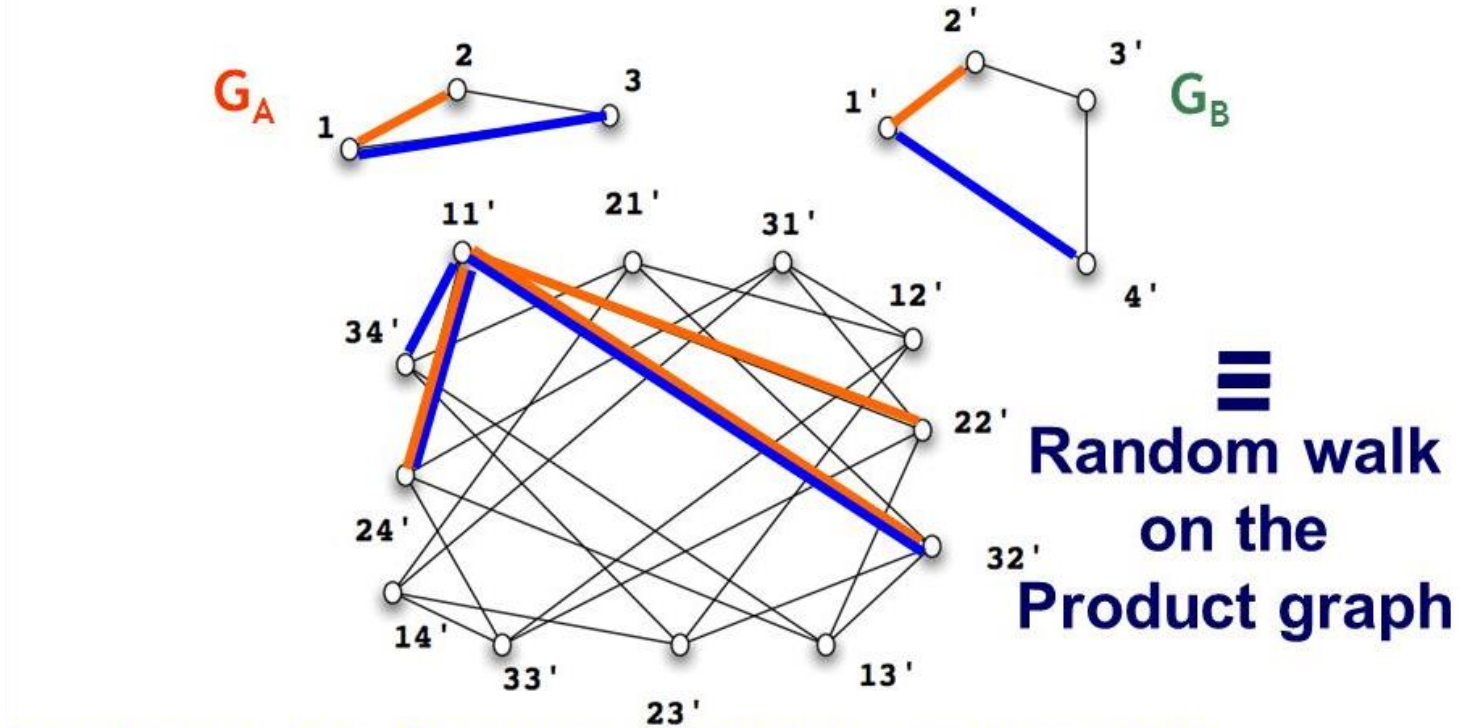
$$V_\times(G_1 \times G_2) = \{(v_1, w_1) \in V_1 \times V_2 : label(v_1) = label(w_1)\}$$

$$E_\times(G_1 \times G_2) = \{((v_1, w_1), (v_2, w_2)) \in V^2(G_1 \times G_2) : (v_1, v_2) \in E_1 \wedge (w_1, w_2) \in E_2 \wedge (label(v_1, v_2) = label(w_1, w_2))\}$$

λ_n must be chosen appropriately for k_X to converge

Random Walk Kernel

- Random walk-kernels are based on the idea to count the number of matching walks in two input graphs



[Kashima+ '03, Gaertner+ '03, Vishwanathan '10]

Isomorphism of Graphs

Definition:

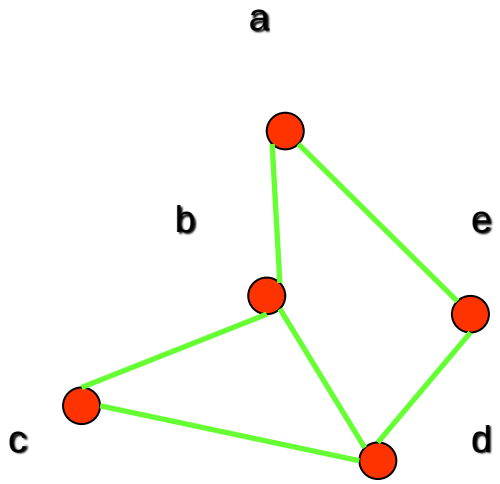
- The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a bijection (a one-to-one and onto function) f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2, for all a and b in V_1
- Such a function f is called an isomorphism
- In other words, G_1 and G_2 are isomorphic if their vertices can be ordered in such a way that the adjacency matrices $M(G_1)\ and\ M(G_2)$ are identical

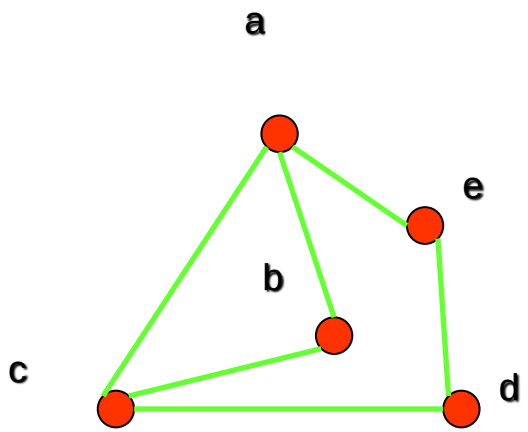
Isomorphism of Graphs

- For this purpose, we can check invariants, that is, properties that two isomorphic simple graphs must both have
- For example, they must have
  - The same number of nodes,
  - The same number of edges,
  - And the same degrees of nodes
- Note that two graphs that differ in any of these invariants are not isomorphic, but two graphs that match in all of them are not necessarily isomorphic

Isomorphism of Graphs: Example 1

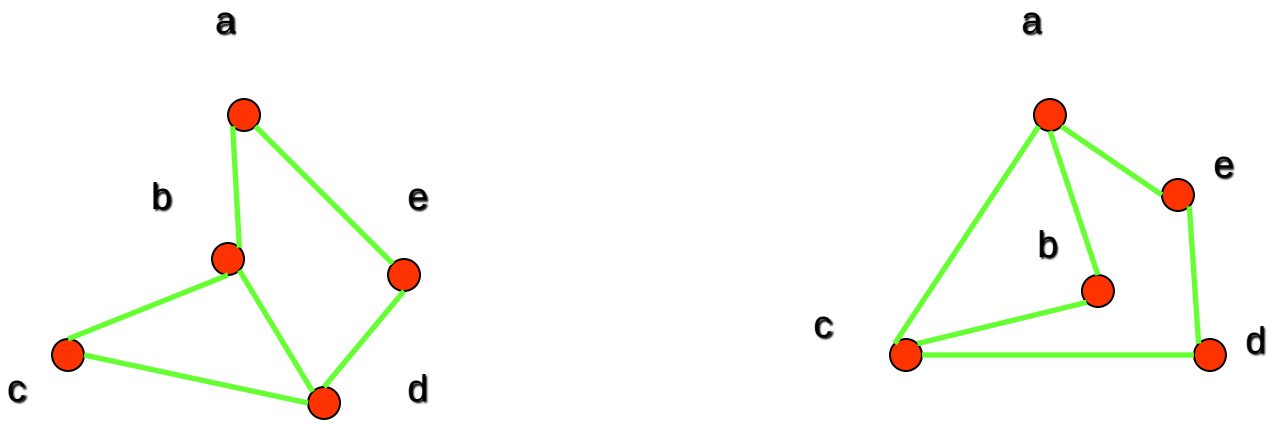
- Are the following two graphs isomorphic?





Isomorphism of Graphs: Example 1

- Are the following two graphs isomorphic?

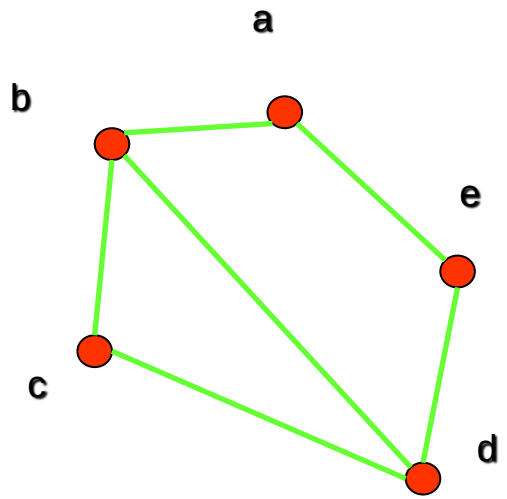


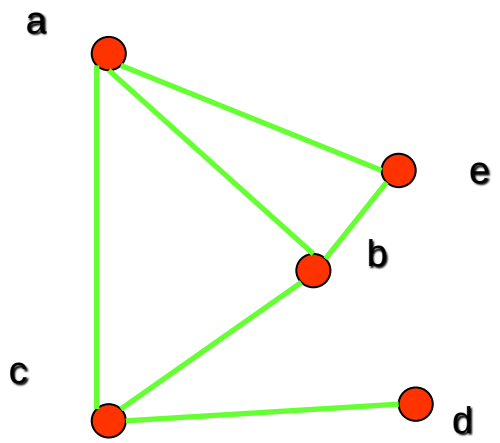
- **Solution**: Yes, they are isomorphic, because they can be arranged to look identical
- You can see this if in the right graph you move node b to the left of the edge {a, c}. Then the isomorphism f from the left to the right graph is

$$f(a) = e,\ f(b) = a,\ f(c) = b,\ f(d) = c,\ f(e) = d$$

Isomorphism of Graphs: Example 2

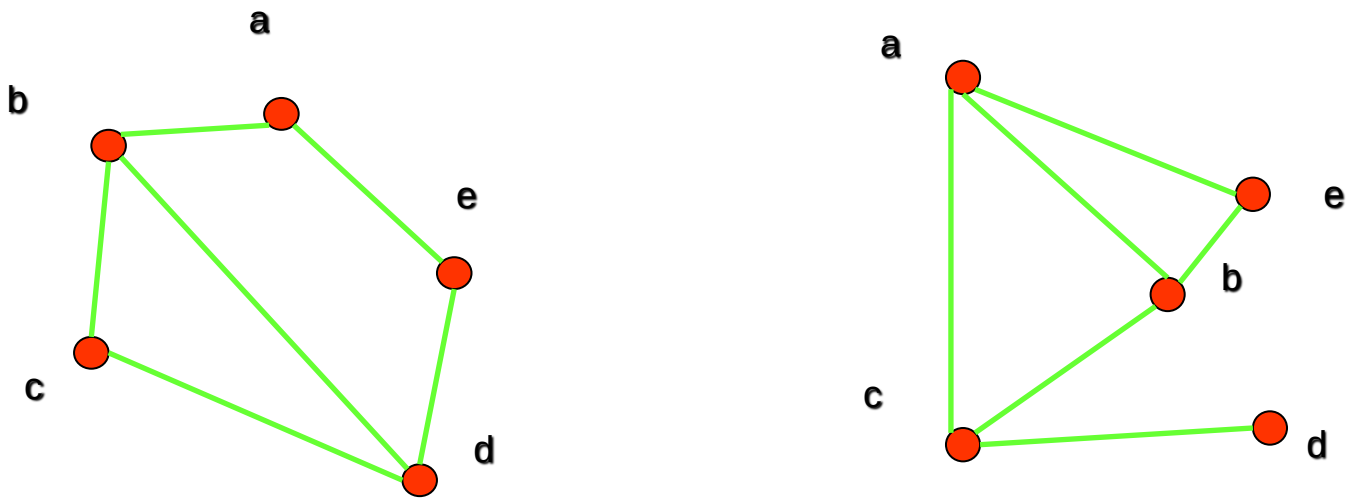
- Are the following two graphs isomorphic?





Isomorphism of Graphs: Example 2

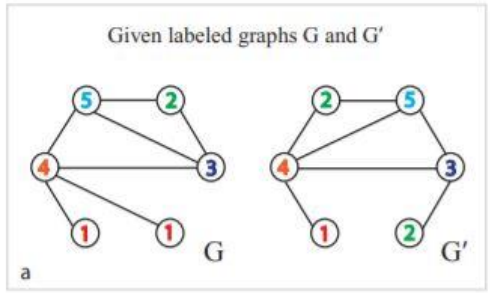
- Are the following two graphs isomorphic?

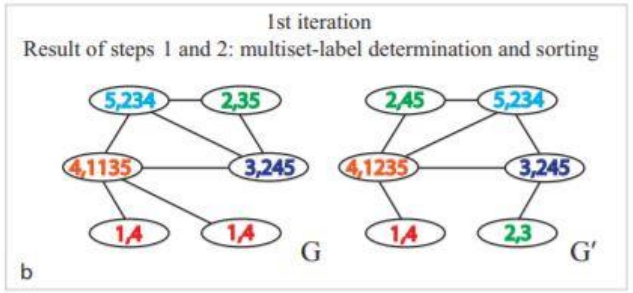


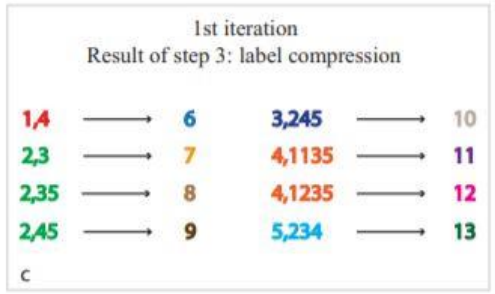
- **Solution**: No, they are not isomorphic, because they differ in the degrees of their nodes
- Vertex d in right graph is of degree one, but there is no such node in the left graph

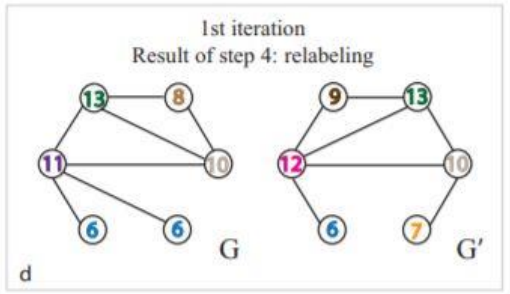
The Weisfeiler-Lehman Isomorphism Test

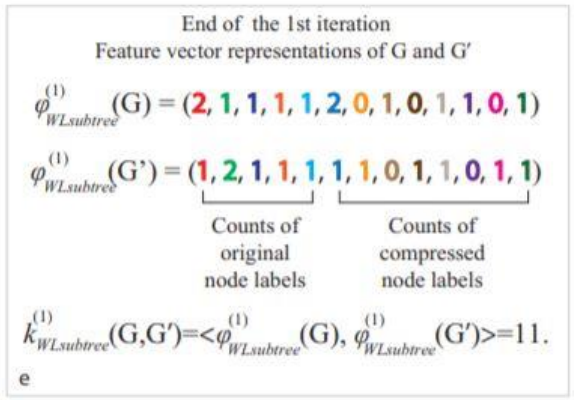
- Weisfeiler-Lehman Isomorphism Testing:











Algorithm 1: WL-1 algorithm (Weisfeiler & Lehmann, 1968)

Input: Initial node coloring $(h_1^{(0)}, h_2^{(0)}, ..., h_N^{(0)})$

Output: Final node coloring $(h_1^{(T)}, h_2^{(T)}, ..., h_N^{(T)})$

$t \leftarrow 0$;

repeat

  for $v_i \in \mathcal{V}$ **do**

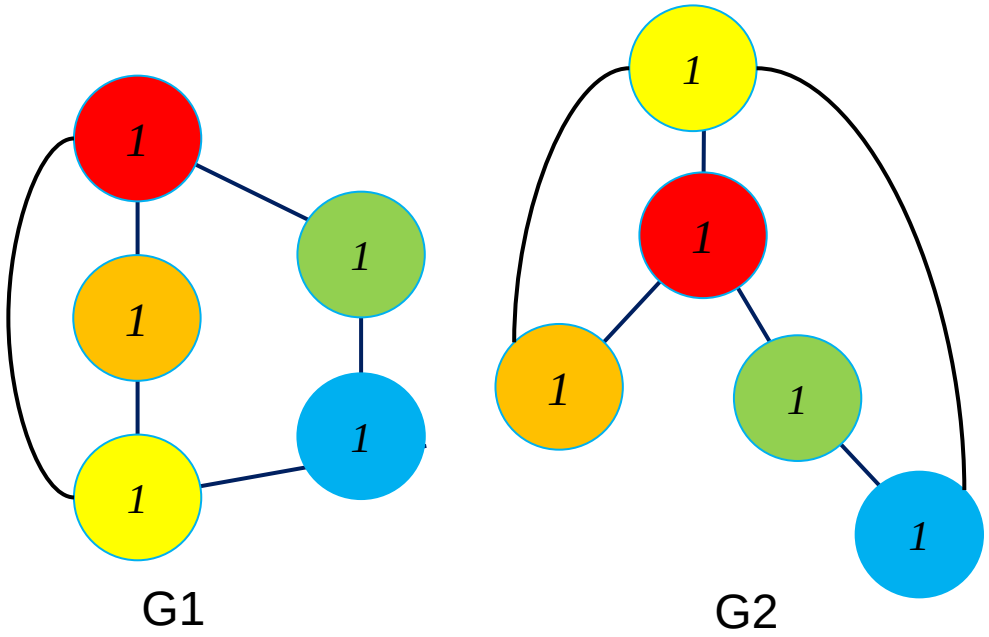
    $h_i^{(t+1)} \leftarrow \text{hash}\left(\sum_{j \in \mathcal{N}_i} h_j^{(t)}\right)$;

  $t \leftarrow t + 1$;

until *stable node coloring is reached*;

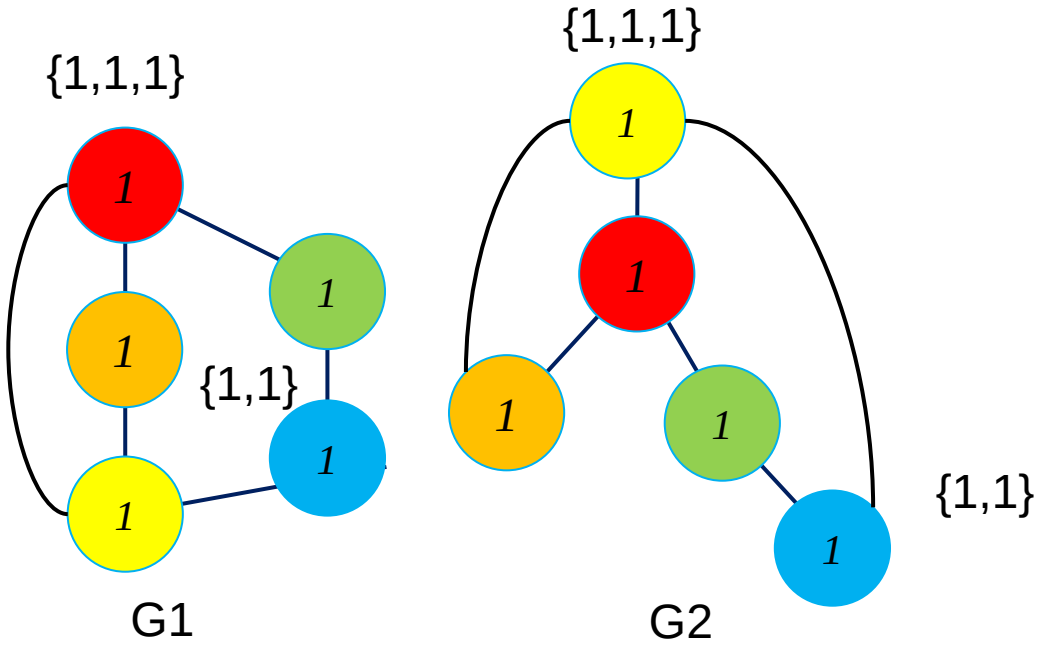
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes



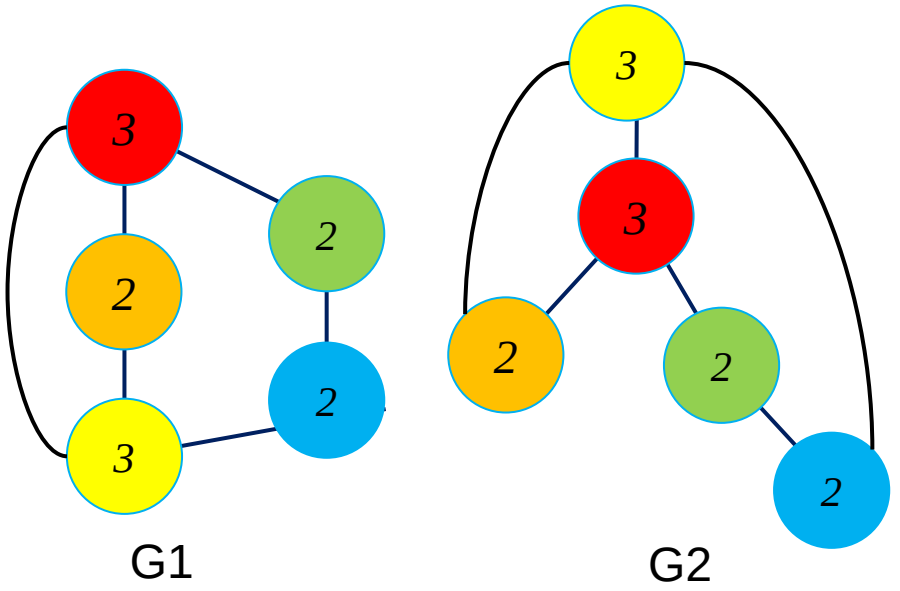
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels



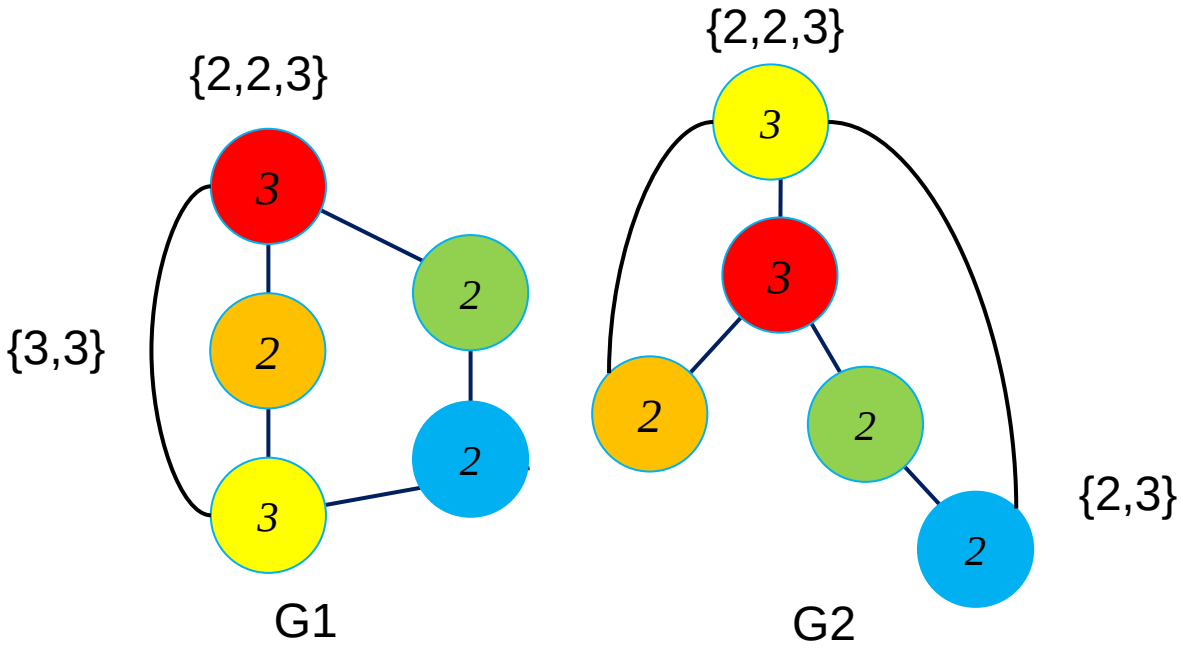
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels



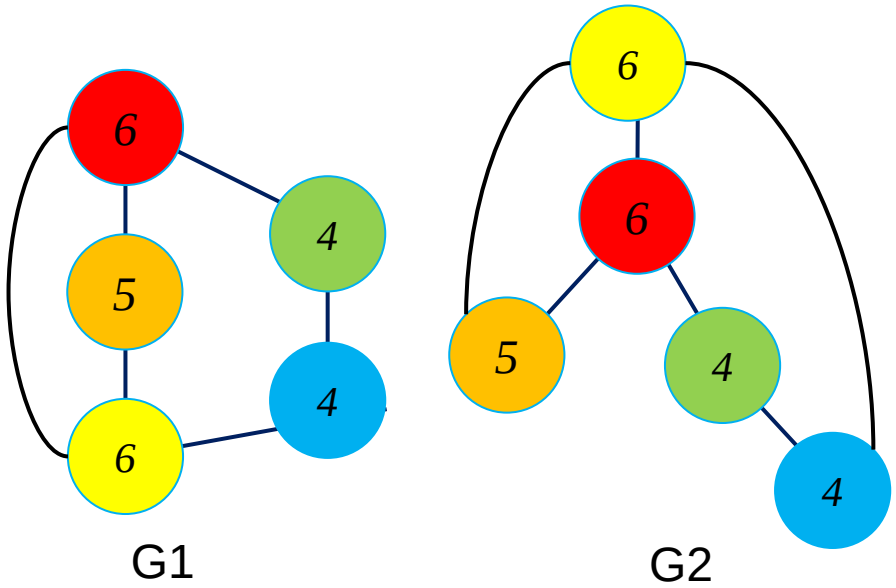
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels
- Step 3: Continuous



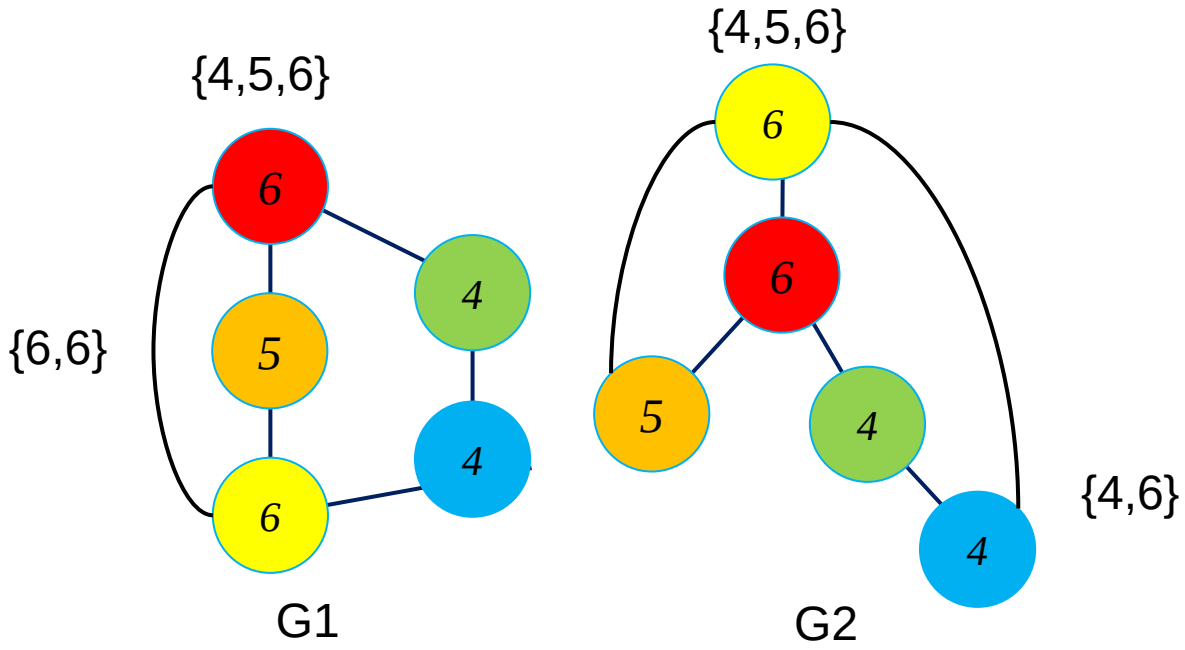
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous…



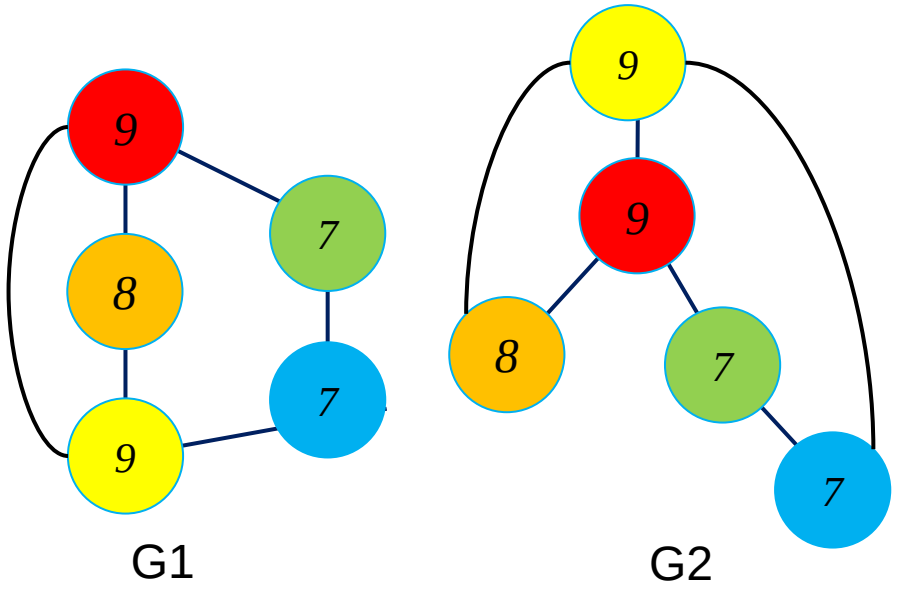
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous…



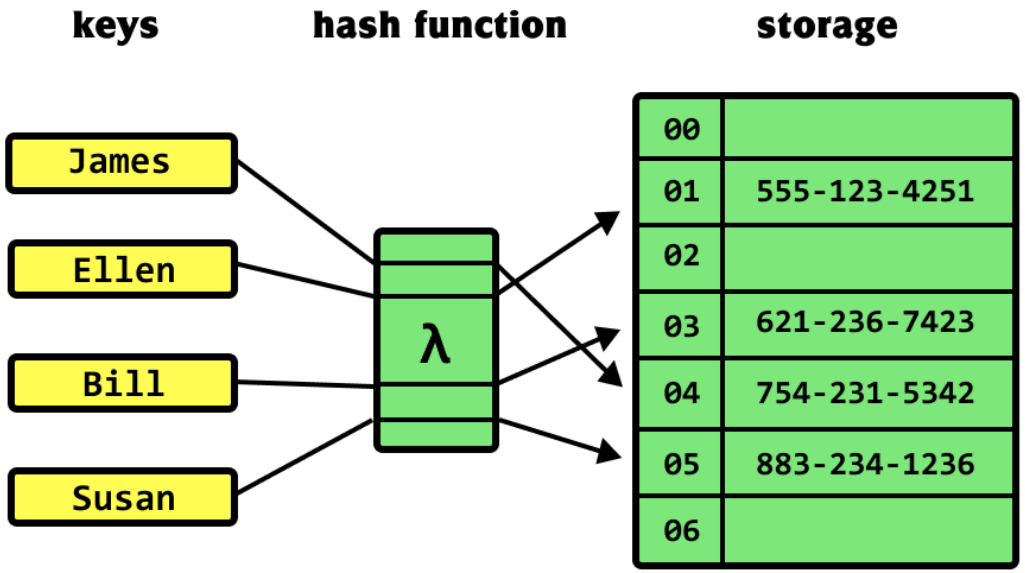
The Weisfeiler-Lehman Isomorphism Test

- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: Set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous…
- Step 4: Since the partition of nodes by compressed label has not changed, we terminate the algorithm here



Hash Function

- Converts a given numeric or alphanumeric key to a small practical integer value
- Hash function $h: (\{0,1\}^k \rightarrow \{0,1\}^{t(k)}$
  - Compresses
- Main goal: a little bit of difference in inputs will cause a big difference in outputs



Subgraph Matching Kernel

- Counts the number of matchings between subgraphs of bounded size in two graphs.
  - Can apply to graphs that contain node labels, edge labels, node attributes or edge attributes

- Given two graphs $G = (V, E)$ and $G' = (V', E')$
- Let $B(G, G')$ denote the set of all bijections between sets $S \subseteq V$ and $S' \subseteq V'$ $\lambda: B(G, G') \rightarrow \mathbb{R}^+$ be a weight function
- The subgraph matching kernel is

$$k(G, G') = \sum_{\phi \in B(G,G')} \lambda(\phi) \prod_{v \in S} \kappa_V(v, \phi(v)) \prod_{e \in S \times S} \kappa_E(e, \psi(e))$$

where $S = dom(\phi)$ and K_V, K_E are kernel functions defined on vertices and edges

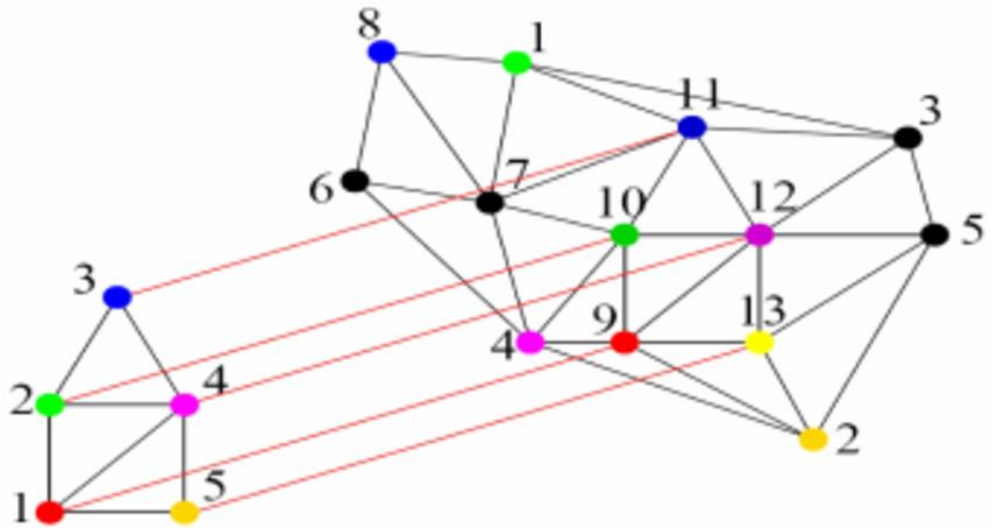
Subgraph Matching Kernel

- The common subgraph isomorphism kernel:

$$\kappa_V(v, v') = \begin{cases} 1, & \text{if } \ell(v) \equiv \ell(v'), \\ 0, & \text{otherwise and} \end{cases}$$

$$\kappa_E(e, e') = \begin{cases} 1, & \text{if } e \in E \wedge e' \in E' \wedge \ell(e) \equiv \ell(e') \text{ or } e \notin E \wedge e' \notin E', \\ 0, & \text{otherwise.} \end{cases}$$

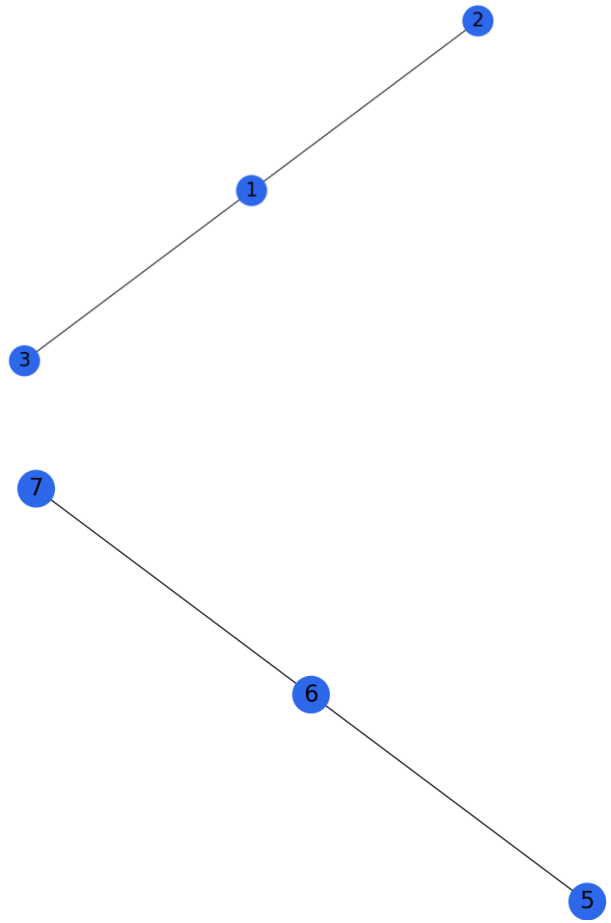
⇒ counts the number of isomorphic subgraphs contained in two graphs



Sample code: Graphlet kernels

- Find Graphlet kernels size 3:

```
import networkx as nx
import itertools
import matplotlib.pyplot as plt

g = nx.Graph()
g.add_edge(1,2)
g.add_edge(1,3)
g.add_edge(1,7)
g.add_edge(2,4)
g.add_edge(3,4)
g.add_edge(3,5)
g.add_edge(3,6)
g.add_edge(4,5)
g.add_edge(5,6)
g.add_edge(6,7)

target = nx.Graph()
target.add_edge(1,2)
target.add_edge(2,3)

colors = ['#DE7A6D', '#17BDF0',  '#3B88E9', '#212B83', '#EA43F3', '#EA3D11', '#C6A8C9', '#96FE21',
          '#C53A08', '#56B059', '#8EA7A2', '#BF9207', '#006843', '#2E68EA']
for sub_nodes in itertools.combinations(g.nodes(),len(target.nodes())):
    subg = g.subgraph(sub_nodes)
    if nx.is_connected(subg) and nx.is_isomorphic(subg, target):
        print(subg.edges())
        visualize_subgraph(subg, subg.nodes(), colors[i])
```



Shortest Path kernels

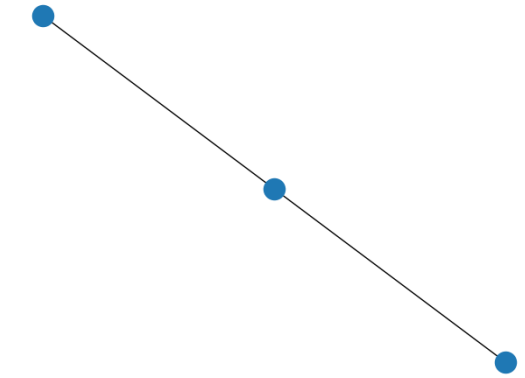
Shortest Path

Consider 2 molecules H20 and H30, an ion of water produced by protonation. Compute the Shortest Path Kernel.

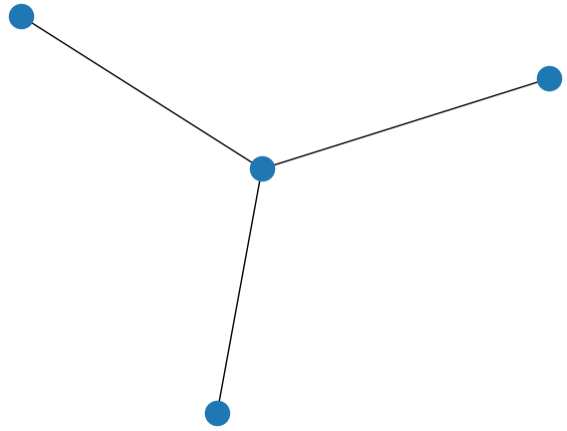
The number are proximity same to each other, meaning that 2 molecules has same graph kernel.

```
from grakel import Graph
from grakel.kernels import ShortestPath
import numpy as np

H2O_adjacency = [[0, 1, 1], [1, 0, 0], [1, 0, 0]]
H2O_node_labels = {0: 'O', 1: 'H', 2: 'H'}
H2O = Graph(initialization_object=H2O_adjacency, node_labels=H2O_node_labels)

H3O_adjacency = [[0, 1, 1, 1], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
H3O_node_labels = {0: 'O', 1: 'H', 2: 'H', 3:'H'}
H3O = Graph(initialization_object=H3O_adjacency, node_labels=H3O_node_labels)

sp_kernel = ShortestPath(normalize=True)
g1_sp = sp_kernel.fit_transform([H2O])
g2_sp = sp_kernel.transform([H3O])
print(g1_sp,g2_sp)

H2O_adjacency = np.matrix([[0, 1, 1], [1, 0, 0], [1, 0, 0]])
H3O_adjacency = np.matrix([[0, 1, 1, 1], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]])
G_h20=nx.from_numpy_array(H2O_adjacency)
nx.draw(G_h20)
```

```
[[1.]] [[0.94280904]]
```



H_2O



H_3O

Random Walk kernels

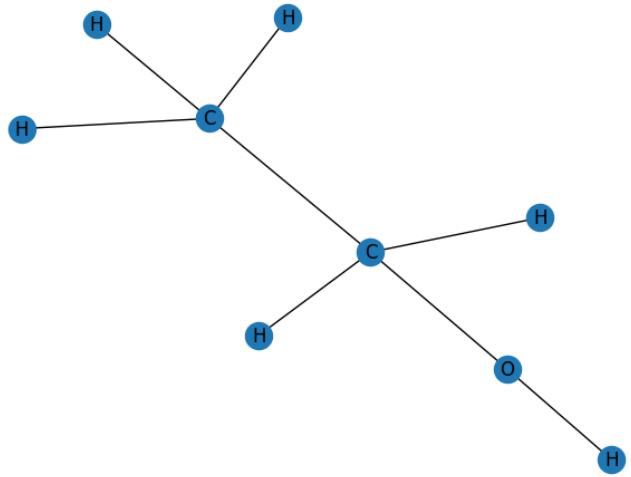
Random Walk

Consider 2 molecules Dimethyl_Ether and Ethanol. Compute the Random Walk Kernel.

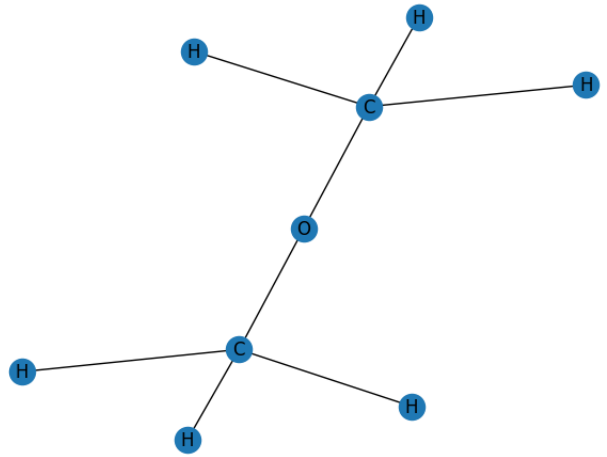
The number are proximity same to each other, meaning that 2 molecules has same graph kernel.

```
from grakel import Graph
from grakel.kernels import RandomWalk
import numpy as np

Ethanol_adjacency = [[0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [0, 0, 0, 0, 0, 0, 0, 1, 0],
                     [0, 0, 0, 0, 0, 0, 0, 0, 1],
                     [0, 0, 0, 0, 0, 0, 0, 1, 0],
                     [0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [1, 1, 0, 0, 0, 1, 0, 1, 0],
                     [0, 0, 1, 0, 1, 0, 1, 0, 1],
                     [0, 0, 0, 1, 0, 0, 0, 1, 0],
                    ]
Ethanol_node_labels = {0: 'H', 1: 'H', 2: 'H', 3:'H', 4:'H', 5:'H', 6:'C', 7:'C', 8 :'O'}

Ethanol = Graph(initialization_object=Ethanol_adjacency, node_labels=Ethanol_node_labels)

Dimethyl_Ether_adjacency = [[0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [0, 0, 0, 0, 0, 0, 0, 0, 1],
                     [0, 0, 0, 0, 0, 0, 0, 0, 1],
                     [0, 0, 0, 0, 0, 0, 0, 0, 1],
                     [0, 0, 0, 0, 0, 0, 1, 0, 0],
                     [1, 1, 0, 0, 0, 1, 0, 1, 0],
                     [0, 0, 0, 0, 0, 0, 1, 0, 1],
                     [0, 0, 1, 1, 1, 0, 0, 1, 0],
                    ]
Dimethyl_Ether_node_labels = {0: 'H', 1: 'H', 2: 'H', 3:'H', 4:'H', 5:'H', 6:'C', 7:'O', 8 :'C'}
Dimethyl_Ether = Graph(initialization_object=Dimethyl_Ether_adjacency, node_labels=Dimethyl_Ether_node_labels)

rw_kernel = RandomWalk(normalize=True)
g1_rw = rw_kernel.fit_transform([Ethanol])
g2_rw = rw_kernel.transform([Dimethyl_Ether])
print(g1_rw,g2_rw)
```

```
[[1.]] [[0.99860269]]
```



Ethanol (C_2H_5OH)



Dimethyl Ether (CH_3OCH_3)

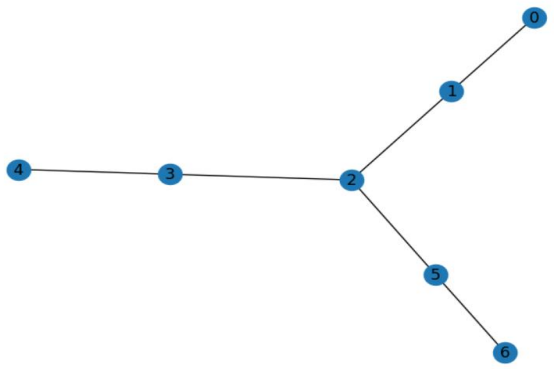
Graph isomorphism testing in NetworkX

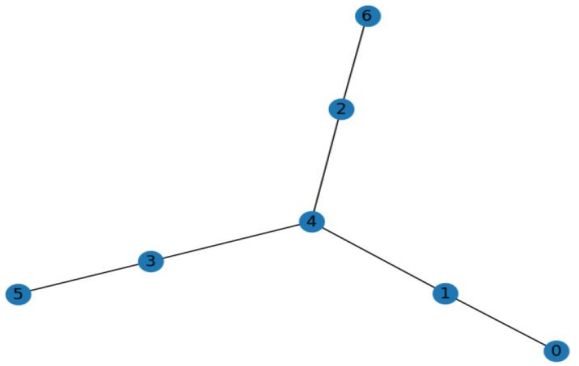
- Check isomorphism mapping from undirected graph G1 to undirected graph G2

```
import networkx as nx
import numpy as np
from networkx.algorithms import isomorphism
```

```
# Generate 2 graphs
G1 = nx.Graph()
G2 = nx.Graph()

G1.add_nodes_from(range(0,7))
G2.add_nodes_from(range(0,7))
G1.add_edges_from([(0,1), (1,2), (2,3), (3,4), (2,5), (5,6)])
G2.add_edges_from([(0,1), (1,4), (2,4), (2,6), (4,3), (3,5)])

# Mapping 2 graph G1 and G2
GM = isomorphism.GraphMatcher(G2,G1)
print(f" Checking graph isomorphic: {GM.is_isomorphic()}")
GM.mapping
```

```
 Checking graph isomorphic: True

{0: 0, 1: 1, 4: 2, 2: 3, 6: 4, 3: 5, 5: 6}
```





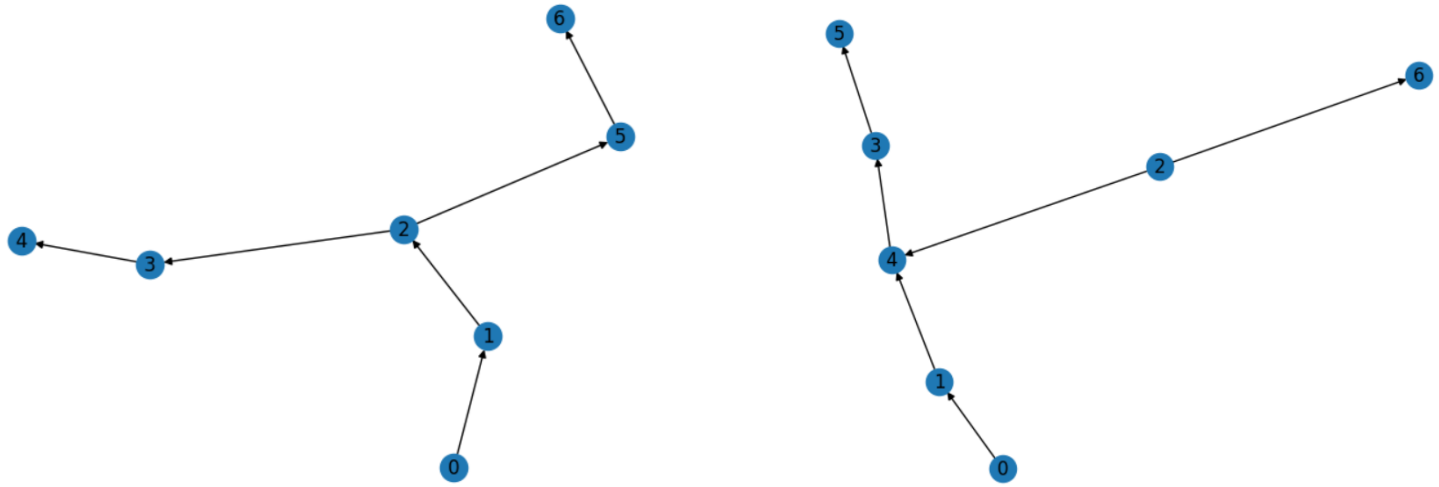
Graph isomorphism testing in NetworkX

- In directed graph:

```
import networkx as nx
import numpy as np
from networkx.algorithms import isomorphism
```

```
# Generate 2 graphs
G1 = nx.DiGraph()
G2 = nx.DiGraph()

G1.add_nodes_from(range(0,7))
G2.add_nodes_from(range(0,7))
G1.add_edges_from([(0,1), (1,2), (2,3), (3,4), (2,5), (5,6)])
G2.add_edges_from([(0,1), (1,4), (2,4), (2,6), (4,3), (3,5)])

# Mapping 2 graph G1 and G2
GM = isomorphism.GraphMatcher(G2,G1)
print(f" Checking graph isomorphic: {GM.is_isomorphic()}")
GM.mapping
```

```
 Checking graph isomorphic: False

{}
```



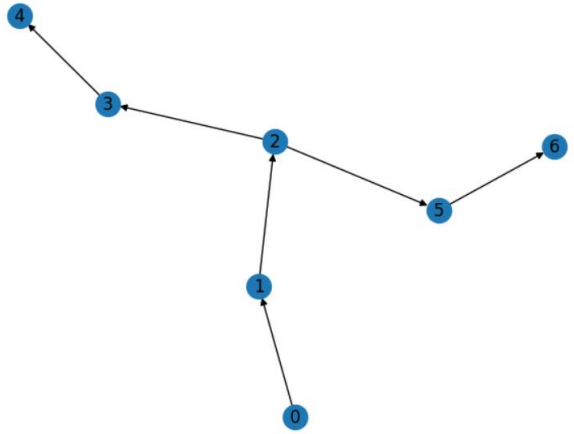
Graph isomorphism testing in NetworkX

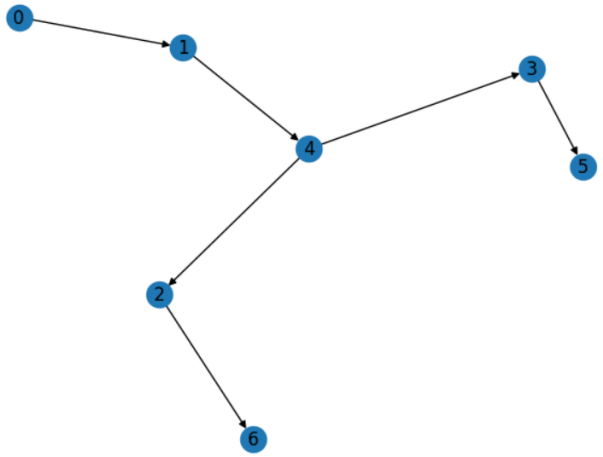
- Another example for directed graph

```
import networkx as nx
import numpy as np
from networkx.algorithms import isomorphism
```

```
# Generate 2 directed graphs
G1 = nx.DiGraph()
G2 = nx.DiGraph()

G1.add_nodes_from(range(0,7))
G2.add_nodes_from(range(0,7))
G1.add_edges_from([(0,1), (1,2), (2,3), (3,4), (2,5), (5,6)])
G2.add_edges_from([(0,1), (1,4), (4,2), (2,6), (4,3), (3,5)])

# Mapping 2 graph G1 and G2
GM = isomorphism.GraphMatcher(G2,G1)
print(f" Checking graph isomorphic: {GM.is_isomorphic()}")
GM.mapping
```

```
 Checking graph isomorphic: True

{0: 0, 1: 1, 4: 2, 2: 3, 6: 4, 3: 5, 5: 6}
```



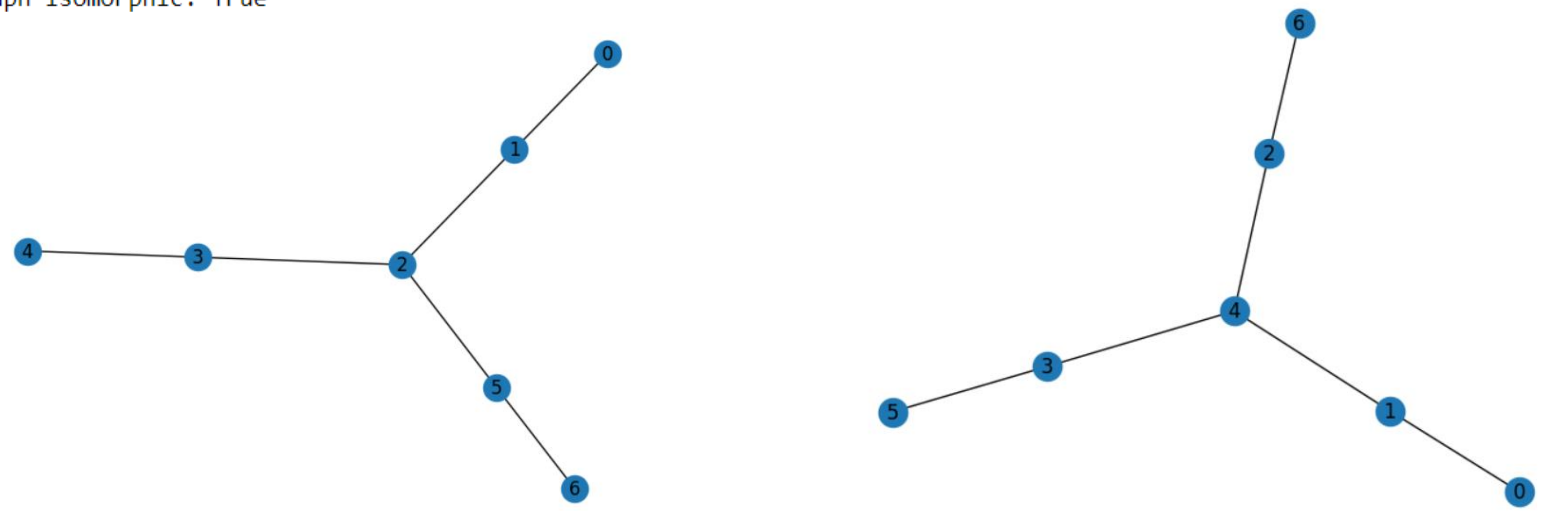


Graph kernels using WL relabelling process in NetworkX

- Weisfeiler Lehman (WL) – hash function: Undirected graph

```python
import networkx as nx

G1 = nx.Graph()
G2 = nx.Graph()

G1.add_nodes_from(range(0,7))
G2.add_nodes_from(range(0,7))
G1.add_edges_from([(0,1), (1,2), (2,3), (3,4), (2,5), (5,6)])
G2.add_edges_from([(0,1), (1,4), (2,4), (2,6), (4,3), (3,5)])

g1_hash = nx.weisfeiler_lehman_graph_hash(G1)
g2_hash = nx.weisfeiler_lehman_graph_hash(G2)

# g1_hash and g2_hash are equal when they are isomorphic
print(f"First  graph hash: {g1_hash}")
print(f"Second graph hash: {g2_hash}")
print(f"Checking graph isomorphic: {g1_hash==g2_hash}")
```

```
First  graph hash: 61f645001e86ad8a32357cc828ae33cb
Second graph hash: 61f645001e86ad8a32357cc828ae33cb
Checking graph isomorphic: True
```

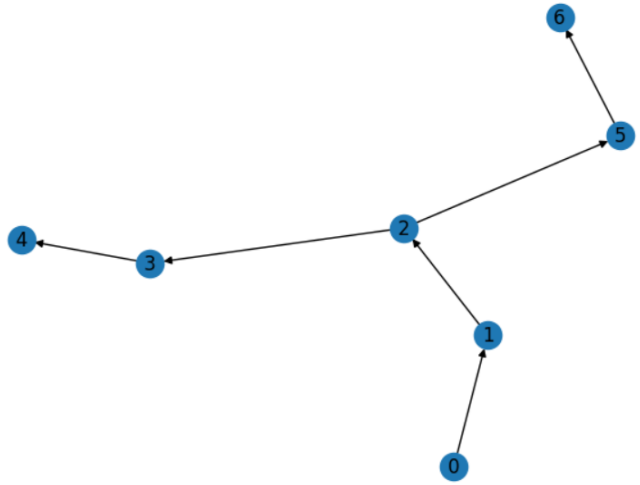


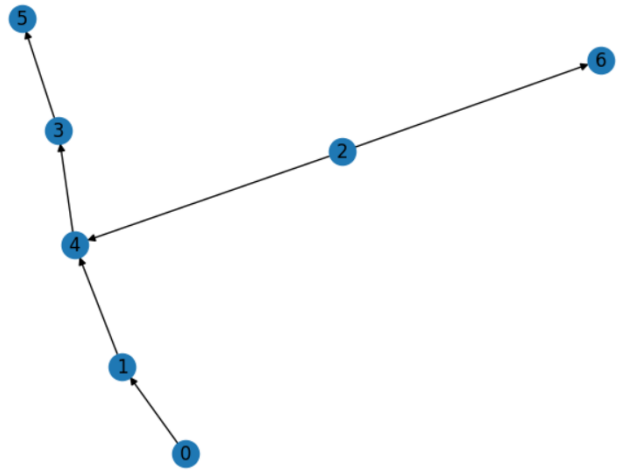
Graph kernels using WL relabelling process in NetworkX

- Weisfeiler Lehman (WL) – hash function : Directed graph

```
import networkx as nx

G1 = nx.DiGraph()
G2 = nx.DiGraph()

G1.add_nodes_from(range(0,7))
G2.add_nodes_from(range(0,7))
G1.add_edges_from([(0,1), (1,2), (2,3), (3,4), (2,5), (5,6)])
G2.add_edges_from([(0,1), (1,4), (2,4), (2,6), (4,3), (3,5)])

g1_hash = nx.weisfeiler_lehman_graph_hash(G1)
g2_hash = nx.weisfeiler_lehman_graph_hash(G2)

# g1_hash and g2_hash are equal when they are isomorphic
print(f"First  graph hash: {g1_hash}")
print(f"Second graph hash: {g2_hash}")
print(f"Checking graph isomorphic: {g1_hash==g2_hash}")
```

```
First  graph hash: 7d77c6474bd3835fe0f19ac0f27881e2
Second graph hash: 9dacf03794ba1624e5a8f373848e5ec5
Checking graph isomorphic: False
```





네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA