

Graph Kernels

Prof. O-Joun Lee

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

Content s



- Overview of graph kernels
- WL relabeling process
- Applications of graph kernels

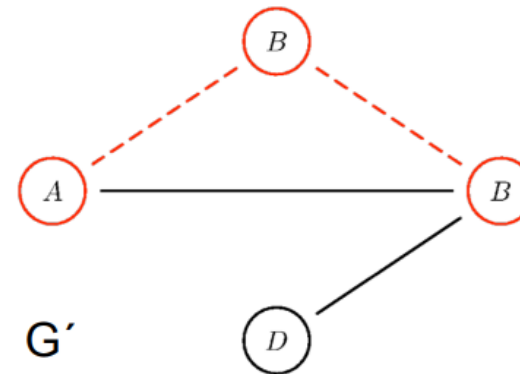
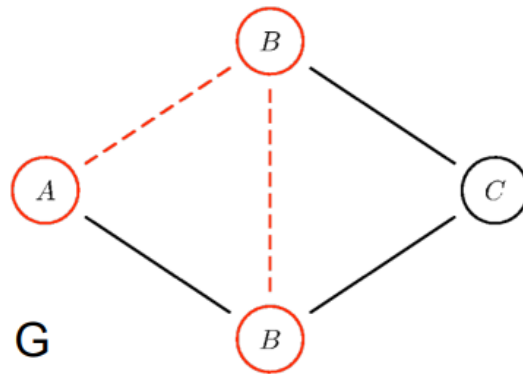
- From an algorithmic perspective, graphs are the most general data structures, as all common data types are simple instances of graphs.
 - E.g. 1, A time series of vectors can be represented as a graph that contains one node per time step, and consecutive steps are linked by an edge.
 - E.g. 2, A string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

- 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.

- 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'



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

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 iff $(f(x),f(y))$ 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-complete

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

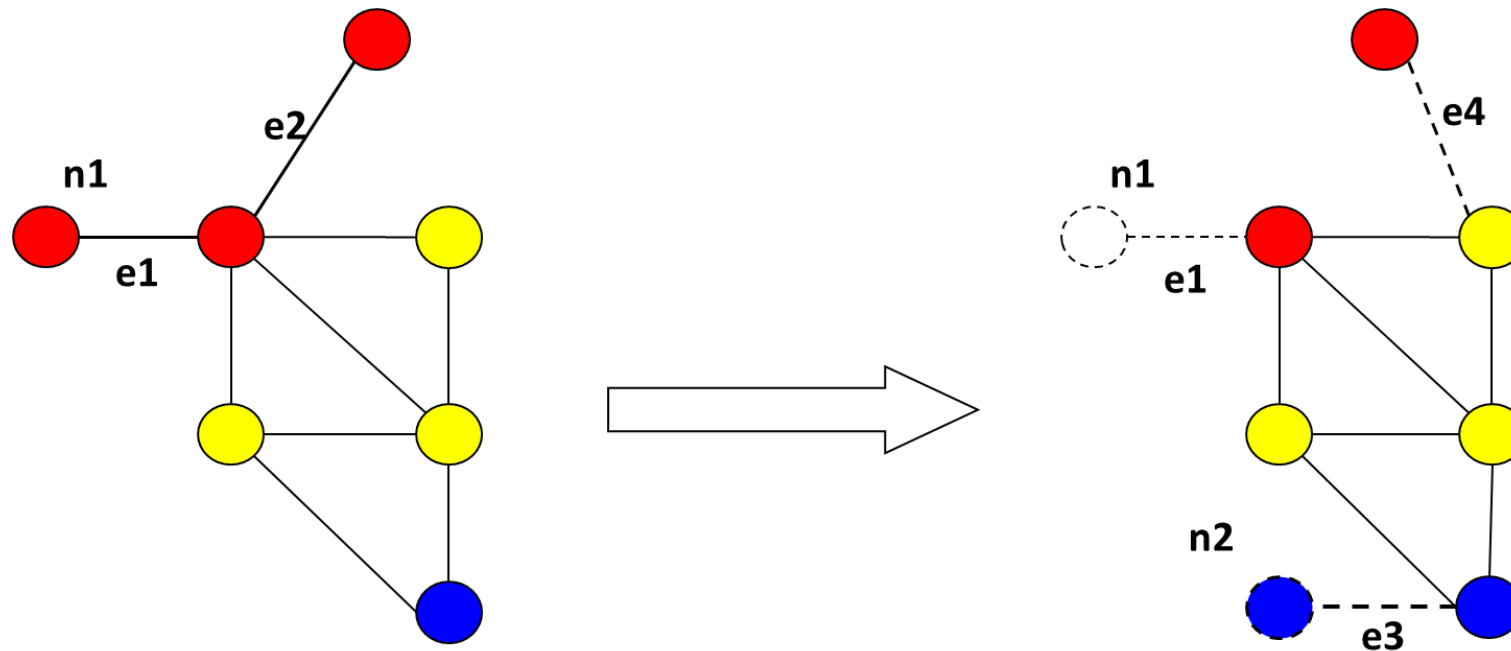
➤ **NP-completeness:**

- A decision problem C is NP-complete iff
- 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

- 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)

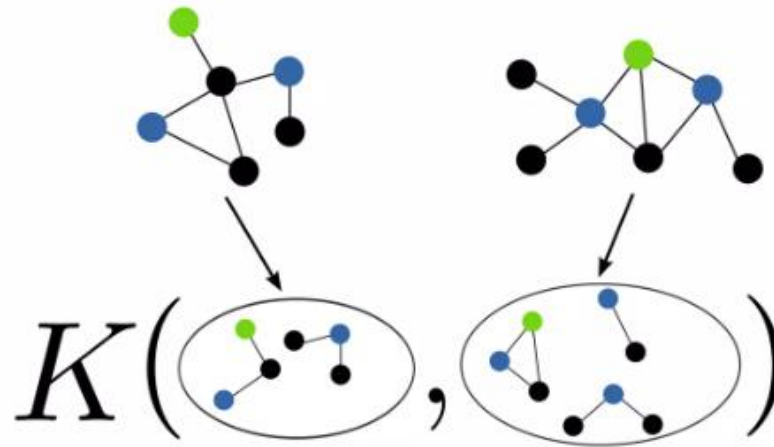


- Principle
 - Count operations that are necessary to transform G_1 into G_2
 - Assign costs to different types of operations (edge/node insertion/deletion, modification of labels)
- Advantages
 - Captures **partial similarities** between graphs
 - Allows for noise in the nodes, edges and their labels
 - 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

- Principle
 - Map each graph to a feature vector
 - Use distances and metrics on vectors for learning on graphs
- Advantages
 - Reuses known and efficient tools for feature vectors
- Disadvantages
 - Efficiency comes at a price: feature vector transformation leads to loss of topological information (or includes subgraph isomorphism as one step)

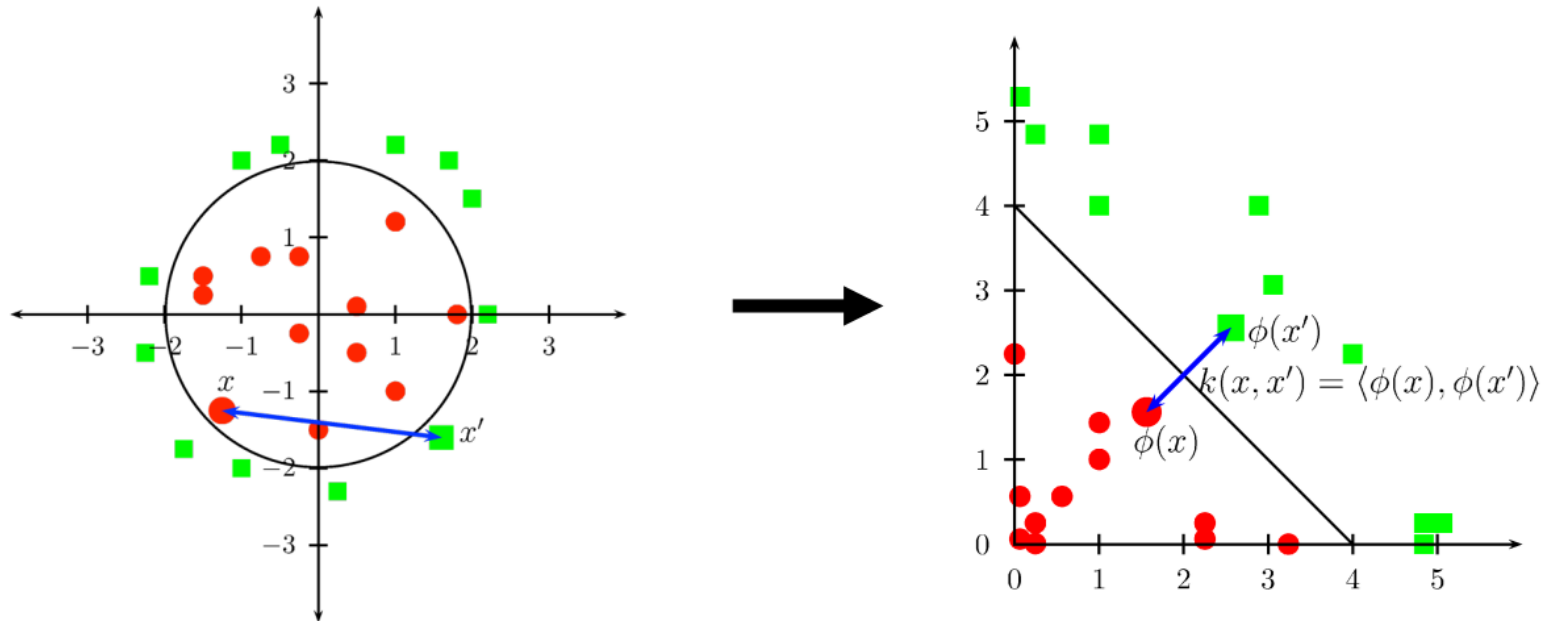
- We want:
 - Polynomial-time similarity measure for graphs
- Graph kernels
 - Compare substructures of graphs that are computable in polynomial time.
- Criteria for a good graph kernel:
 - Expressive
 - Efficient to compute
 - Positive definite
 - Applicable to wide range of graphs

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



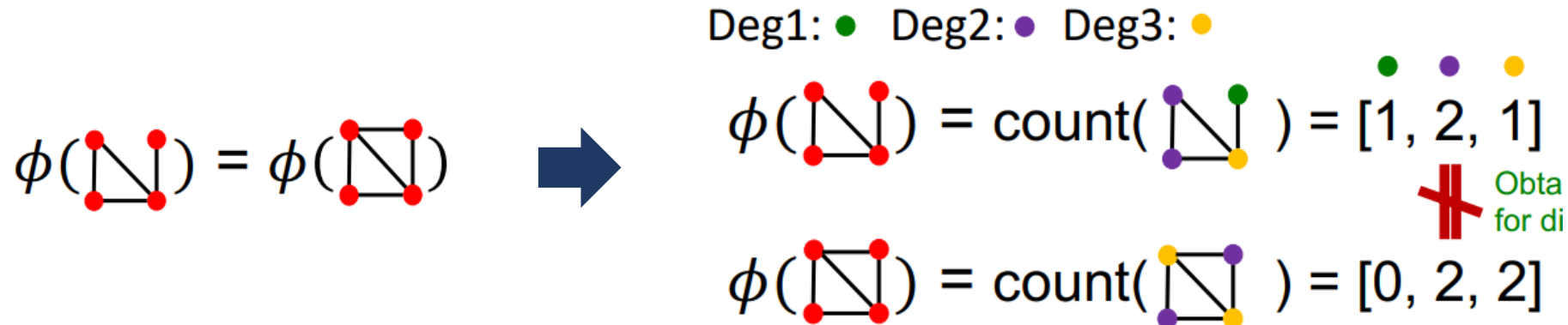
- Mapping two objects x and x' via mapping ϕ into feature space H
- Measure their similarity in H as $\langle \phi(x), \phi(x') \rangle$.
- **Kernel Trick**: Compute inner product in H as kernel in input space

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



- 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

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



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

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

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

- Link to graph isomorphism
 - Let $k(G, G') = \langle \phi(G), \phi(G') \rangle$ 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.**

- 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

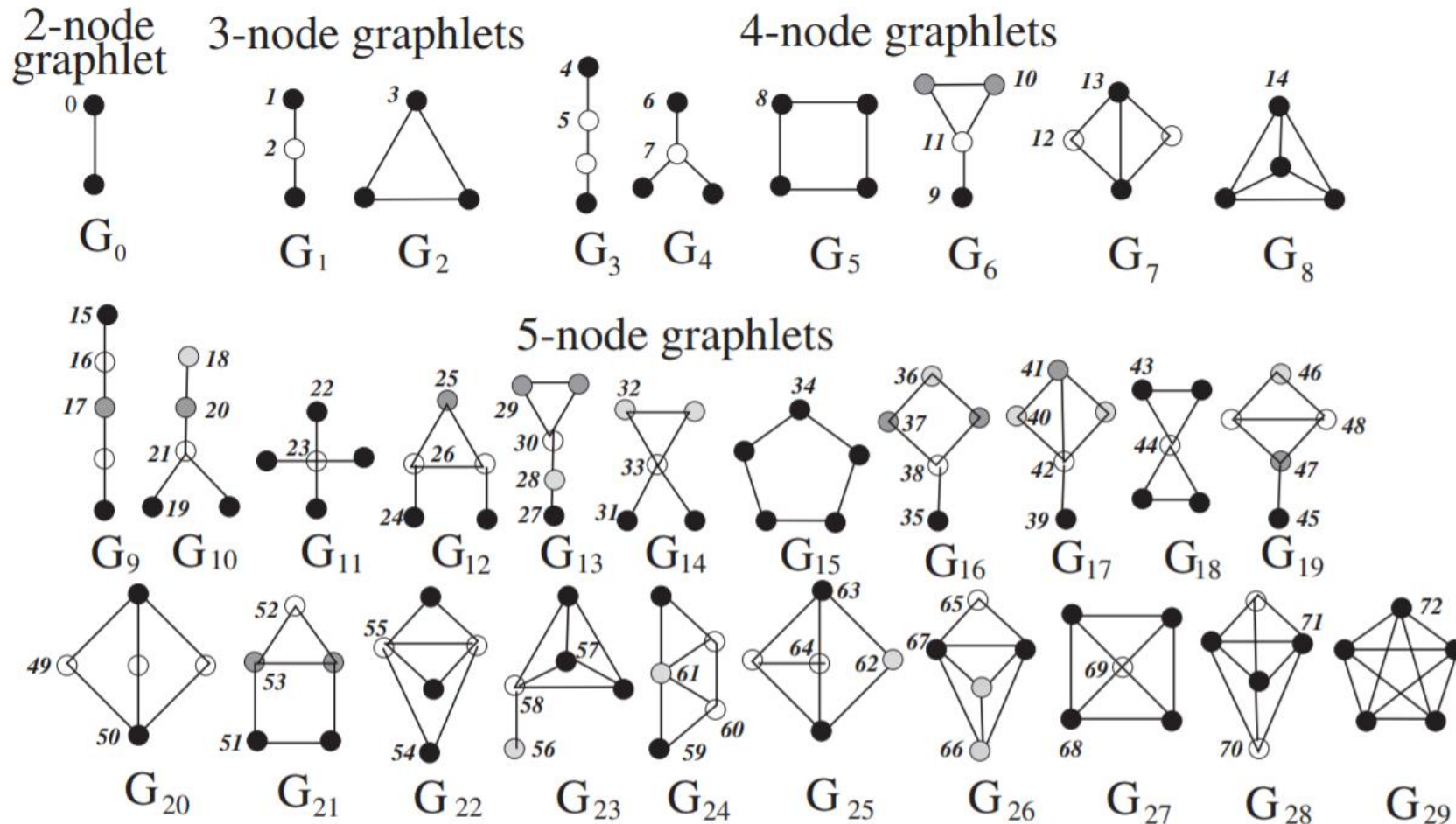
$$K(G_1, G_2) = K(X(G_1), X(G_2)) = |X(G_1) \cap X(G_2)|$$
- This kernel looks for **all overlapping substructures between two graphs**
- Complexity high

There are four graph kernels:

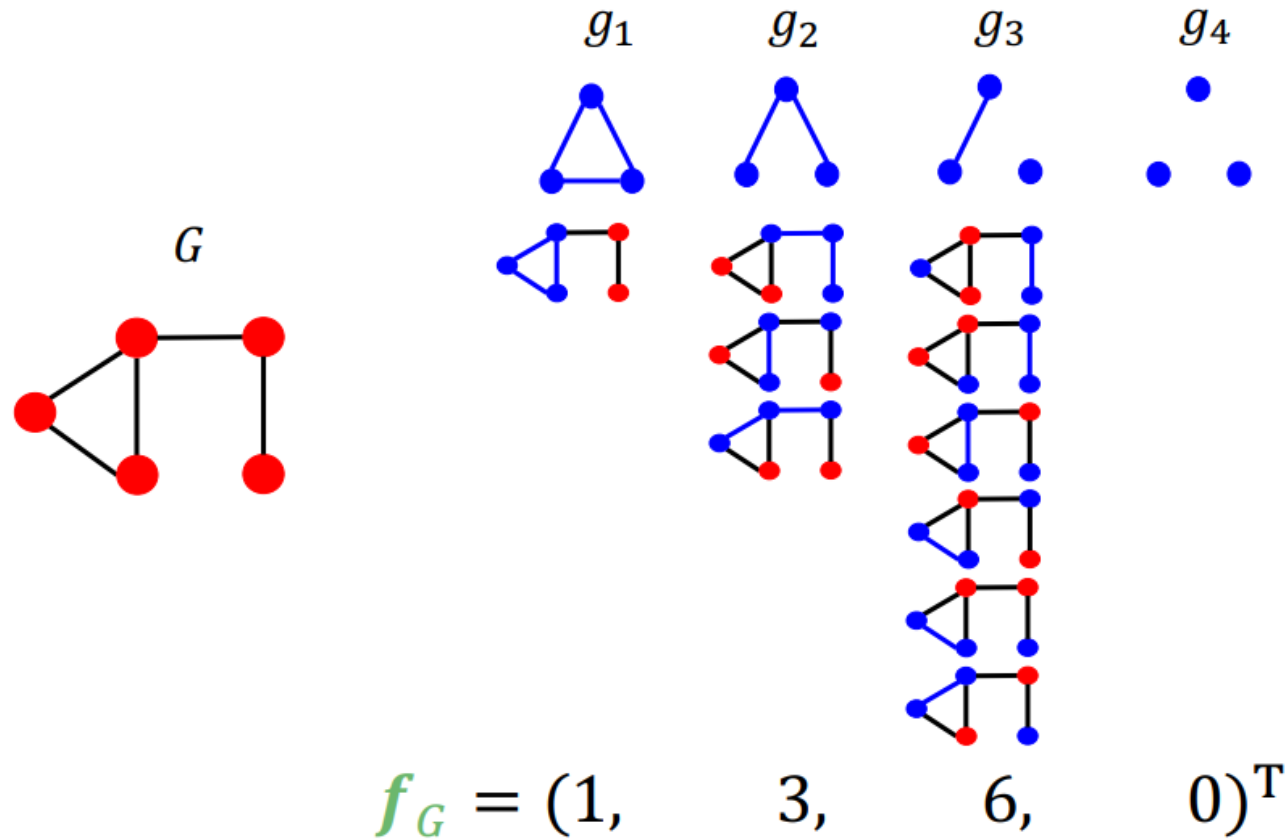
- **Graphlet kernel** [Shervashidze et al., 2009]:
 - **Counts identical pairs of graphlets**
(i.e., subgraphs with k nodes where $k = 3, 4, 5$) in two graphs.
- **Shortest path kernel** [Borgwardt and Kriegel, 2005]:
 - **Counts pairs of shortest paths** in two graphs
having the same source and sink labels and identical length.
- **Weisfeiler-Lehman** [Weisfeiler-Lehman isomorphic testing, 1968]:
 - WL relabeling 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.

- 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**

- Variety of subgraphs according to size k :



- Count subgraphs of limited size 3:



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

Definition (Shortest-path graph kernel):

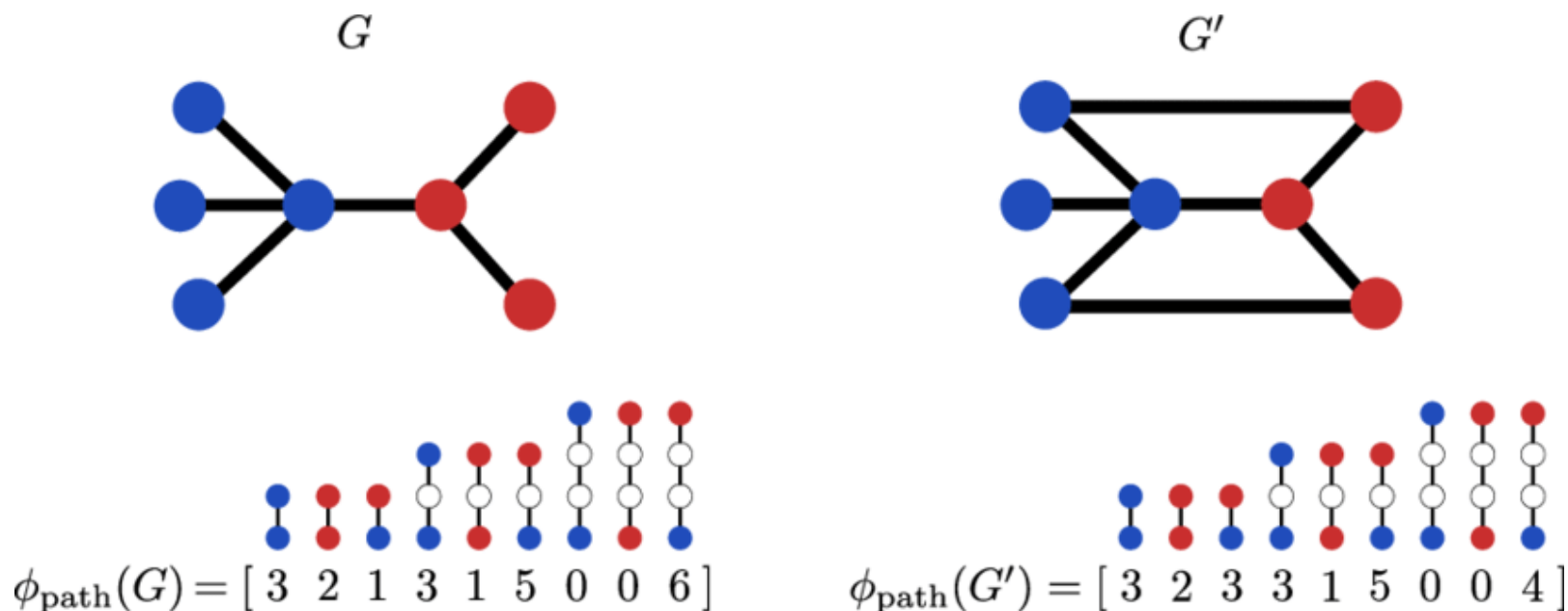
Let G_1 and G_2 be two graphs that are Floyd-transformed into S_1 and S_2 . We can then define shortest-path graph kernel on $S_1 = (G_1, E_1)$ and $S_2 = (G_2, E_2)$ as:

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

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

Shortest-path graph kernel:

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



- 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 :

$$k_X(G_1, G_2) = \sum_{i,j=1}^{|V_X|} \left[\sum_{n=0}^{\infty} \lambda_n A_X^n \right]_{ij},$$

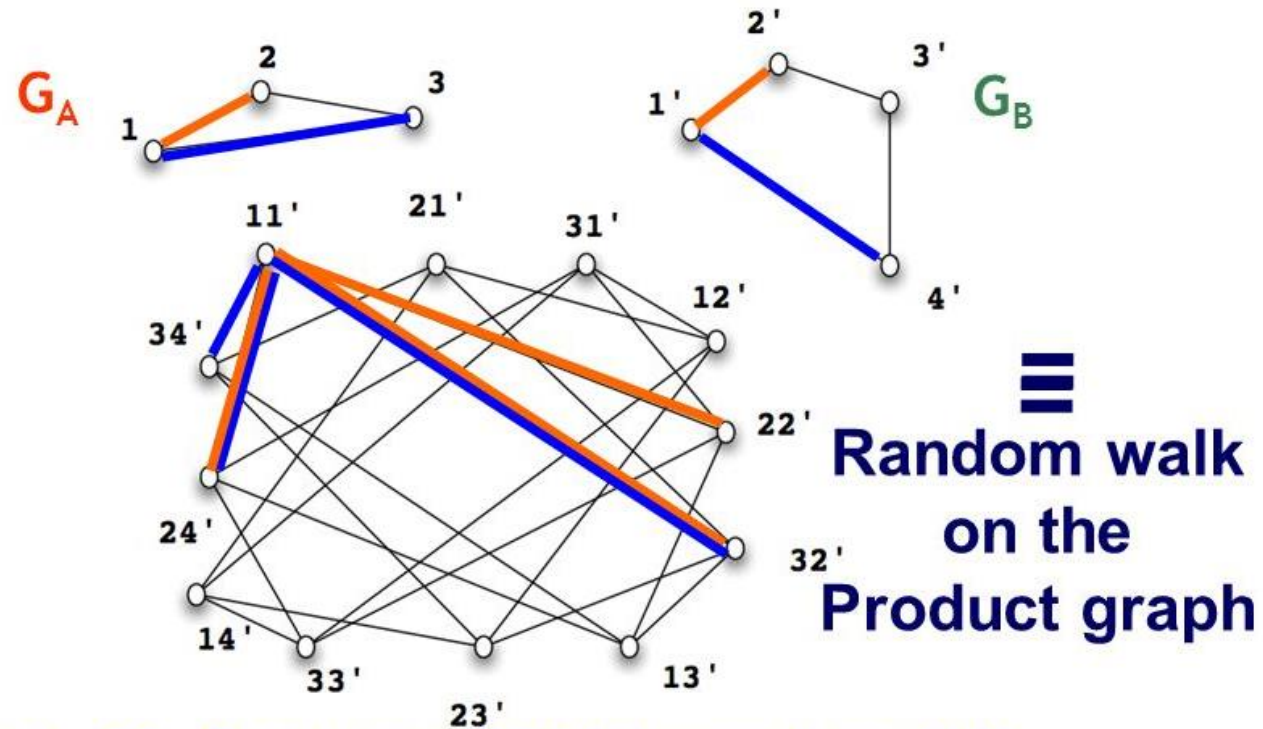
where A_X is the adjacency matrix of G_X , defined via:

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

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

λ_n must be chosen appropriately for k_X to converge

- 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]

SDM'14 Tutorial

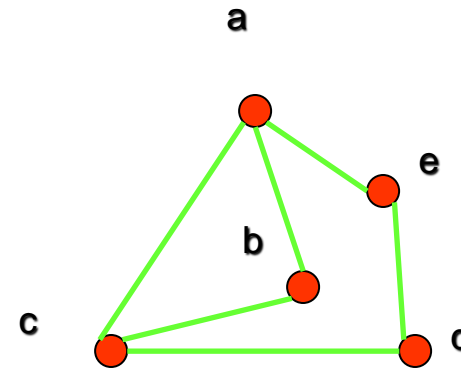
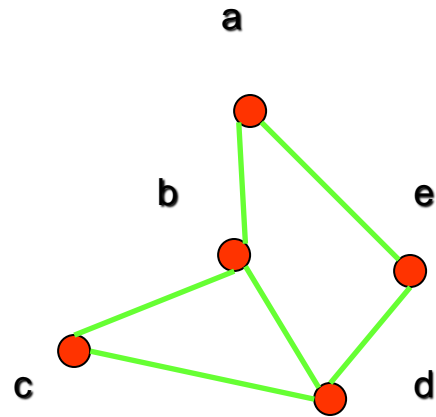
D. Koutra & T. Eliassi-Rad & C. Faloutsos

Definition:

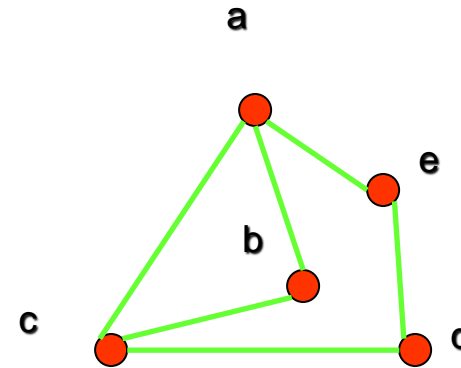
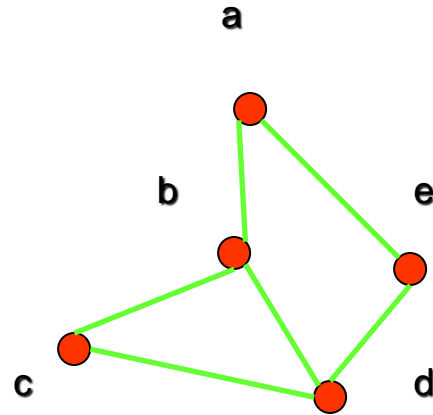
- The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a bijection (a one-to-one and onto function) f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2 , for all a and b in V_1 .
- Such a function 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.

- 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.

➤ Are the following two graphs isomorphic?

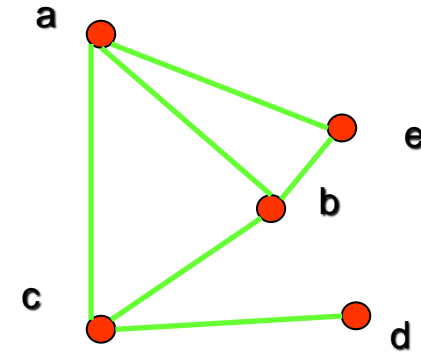
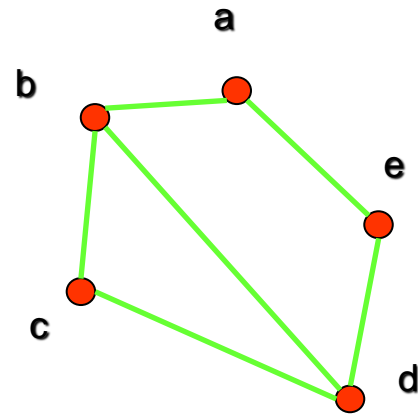


- Are the following two graphs isomorphic?

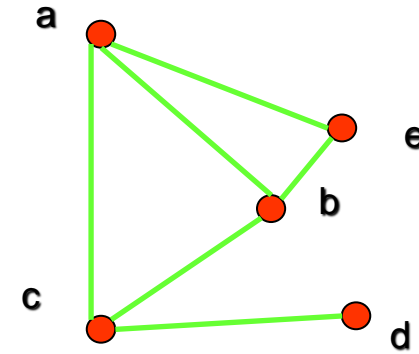
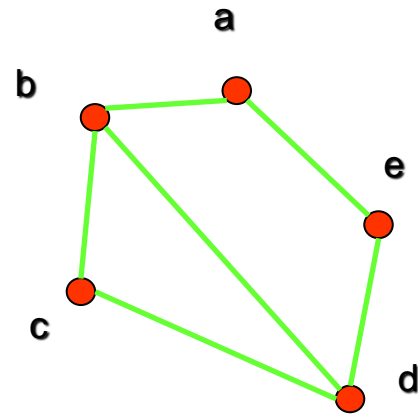


- Solution: Yes, they are isomorphic, because they can be arranged to look identical. You can see this if in the right graph you move vertex b to the left of the edge {a, c}. Then the isomorphism f from the left to the right graph is: $f(a) = e$, $f(b) = a$, $f(c) = b$, $f(d) = c$, $f(e) = d$.

➤ Are the following two graphs isomorphic?

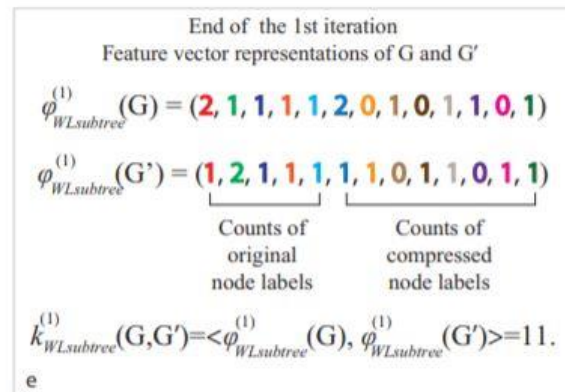
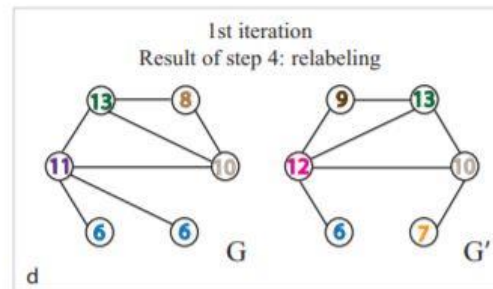
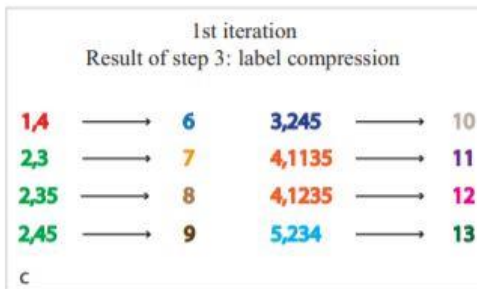
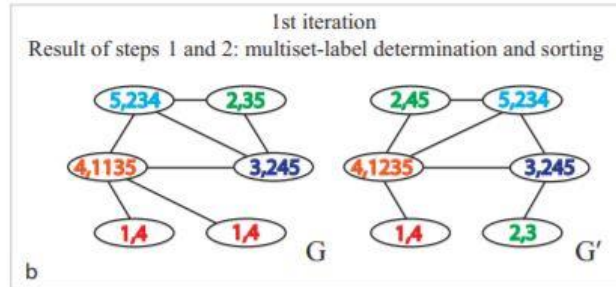
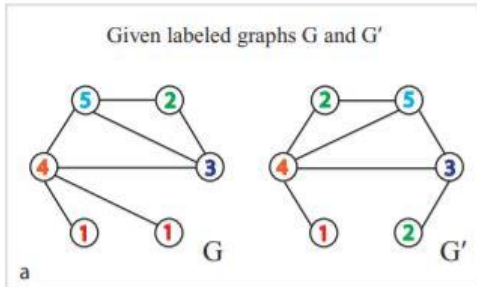


- Are the following two graphs isomorphic?



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

➤ Weisfeiler-Lehman Isomorphism Testing:



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

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

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

$t \leftarrow 0$;

repeat

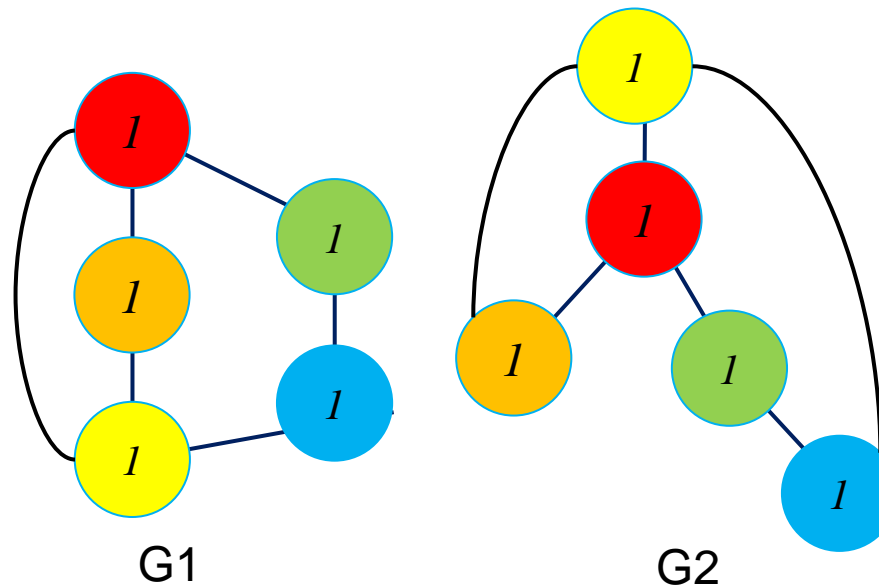
for $v_i \in \mathcal{V}$ **do**

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

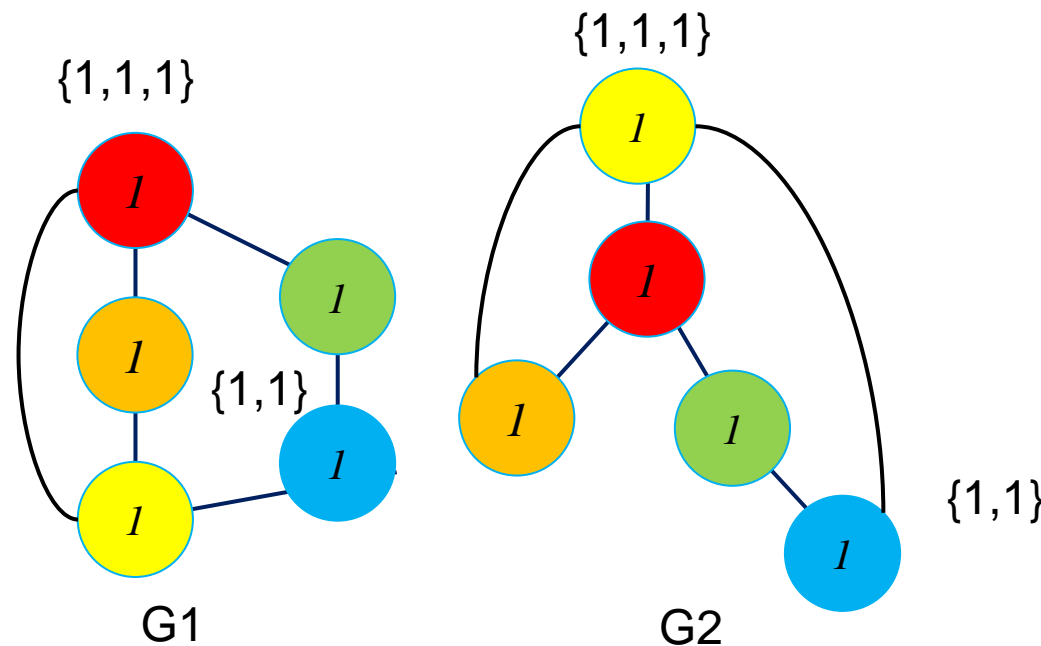
$t \leftarrow t + 1$;

until *stable node coloring is reached*;

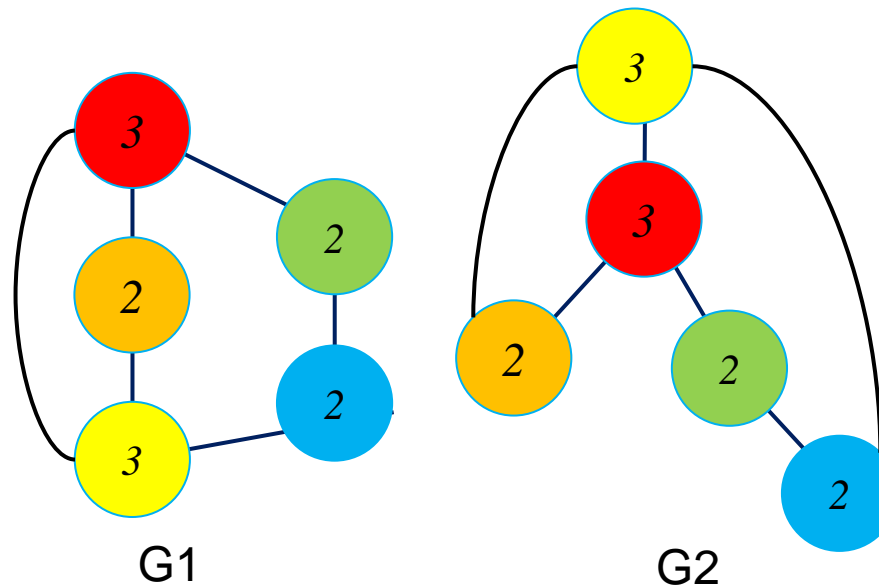
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label =1 for all nodes



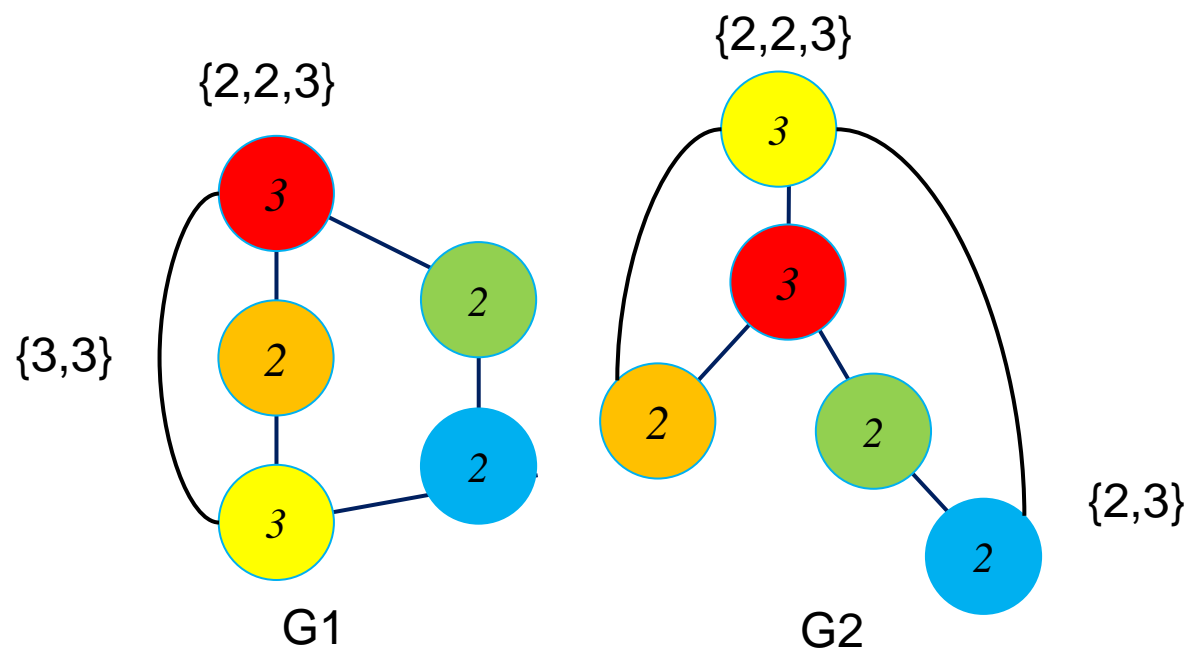
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label = 1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.



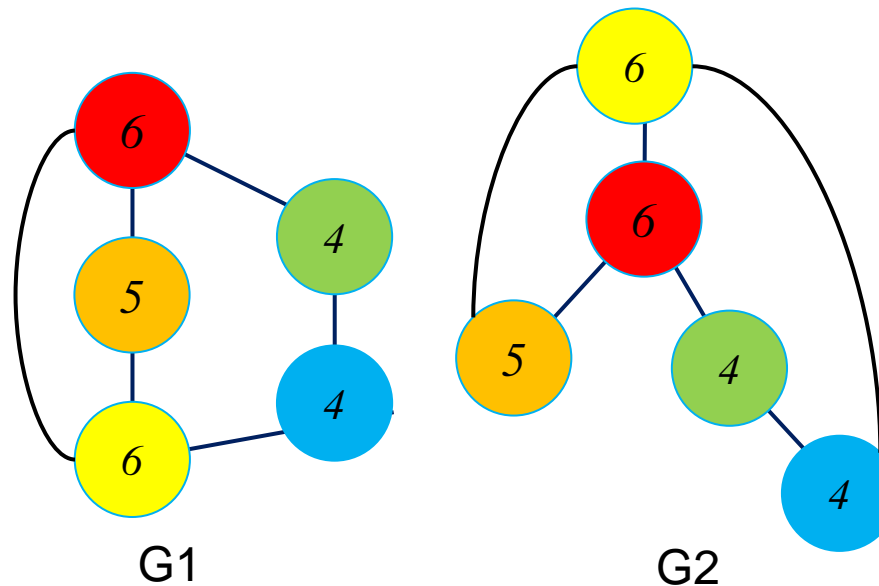
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.



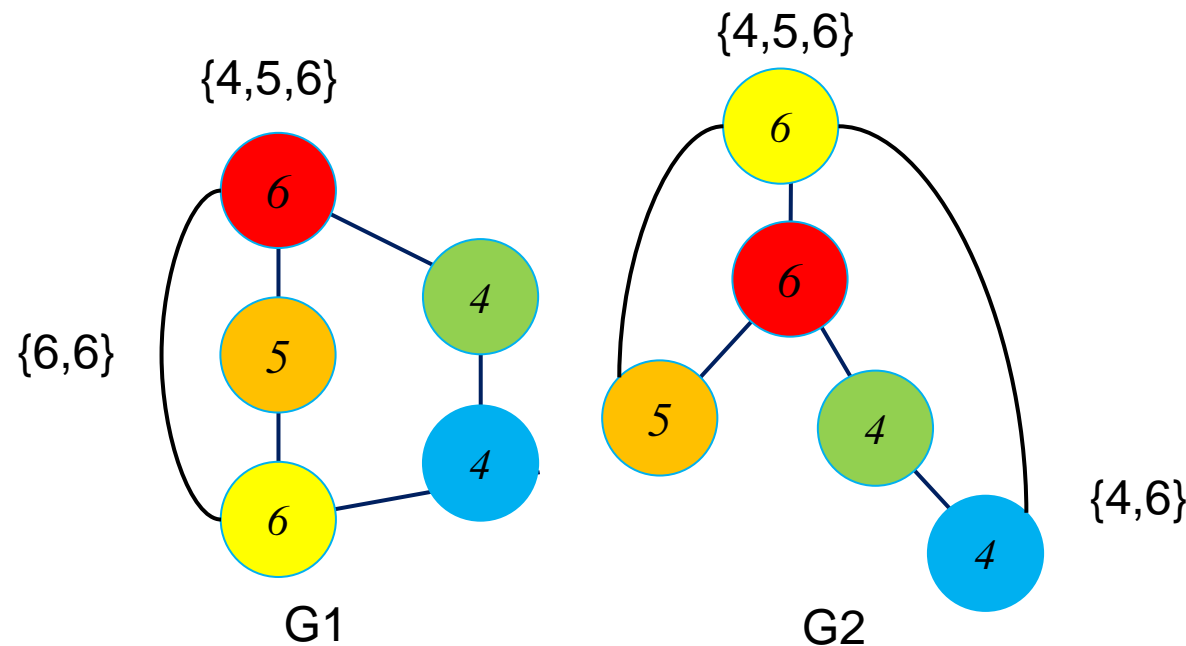
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous.



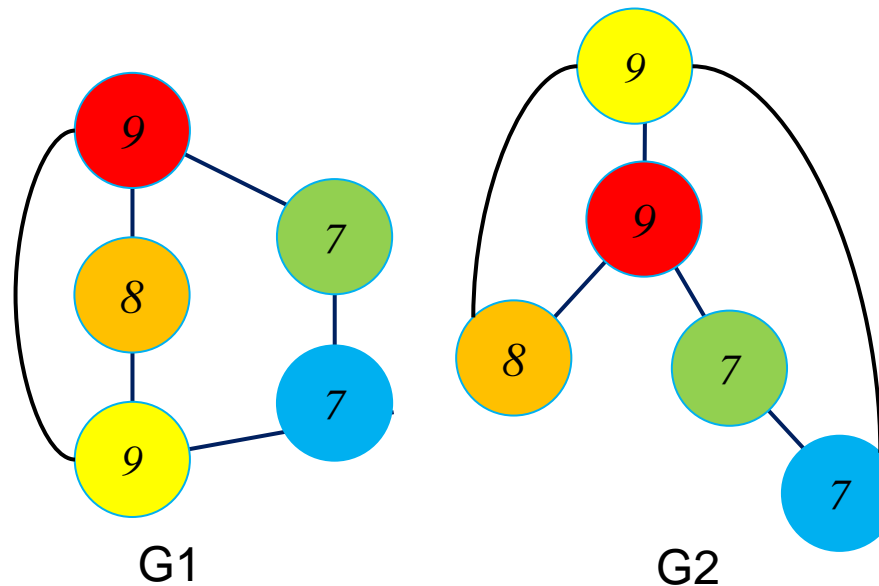
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous....



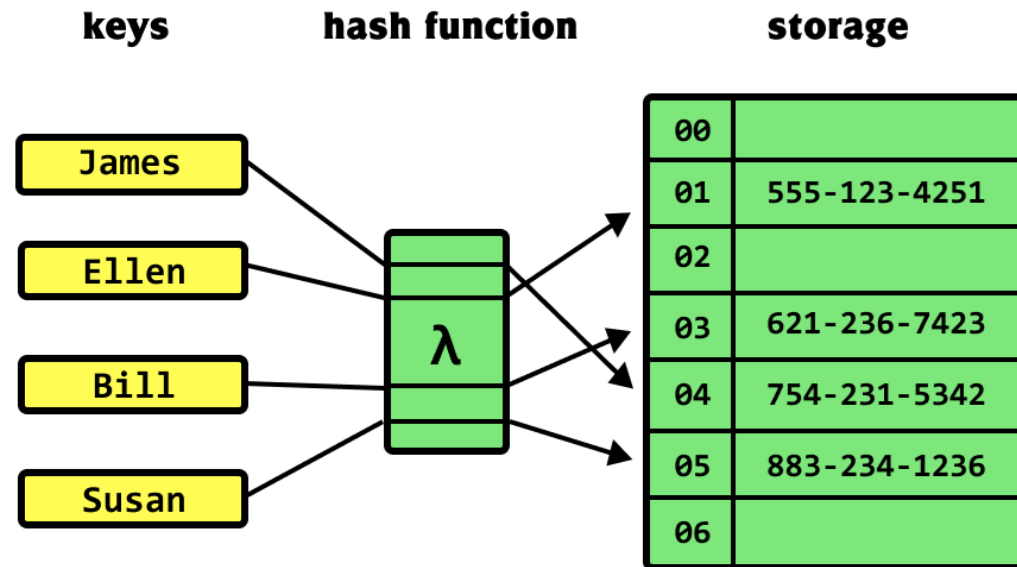
- We will apply the Weisfeiler-Lehman isomorphism test to these graphs as a means of illustrating the test.
- Step 1: set node label =1 for all nodes
- Step 2: Compute multiset of the neighboring nodes' compressed labels.
- Step 3: Continuous....



- 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 may terminate the algorithm here



- 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



- VF2 Algorithm: stores the isomorphism mapping from G1 to G2

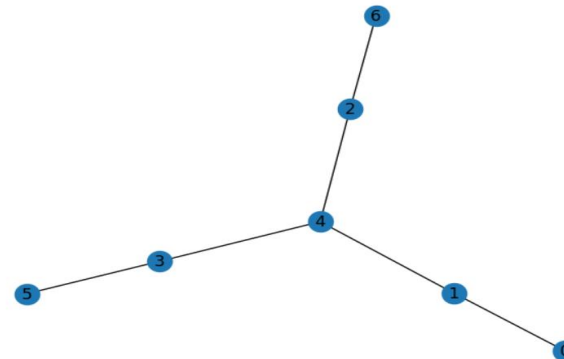
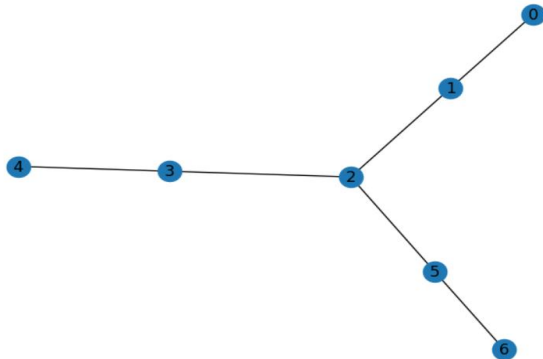
```
# Generate 2 graphs
G1 = nx.Graph()
G2 = nx.Graph()

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}



➤ In directed graph:

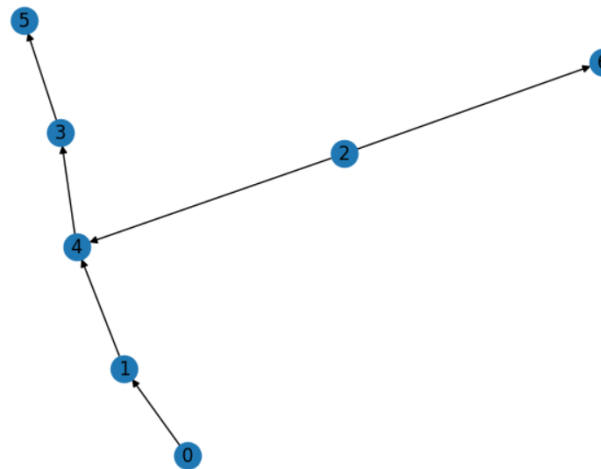
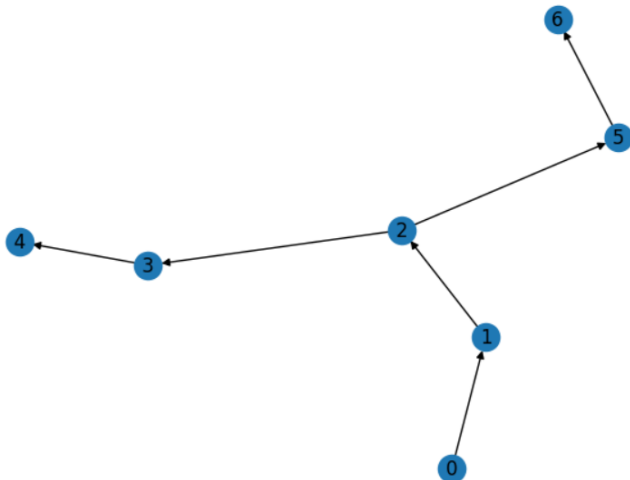
```
# Generate 2 graphs
G1 = nx.DiGraph()
G2 = nx.DiGraph()

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

{}



➤ Another example for directed graph

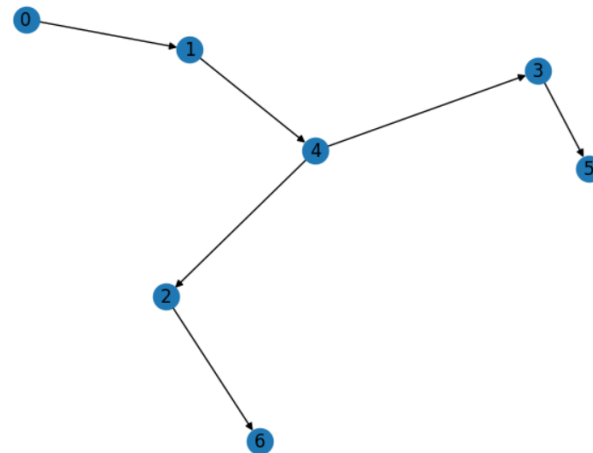
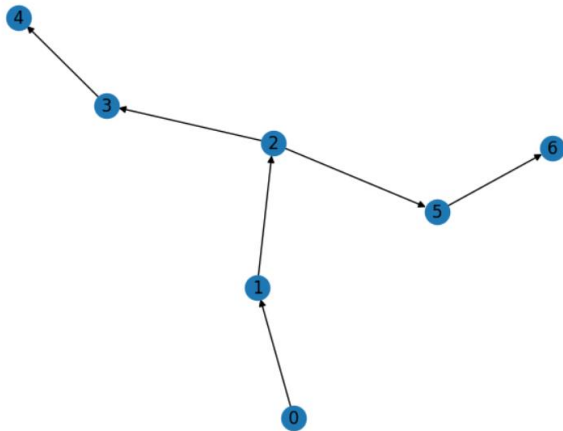
```
# Generate 2 directed graphs
G1 = nx.DiGraph()
G2 = nx.DiGraph()

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}



➤ Weisfeiler Lehman (WL) graph hash

```
import networkx as nx

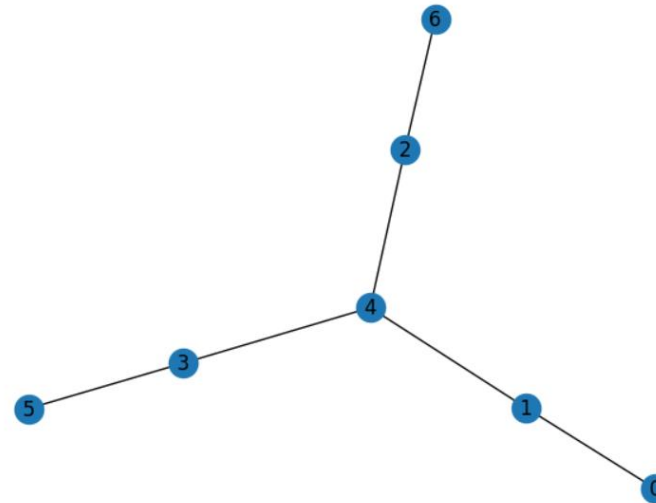
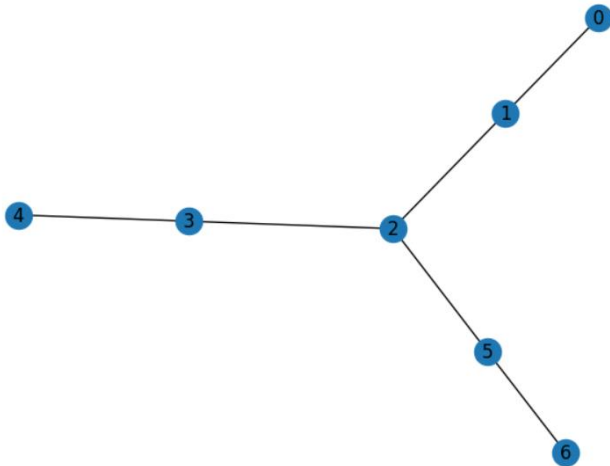
G1 = nx.Graph()
G2 = nx.Graph()

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



➤ Weisfeiler Lehman (WL) graph hash: Directed graph

```
import networkx as nx

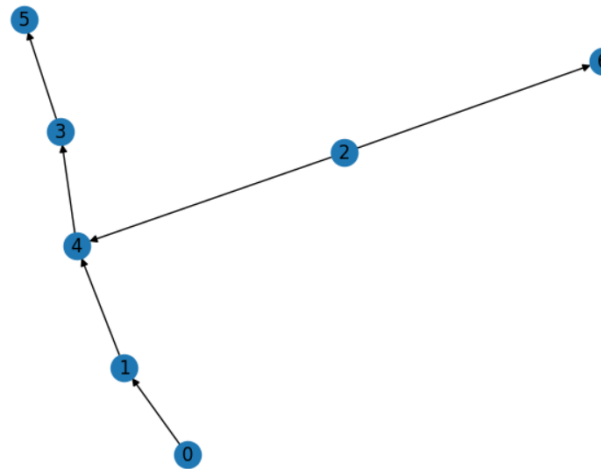
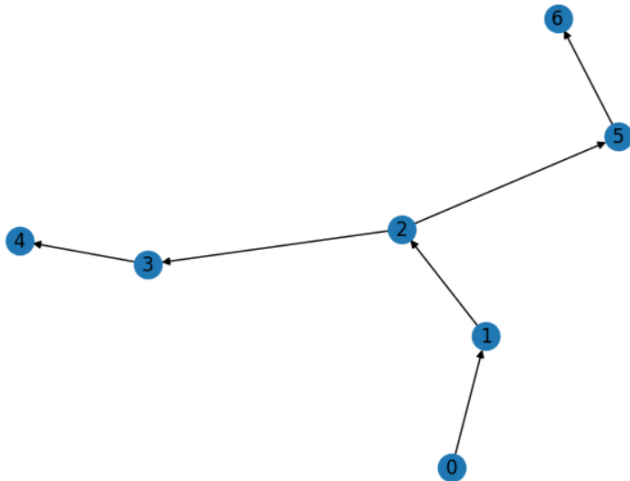
G1 = nx.DiGraph()
G2 = nx.DiGraph()

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 과학연구실
NETWORK SCIENCE LAB



가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA

