

# Mining Frequent Subgraphs

Prof. O-Joun Lee

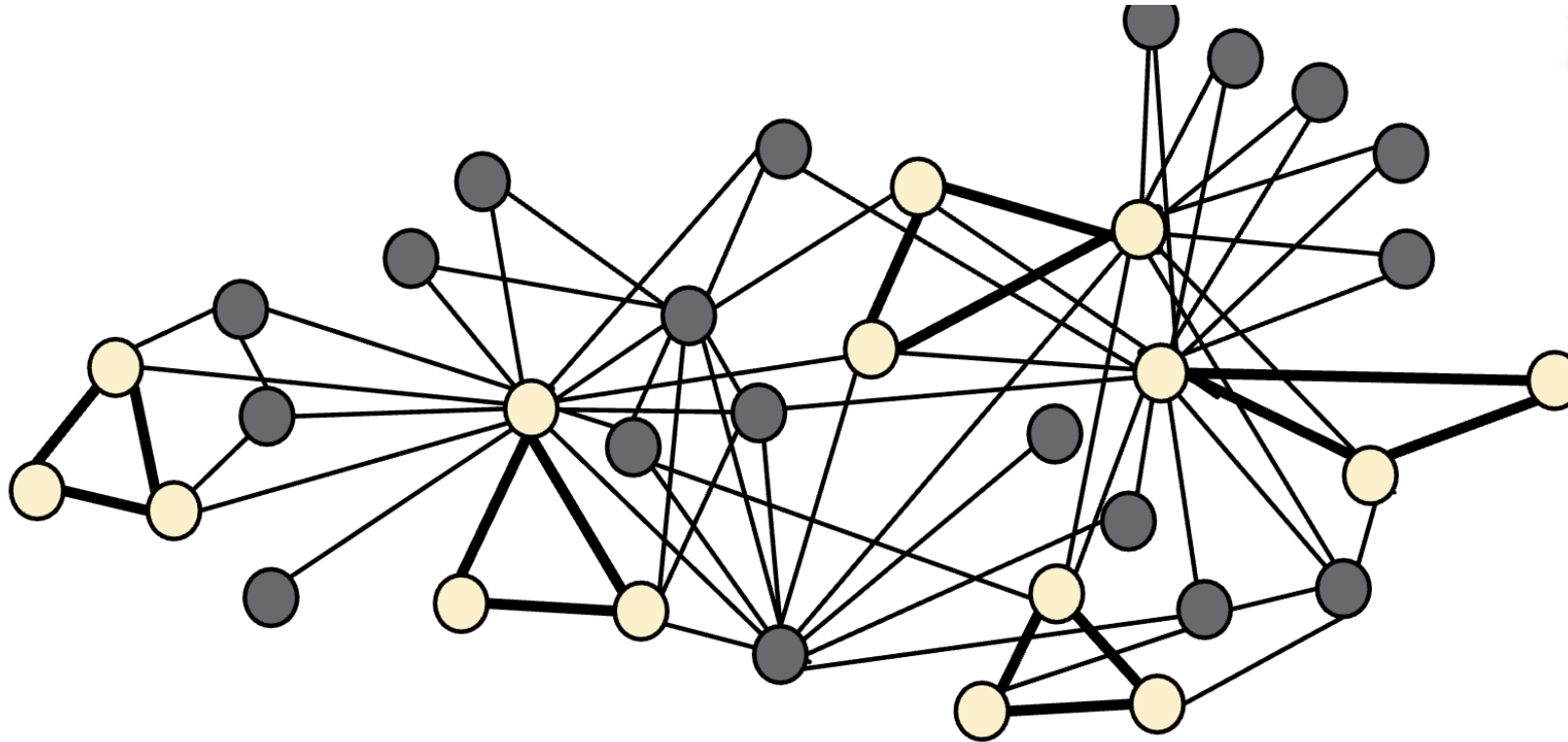
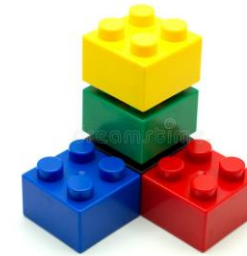
Dept. of Artificial Intelligence,  
The Catholic University of Korea  
*ojlee@catholic.ac.kr*

# Contents



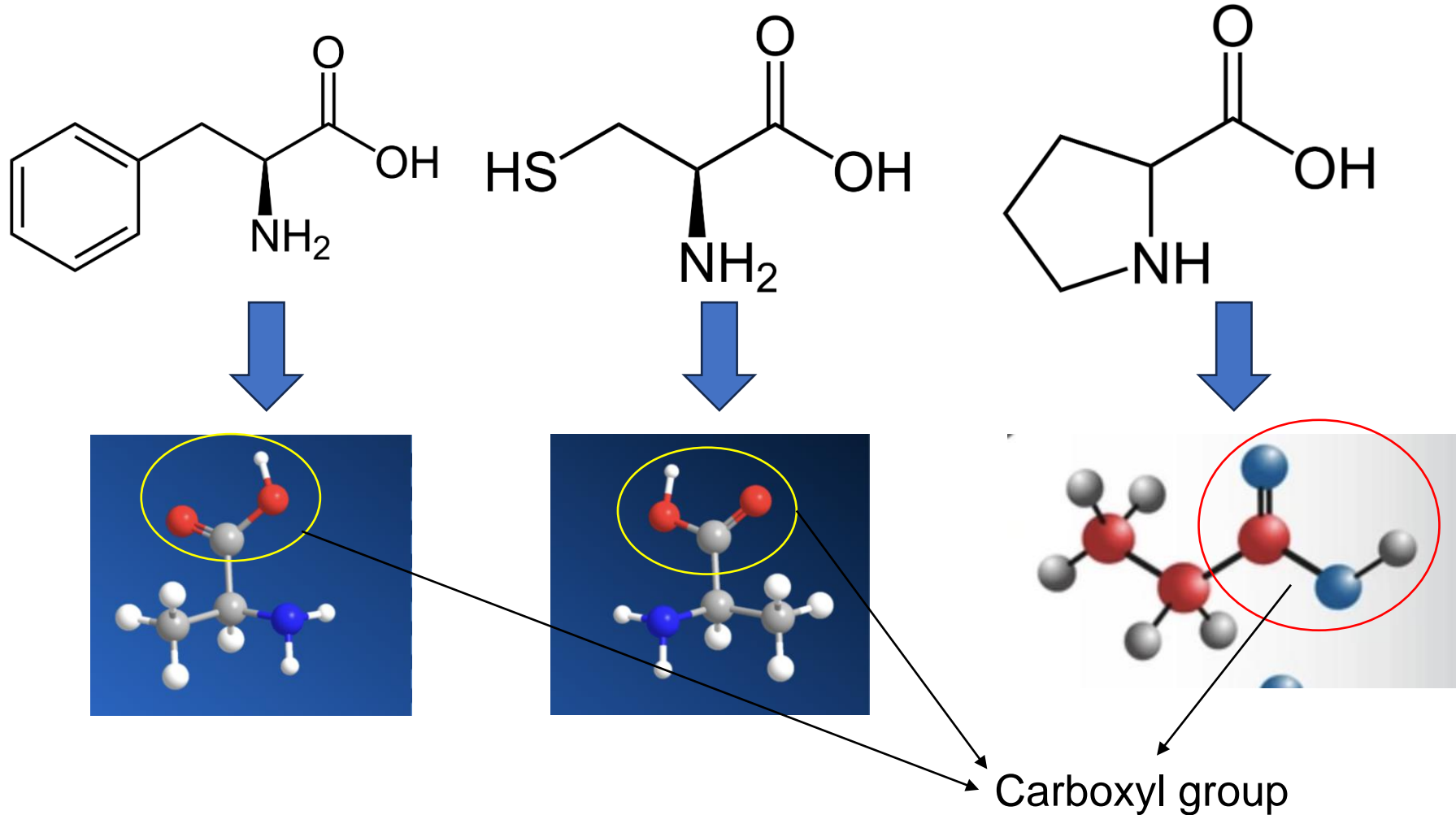
- Subgraph
  - What is Subgraph? How to build subgraph?
- Frequent subgraph mining (FSM).
- FSM algorithms
  - Apriori-Based Approach: FSG.
  - DFS Approach.
  - Subdue Approach.
- Sample code: Mining frequent subgraphs in a graph using the gSpan algorithm in NetworkX.

- **Subgraph:** are the building blocks of networks



- They have the power to **characterize** and **discriminate** networks.

- In many domains, recurring structural components determine the function or behaviour of the graph

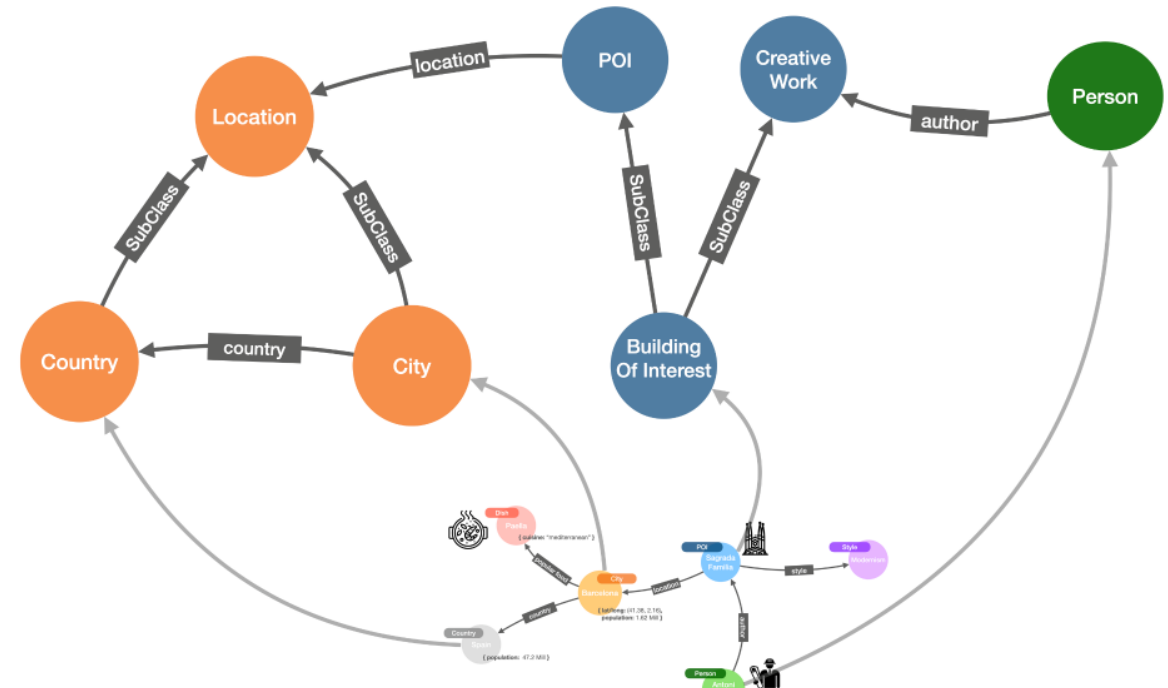
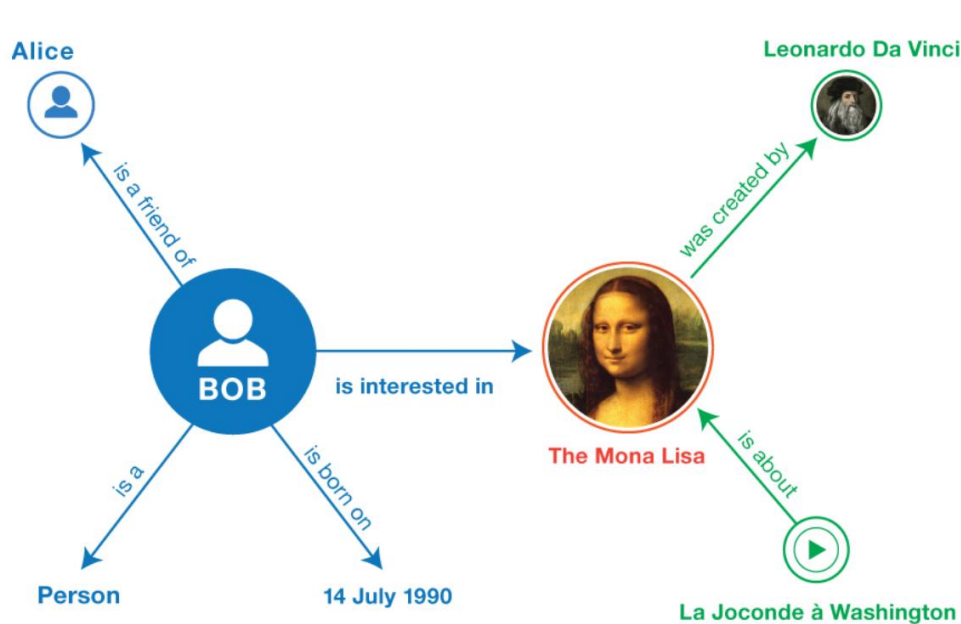


- Given a graph  $G = (V, E)$ .
- Two ways to formalize "network building blocks":
  - Node-induced subgraph.
  - Edge-induced subgraph.
- **Node-induced subgraph:** Take subset of the nodes and all edges induced by the nodes:
  - $G' = (V', E')$  is a node induced subgraph iff
    - $V' \subseteq V$ .
    - $E' = \{(u, v) \in E \mid u, v \in V'\}$ .
    - $G'$  is the subgraph of  $G$  induced by  $V'$ .

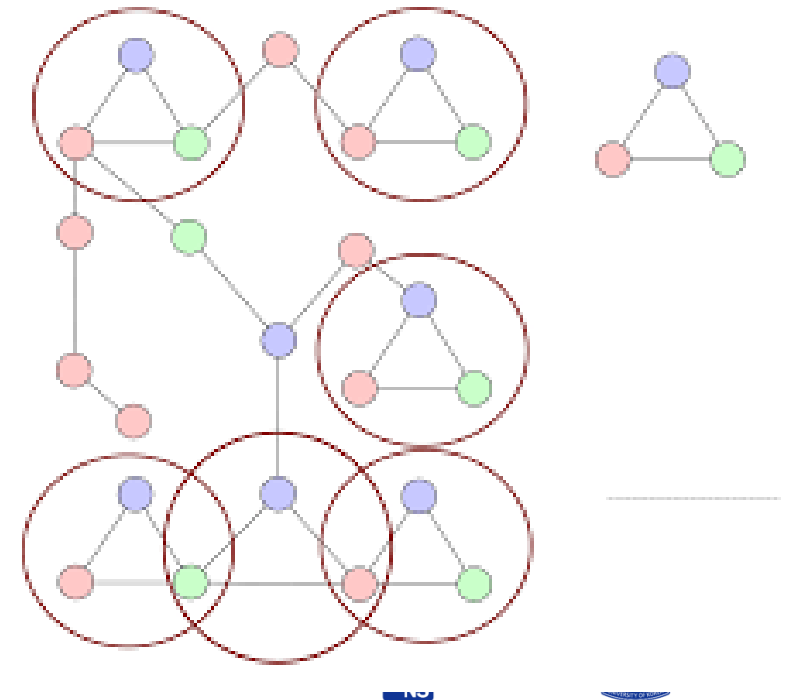


- **Edge-induced subgraph:** Take subset of the edges and all corresponding nodes :
  - $G' = (V', E')$  is an edge induced subgraph iff
    - $E' \subseteq E$ .
    - $V' = \{v \in V | (v, u) \in E' \text{ for some } u\}$ .

- **Chemistry:** Node-induced (functional groups).
- **Knowledge graphs:** Often edge-induced (focus is on edges representing logical relations).



- Frequent pattern: a structure (a set of items, subsequence, substructures, etc.) that occurs frequently in a data set
- Motivation: Finding inherent regularities in data
  - What products were often purchased together?
  - What are the subsequent purchases after buying a PC?
  - What sequences of DNA are sensitive to this new drug?
  - Which topics are in a collection of documents?





➤ *Frequent* subgraphs:

- A (sub)graph is **frequent** if its *support* (occurrence frequency) in a given dataset is no less than a *minimum support* threshold.

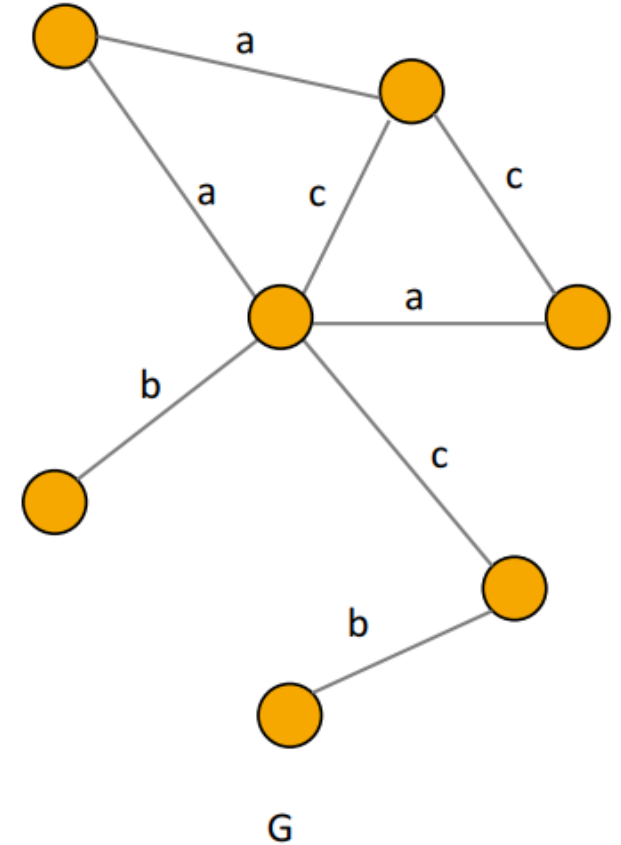
➤ Applications of graph pattern mining:

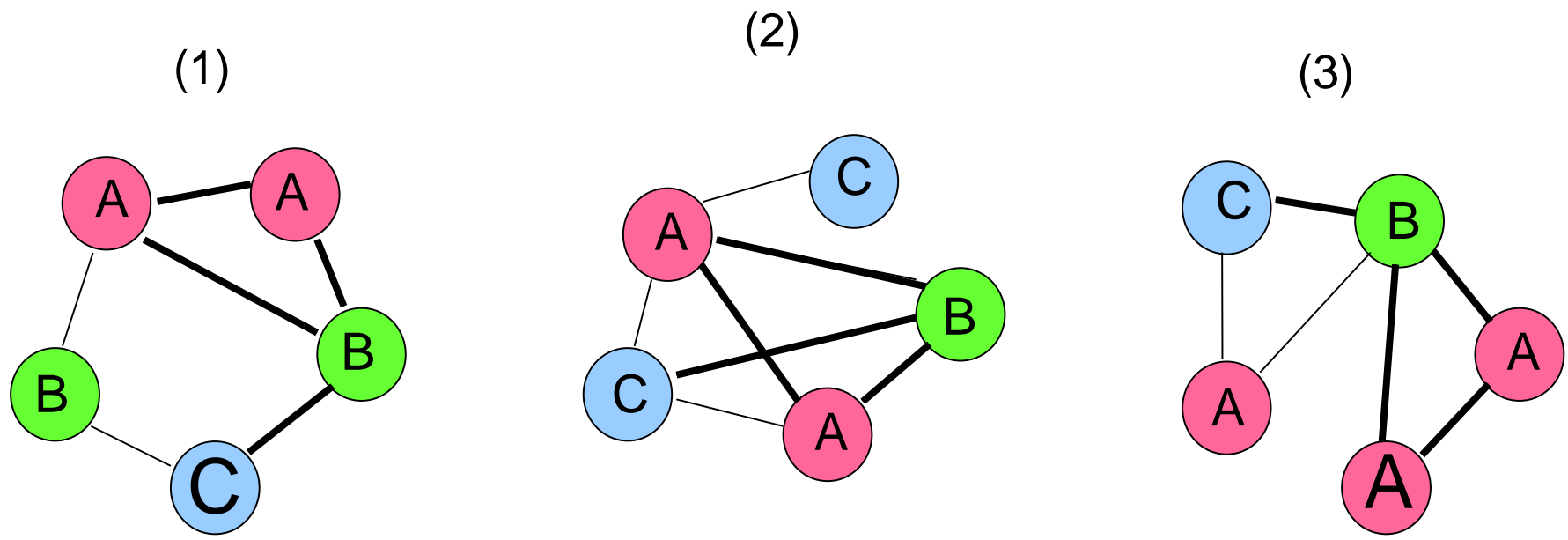
- Mining biochemical structures.
- Program control flow analysis.
- Mining XML structures or Web communities.
- Building blocks for graph classification, clustering, compression, comparison, and correlation analysis.

- Also called Downward closure Property.
- All subsets of a frequent pattern must also be frequent
  - Because any item that contains X must also contains subset of X.

If we have already verified that X is infrequent, there is no need to count X's supersets because they MUST be infrequent too.

- Problem: Find all subgraphs of  $G$  that appear at least  $t$  times
- Suppose  $t = 2$ , the frequent subgraphs are (only edge labels)
  - $a, b, c$
  - $a-a, a-c, b-c, c-c$
  - $a-c-a \dots$
- Exponential number of patterns!

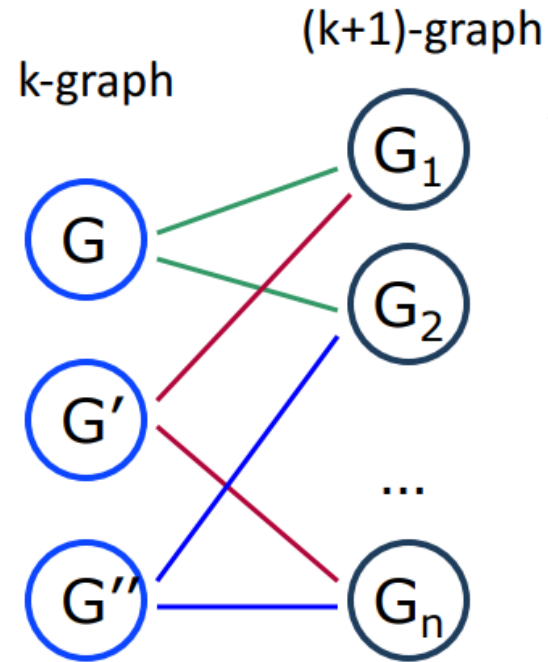




Support	1	3	3
Subgraph			

- Apriori-based approaches:
  - Start with small-size subgraphs and proceeds in a bottom-up manner.
  - Join two patterns to create bigger size patterns (through Apriori principle).
  - Several approaches:
    - FSG.
    - PATH#.
- Pattern-growth approaches:
  - Extends existing frequent graphs by adding one edge.
  - Several approaches:
    - gSpan, MoFa.
    - Gaston FFSM, SPIN.
- Greedy approaches:
  - Subdue.

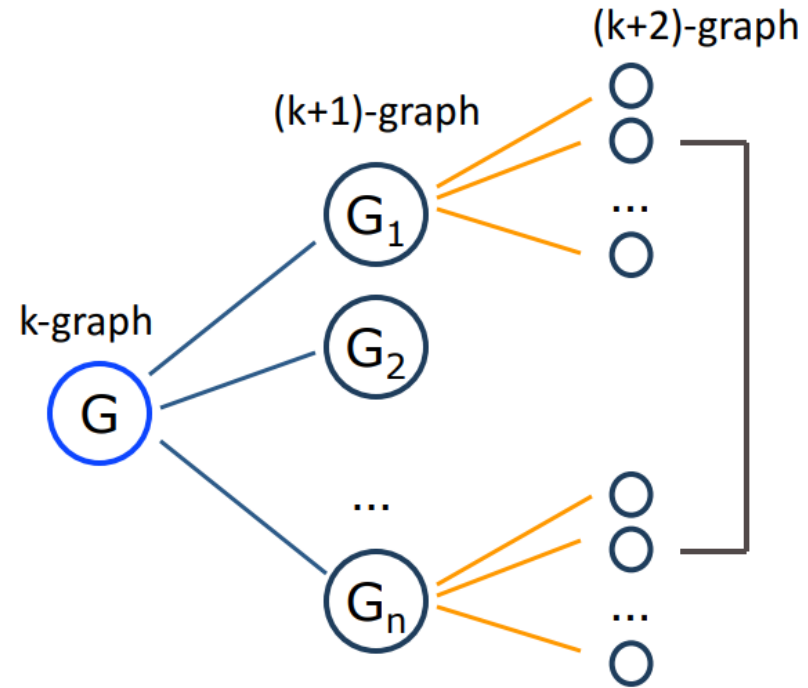
- Start with small-size subgraphs and proceeds in a bottom-up manner
- Join two patterns to create bigger size patterns (through Apriori principle)



- Problem:
  - Join operation among graphs is extremely expensive



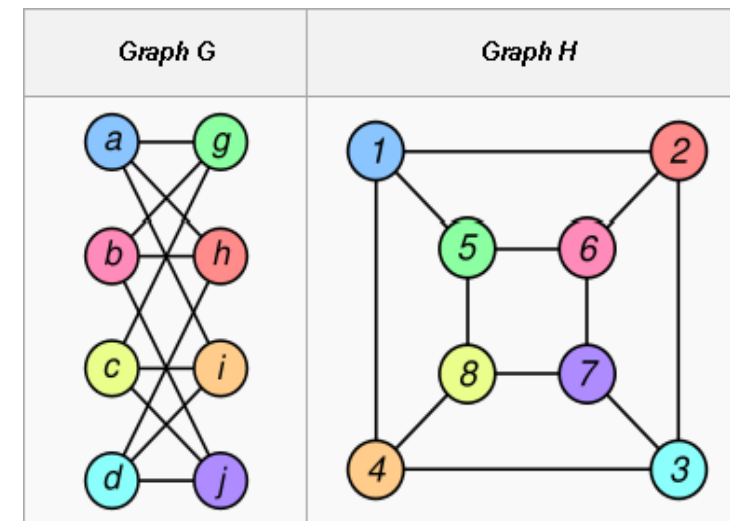
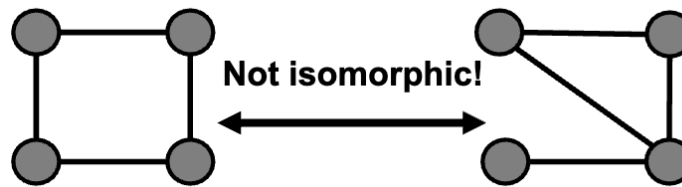
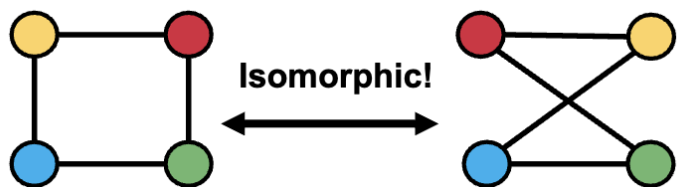
- Generate patterns expanding existing ones.
- Extends existing frequent graphs by adding one edge.



- Problems:
  - Duplicate graphs

## ➤ Graph isomorphism:

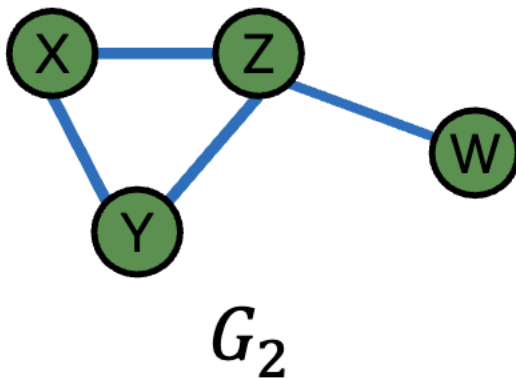
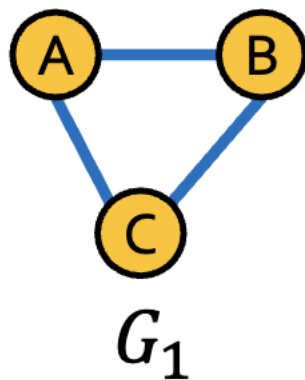
- To detect if two graphs are identical in structure.
- $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  are isomorphic if there exists a **bijection**  $f: V_1 \rightarrow V_2$  such that  $(u, v) \in E_1$  iff  $(f(u), f(v)) \in E_2$ 
  - $f$  is called the **isomorphism**:



## ➤ Graph representation (Canonical Labeling)

- A canonical label is a **unique code** of a given graph.
- Canonical label should be **the same no matter how graphs are represented**, as long as graphs have the same topological structure and the same labeling of edges and vertices.

- $G_2$  is **subgraph-isomorphic** to  $G_1$  if some subgraph of  $G_2$  is isomorphic to  $G_1$ 
  - We also commonly say  $G_1$  is a subgraph of  $G_2$ .
  - We can use either the node-induced or edge-induced definition of subgraph.
  - This is **NP-hard** problem.



$f:$

$V_1$	$V_2$
A	X
B	Y
C	Z

A-B-C matches with X-Y-Z: There is a subgraph isomorphism between  $G_1$  and  $G_2$ .

- Returns True if the graphs G1 and G2 are isomorphic and False otherwise.  
(The two graphs G1 and G2 must be the same type).

```
#Import networkx, isomorphism
import networkx as nx
import networkx.algorithms.isomorphism as iso

#create graph 1
G1 = nx.Graph()

G1.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F'])

G1.add_edges_from([('A', 'B'), ('A', 'C'), ('A', 'D'), ('A', 'E'), ('A', 'F')])

#create graph 2
G2 = nx.star_graph(5)
```

## Testing if two graphs are isomorphic

```
nx.is_isomorphic(G1, G2)
```

True

## ➤ How to find edge mapping

```
import networkx as nx
G1 = nx.Graph()
G1.add_weighted_edges_from([(0,1,0), (0,2,1), (0,3,2)], weight = 'aardvark')
G2 = nx.Graph()
G2.add_weighted_edges_from([(0,1,0), (0,2,2), (0,3,1)], weight = 'baboon')
G3 = nx.Graph()
G3.add_weighted_edges_from([(0,1,0), (0,2,2), (0,3,2)], weight = 'baboon')

def comparison(D1, D2):
    #for an edge u,v in first graph and x,y in second graph
    #this tests if the attribute 'aardvark' of edge u,v is the
    #same as the attribute 'baboon' of edge x,y.

    return D1['aardvark'] == D2['baboon']

nx.is_isomorphic(G1, G2, edge_match = comparison)
```

True

```
nx.is_isomorphic(G1, G3, edge_match = comparison)
```

False

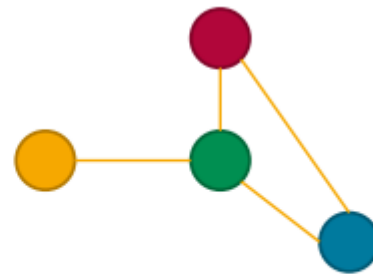
- Given a set of 4 graphs:



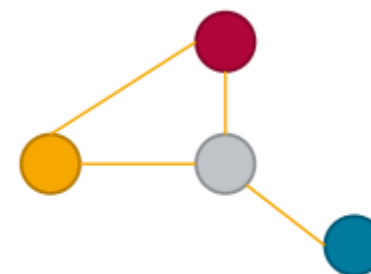
G1



G2

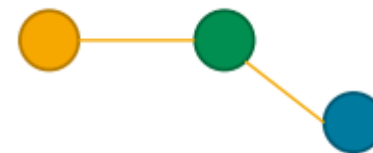


G3



G4

- Support: frequency of a subgraph appearing in a set of graphs
- Frequent subgraph Min support =  $3/4$



Apriori principle (for graphs): If a graph is frequent, all of its subgraphs are frequent



➤ We define a labeled graph  $G$  as a five-element tuple  $G = \{V, E, \Sigma_V, \Sigma_E, \delta\}$

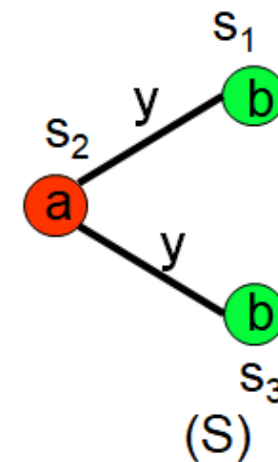
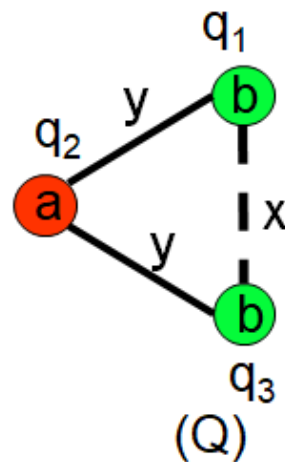
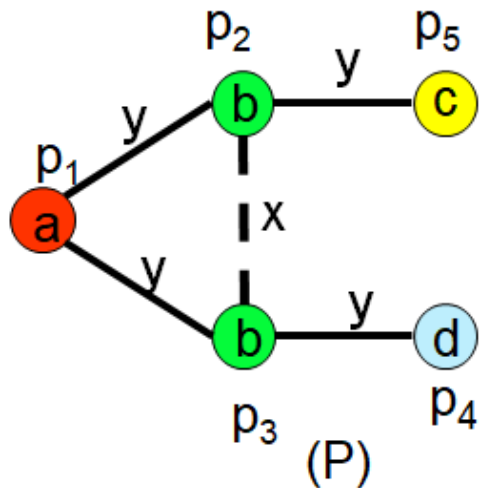
Where:

$V$  is the set of vertices of  $G$ ,

$E \subseteq V \times V$  is a set of undirected edges of  $G$ ,

$\Sigma_V$  ( $\Sigma_E$ ) are set of vertex (edge) labels,

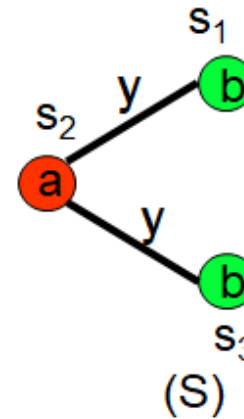
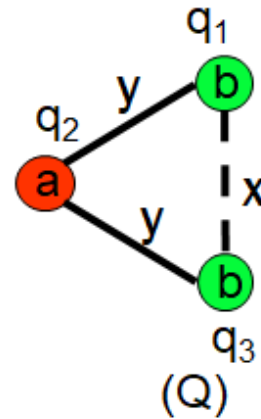
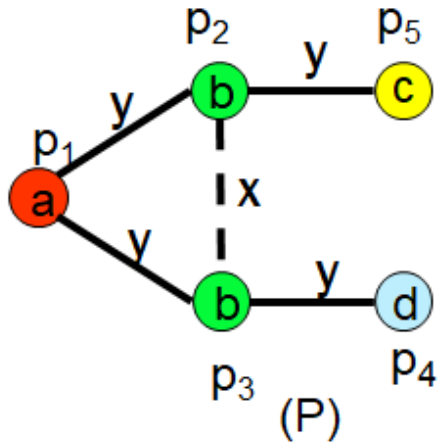
$\delta$  is the labeling function:  $V \subseteq \Sigma_V$  and  $E \rightarrow \Sigma_E$  that maps vertices and edges to their labels.



- Support: Given a set of labelled graphs:
  - $D = \{G_1, G_2, \dots, G_n\}, G_i = \langle V_i, E_i, l_i \rangle$
  - A subgraph  $G$ .
- The supporting set of  $G$  is:  $D = \{G_i | G \sqsubseteq G_i, G_i \in D\}$   
Where  $G \sqsubseteq G_i$  indicates that  $G$  is subgraph isomorphic to  $G_i$ .
- The support is defined as:

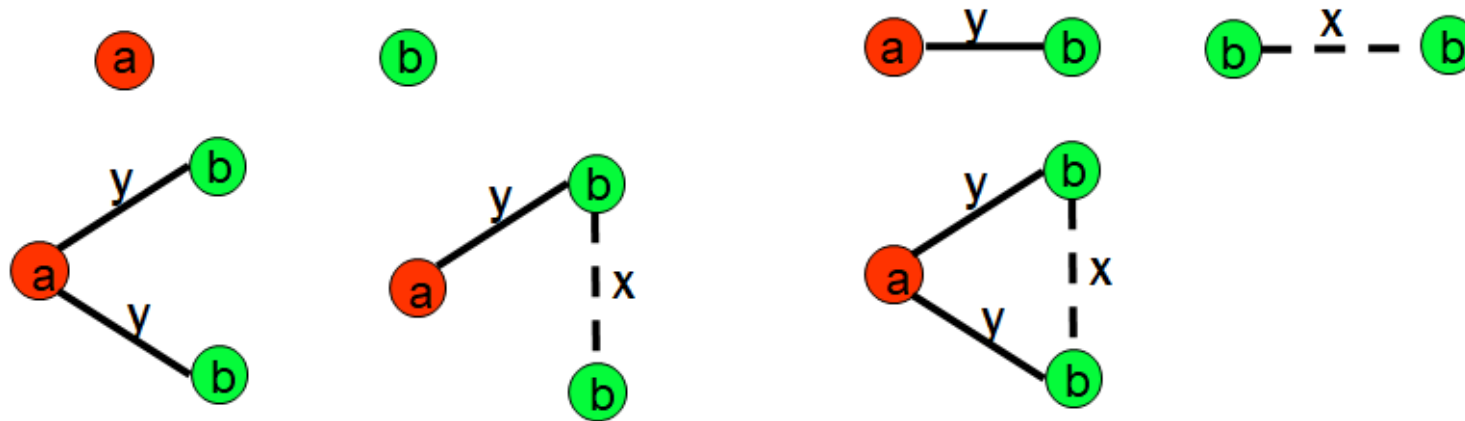
$$\sigma(G) = \frac{|D_G|}{|D|}$$

- *Input: A set Graph data of labeled undirected graphs*



$$\sigma = 2/3$$

- All frequent subgraphs (w. r. t.  $\sigma$ ) from *Graph data*.



## ➤ Input:

- Set of labeled-graphs  $D = \{G_1, G_2, \dots, G_n\}$ ,  $G_i = \langle V_i, E_i, \ell_i \rangle$
- Minimum support  $\text{min\_sup}$

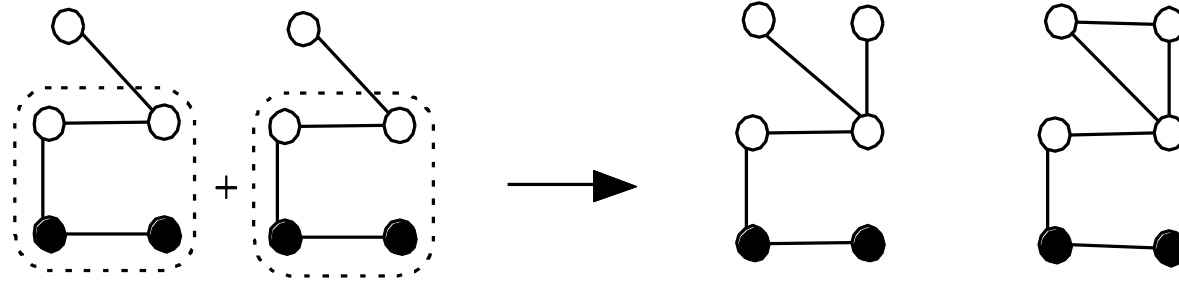
## ➤ Output:

- A subgraph  $G$  is frequent if  $\sigma(G) \geq \text{min\_sup}$
- Each subgraph is connected.

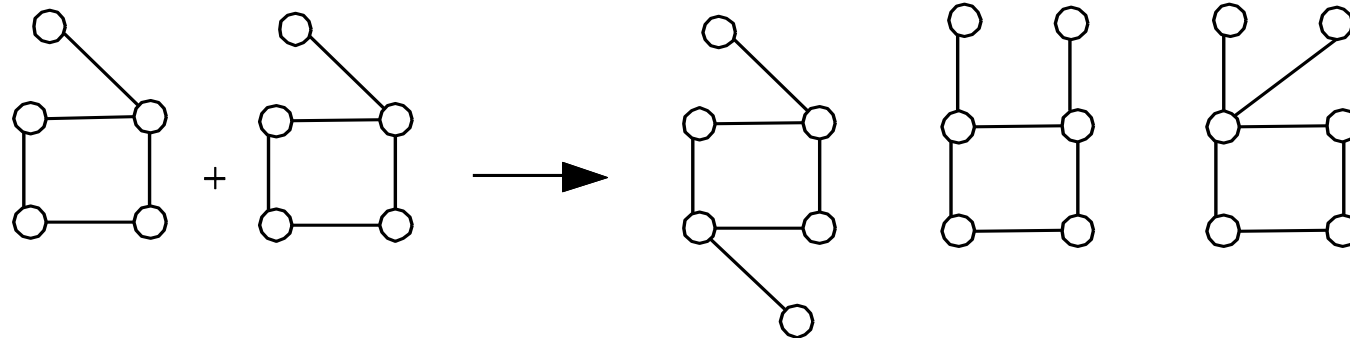
- Apriori-based approaches:
  - FSG.
- Pattern-growth approaches:
  - gSpan.
- Greedy approach:
  - Subdue.

➤ Methodology: breadth-search, joining two graphs

1. Generates new graphs with **one more node**



2. Generates new graphs with **one more edge**



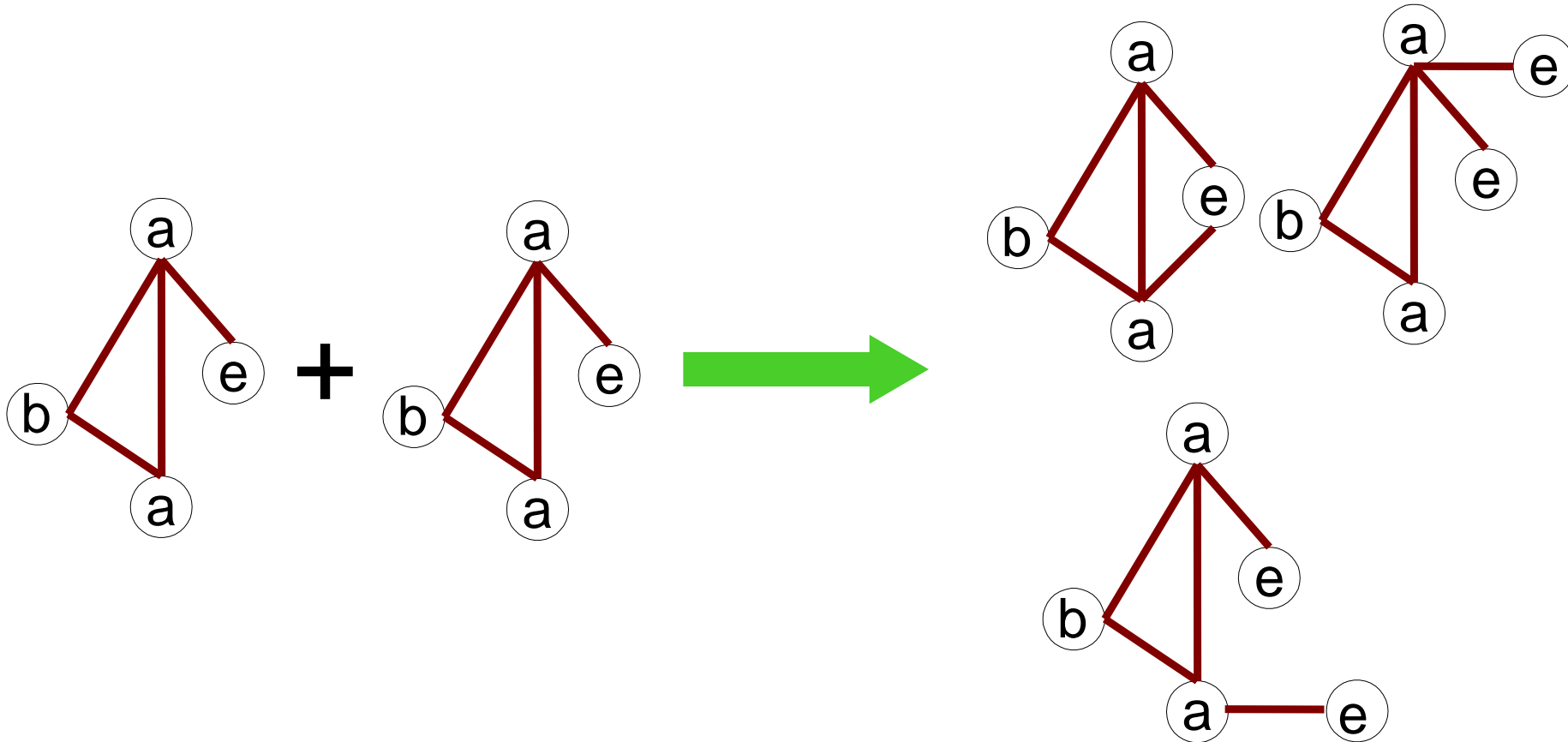


- $K = 1$
- $F_1 =$  all frequent edges
- Repeat
  - $K = K + 1$ ;
  - $C_K = \text{join}(F_{K-1})$
  - $F_K =$  frequent patterns in  $C_K$
  - Until  $F_K$  is empty

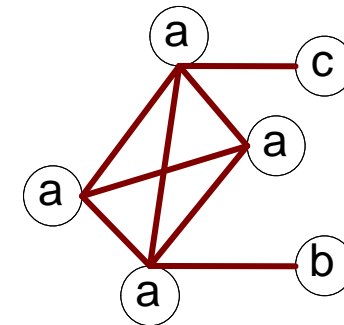
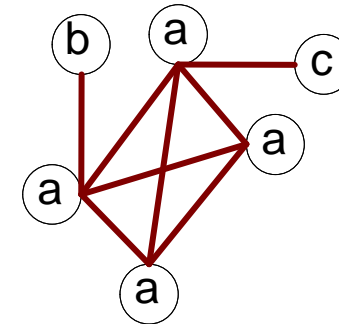
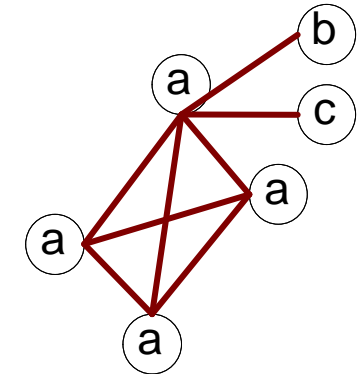
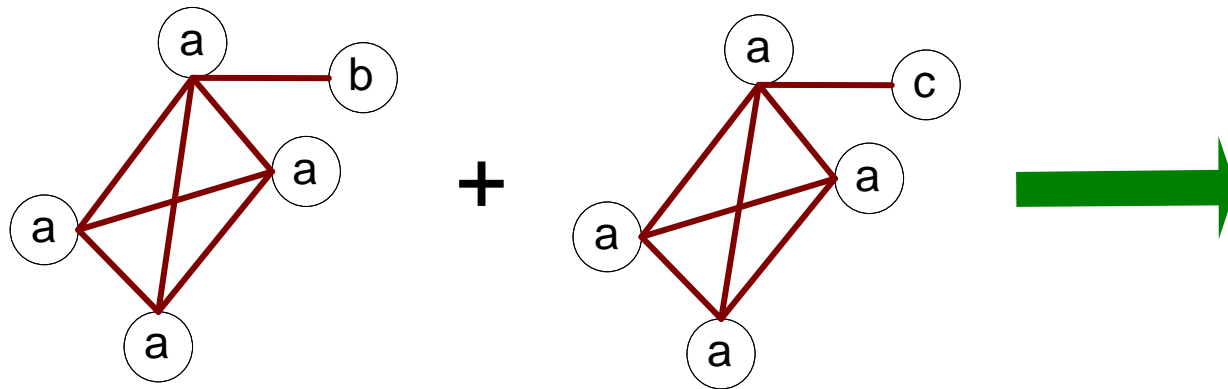
- $K = 1$
- $F_1 =$  all frequent edges
- Repeat
  - $K = K + 1$ ;
  - $C_K = \text{join}(F_{K-1})$
  - $F_K =$  frequent patterns in  $C_K$
  - Until  $F_K$  is empty

- $\text{Join}(L) = \cup \text{join}(P, Q)$  for all  $P, Q \in L$ .
- $\text{Join}(P, Q) = \{G \mid P, Q, \subset G, |G| = |P| + 1, |P| = |Q|\}$
- Two graphs  $P$  and  $Q$  are **joinable** if the join of the two graphs produces a non-empty set.
- Theorem: two graphs  $P$  and  $Q$  are joinable if  $P \cap Q$  is a graph with size  $|P| - 1$  or share a common “core” with size  $P-1$ .

➤ Case 1: Identical node labels

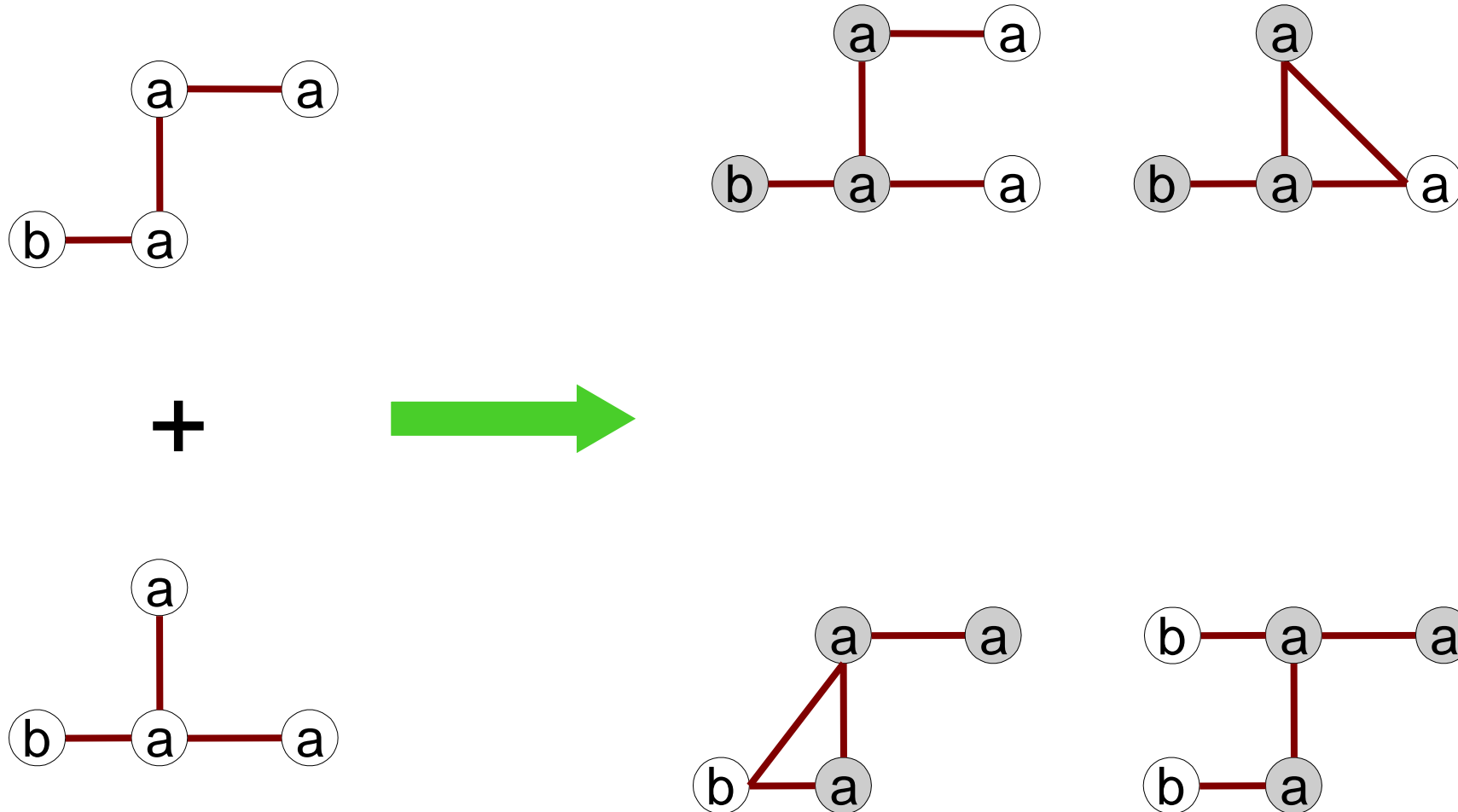


➤ Case 2: Core contains identical labels



Core: The  $(k-1)$  subgraph that is common between the joint graphs

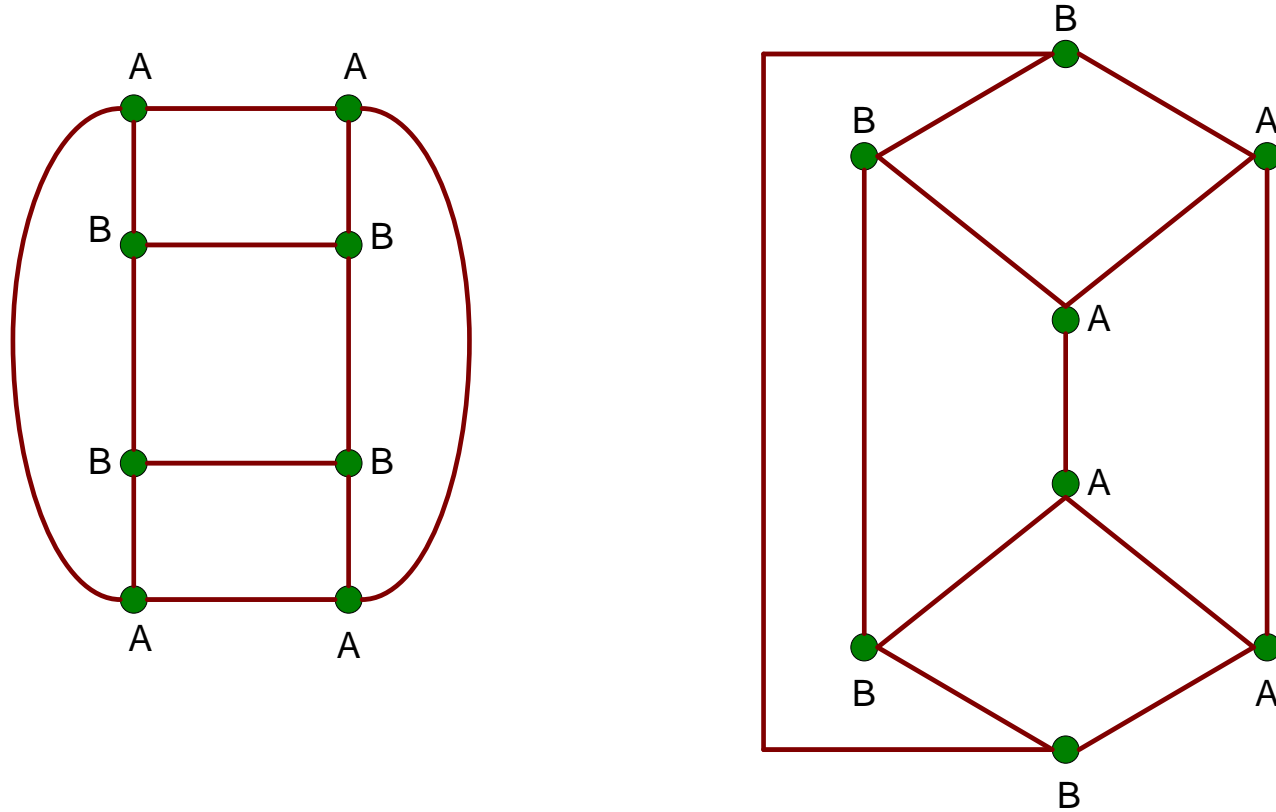
## ➤ Case 3: Core multiplicity



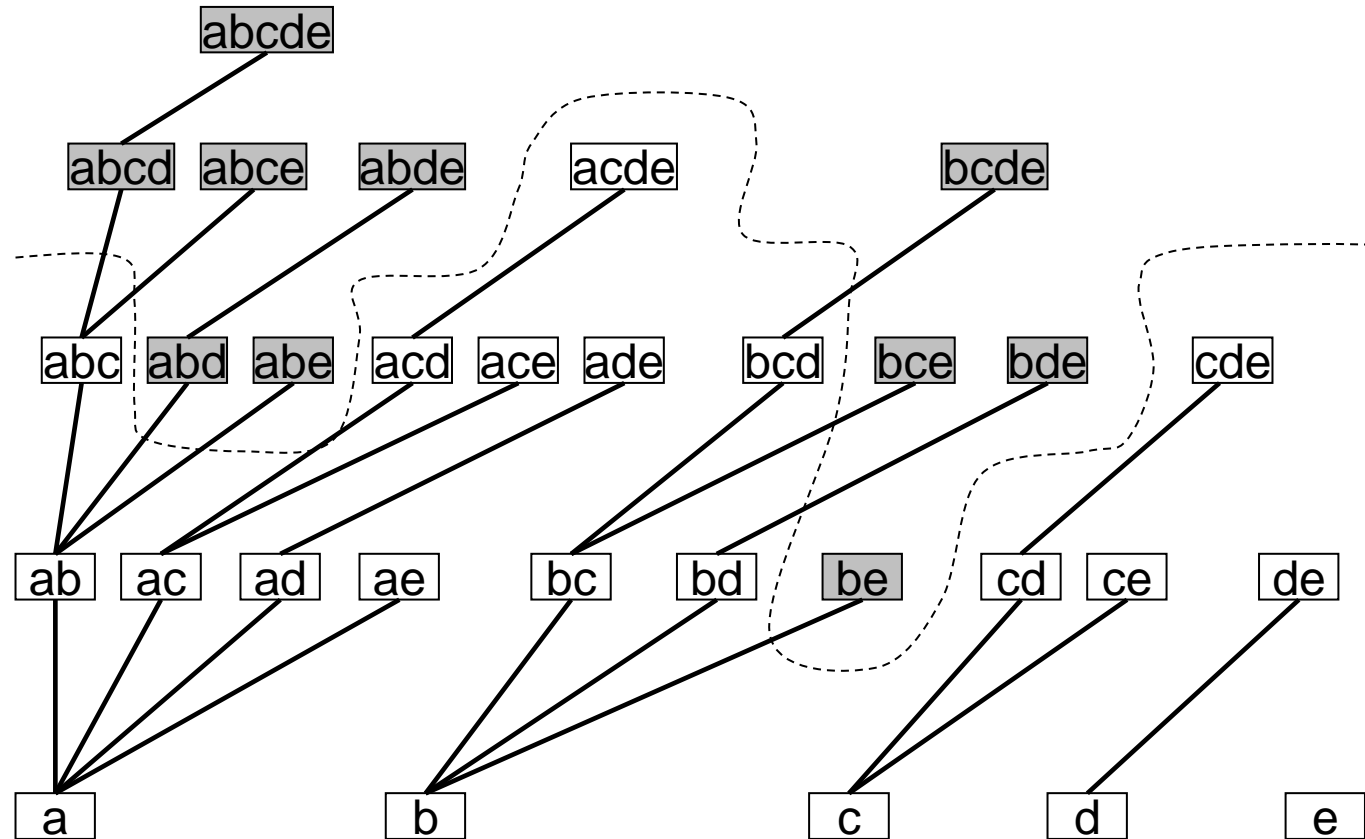


- Graph isomorphism
  - Two graphs may have the same topology though their layouts are different.
- Computational complexity: computing a generating set for the automorphism group of a graph have the same complexity.
  - **NP-hard problem.**
- Subgraph isomorphism
  - How to compute the support value of a pattern.
- Neural subgraph matching uses a machine learning-based approach to learn the NP-hard problem of subgraph isomorphism (from week 10).

- A graph is isomorphic if it is topologically equivalent to another graph

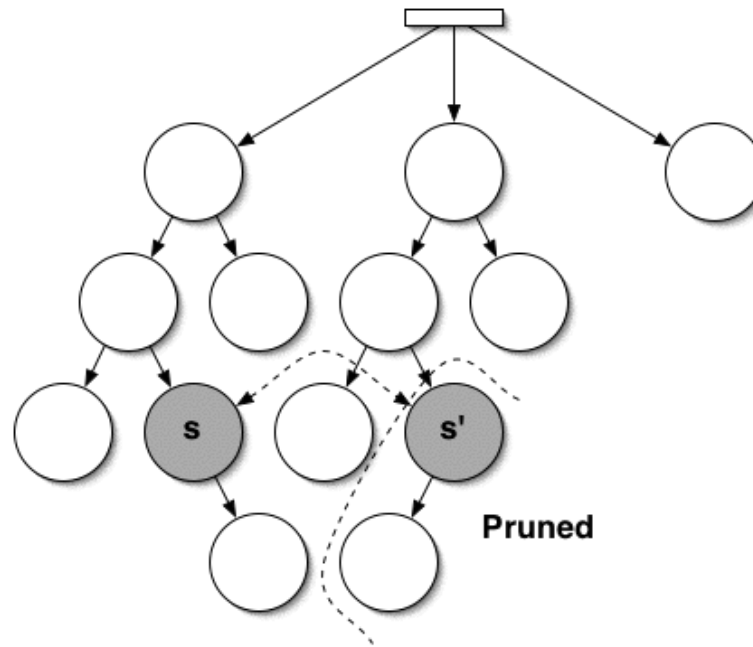


➤ Itemset search space – prefix based



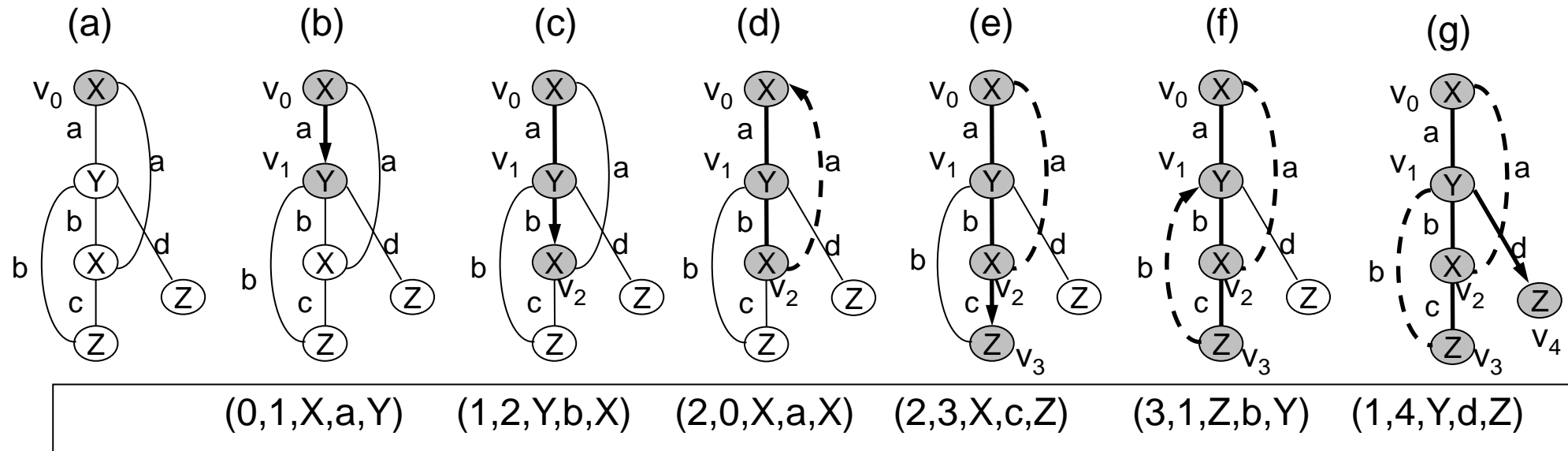
- Canonical representation of itemset is obtained by a complete order over the items.
- Each possible itemset appear in TSS exactly once - no duplications or omissions.
- Properties of Tree search space
  - For each k-label, its parent is the k-1 prefix of the given k-label.
  - The relation among siblings is in ascending lexicographic order.

- Organize DFS code nodes as parent-child.
- Pre-order traversal follows DFS lexicographic order.
- If  $s$  and  $s'$  are the same graph with different DFS codes,  $s'$  is not the minimum and can be pruned.

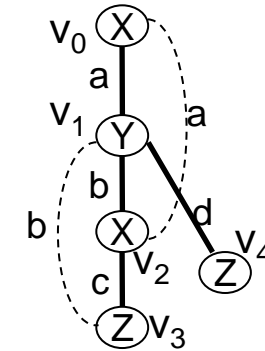
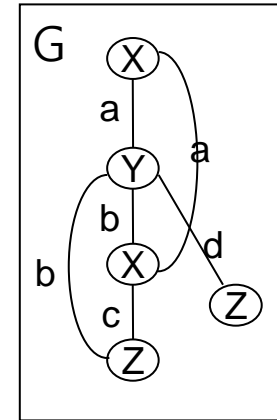


- Map each graph (2-Dim) to a sequential DFS Code (1-Dim).
- Lexicographically order the codes.
- Construct Tree Search Space based on the lexicographic order.

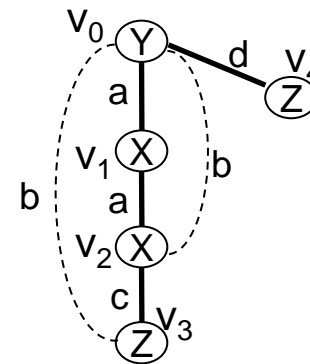
- Given a graph  $G$ . for each Depth First Search over graph  $G$ , construct the corresponding DFS-Code.



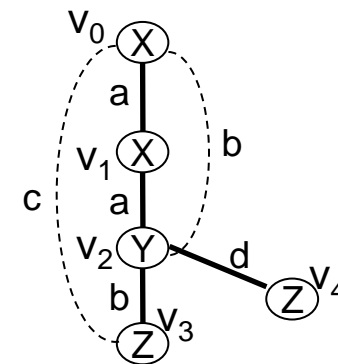
	(a)	(b)	(c)
1	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, X, a, X)
2	(1, 2, Y, b, X)	(1, 2, X, a, X)	(1, 2, X, a, Y)
3	(2, 0, X, a, X)	(2, 0, X, b, Y)	(2, 0, Y, b, X)
4	(2, 3, X, c, Z)	(2, 3, X, c, Z)	(2, 3, Y, b, Z)
5	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	(3, 0, Z, c, X)
6	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	(2, 4, Y, d, Z)



(a)



(b)

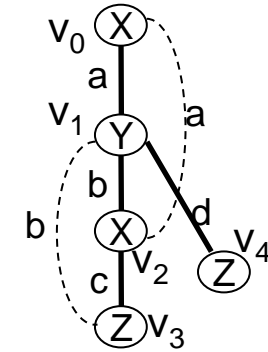
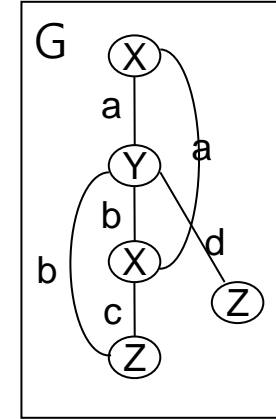


(c)

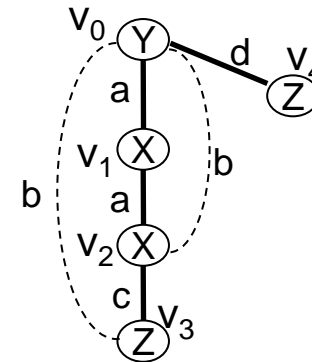


Min  
DFS-Code

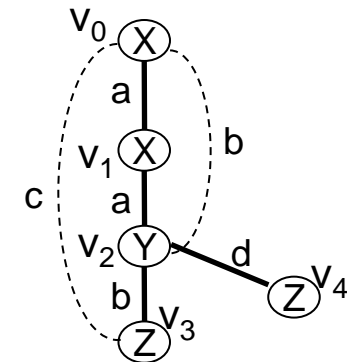
	(a)	(b)	(c)
1	(0, 1, X, a, Y)	(0, 1, Y, a, X)	<b>(0, 1, X, a, X)</b>
2	(1, 2, Y, b, X)	(1, 2, X, a, X)	<b>(1, 2, X, a, Y)</b>
3	(2, 0, X, a, X)	(2, 0, X, b, Y)	<b>(2, 0, Y, b, X)</b>
4	(2, 3, X, c, Z)	(2, 3, X, c, Z)	<b>(2, 3, Y, b, Z)</b>
5	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	<b>(3, 0, Z, c, X)</b>
6	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	<b>(2, 4, Y, d, Z)</b>



(a)



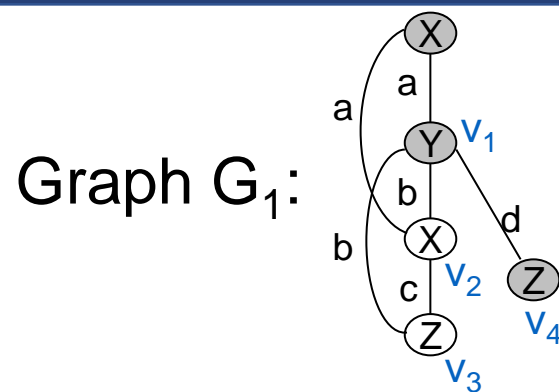
(b)



(c)

- The minimum DFS code  $\min(G)$ , in DFS lexicographic order, is a canonical representation of graph  $G$ .
- Graphs  $A$  and  $B$  are isomorphic if and only if:
$$\min(A) = \min(B)$$

- If  $\min(G_1) = \{a_0, a_1, \dots, a_n\}$  and  $\min(G_2) = \{a_0, a_1, \dots, a_n, b\}$ 
  - $G_1$  is parent of  $G_2$
  - $G_2$  is child of  $G_1$
- A valid DFS code requires that **b** grows from a vertex on the rightmost path (inherited property from the DFS search).

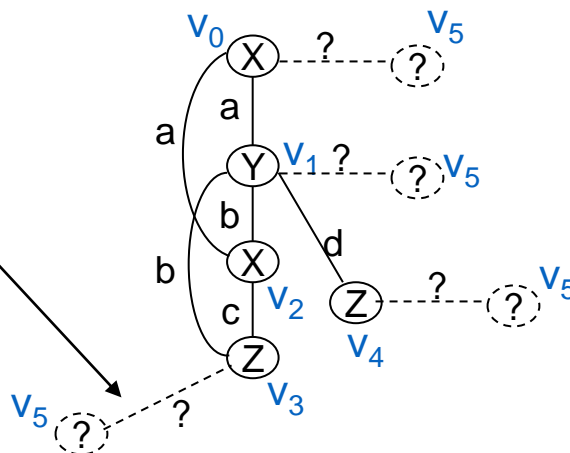


$\text{Min}(g) = (0,1,X,a,Y) \ (1,2,Y,b,X) \ (2,0,X,a,X) \ (2,3,X,c,Z) \ (3,1,Z,b,Y) \ (1,4,Y,d,Z)$

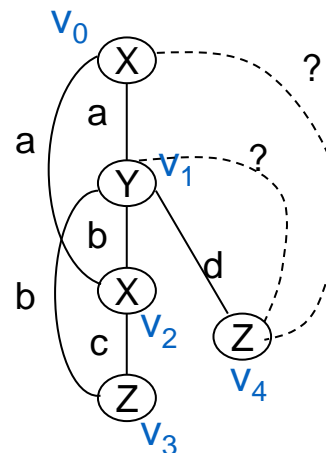
A child of graph  $G_1$  *must grow edge from rightmost path of  $G_1$*  (necessary condition)

Graph  $G_2$ :

**wrong**

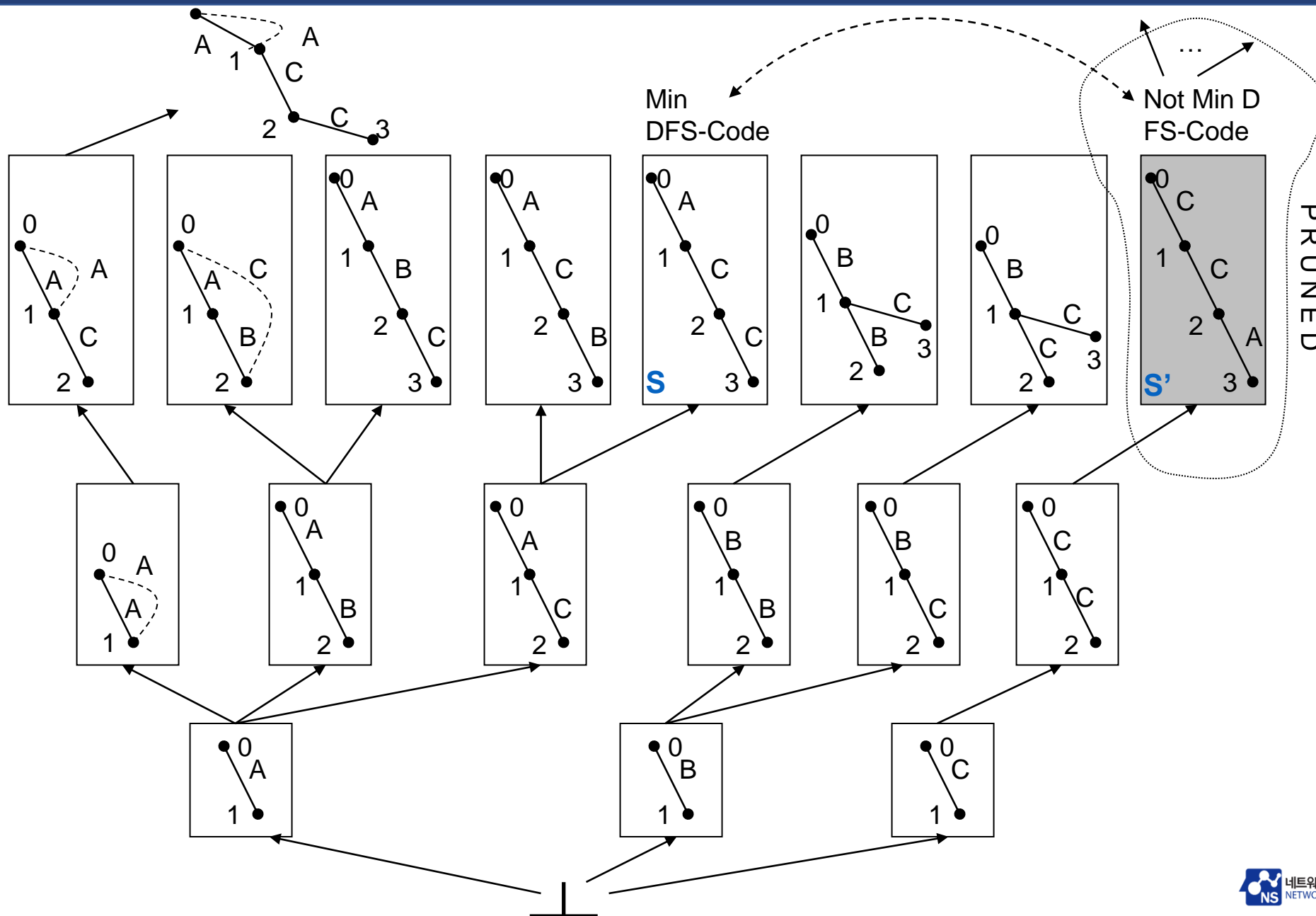


Forward edge



Backward edge

- Organize DFS Code nodes as parent-child.
- Sibling nodes organized in ascending DFS lexicographic order.
- *In-Order* traversal follows DFS lexicographic order.



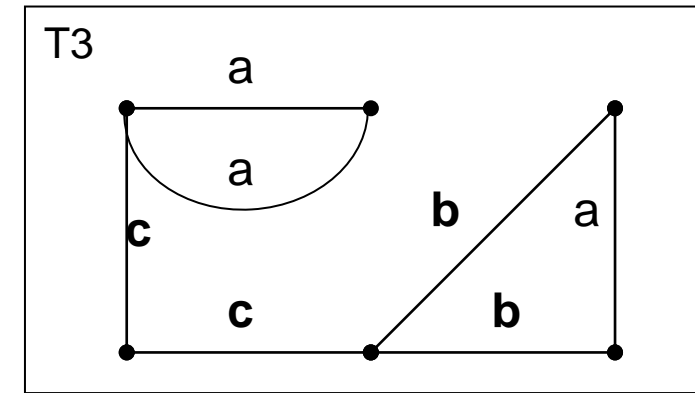
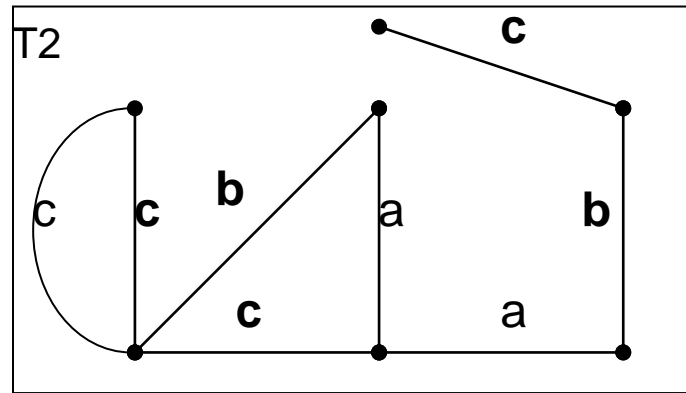
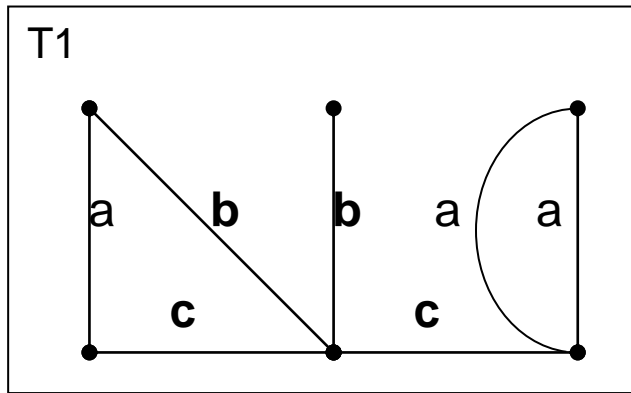
- All of the descendants of infrequent node are infrequent also.
- All of the descendants of a not minimal DFS code are also not minimal DFS codes.

**Function gSpan( $D, F, g$ ):**

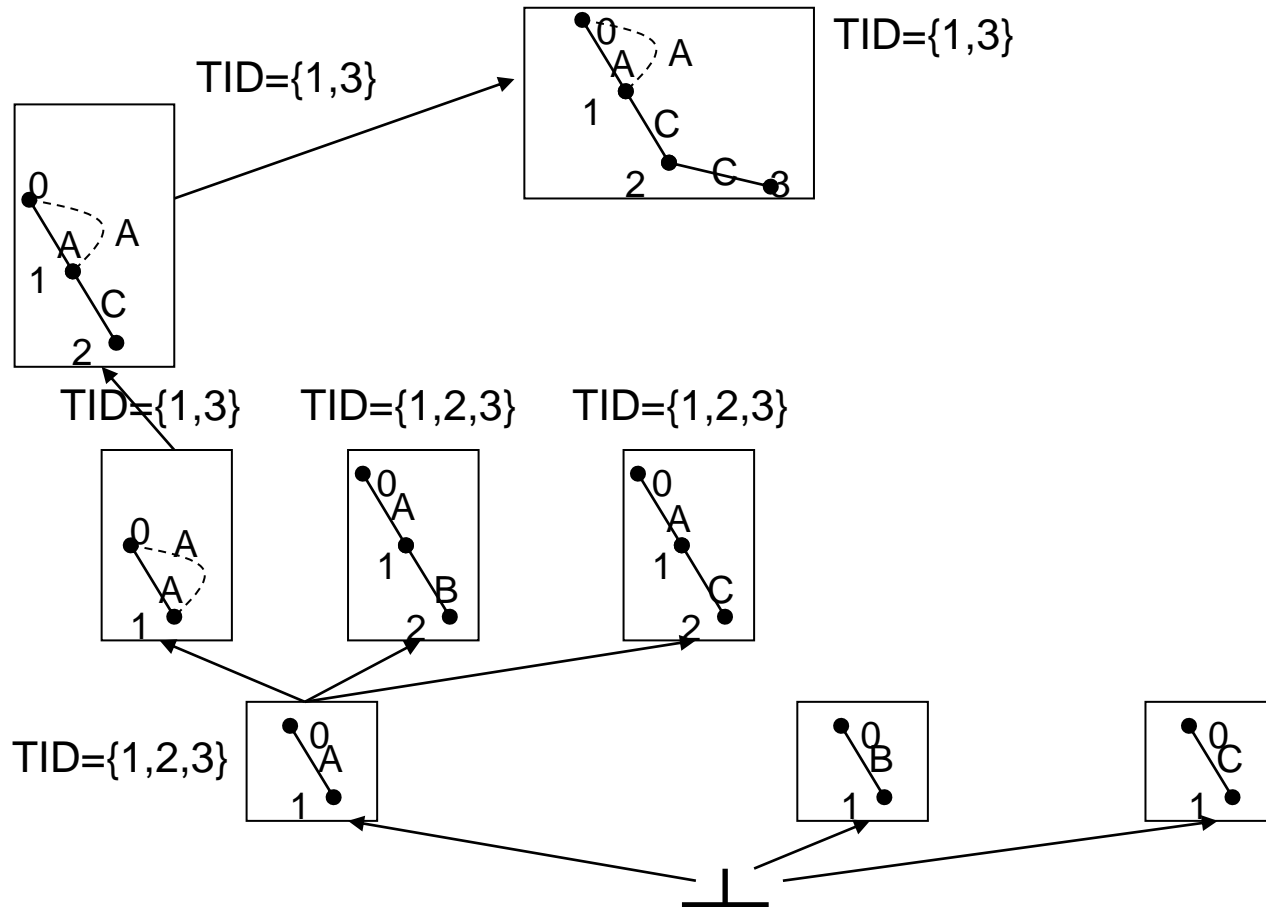
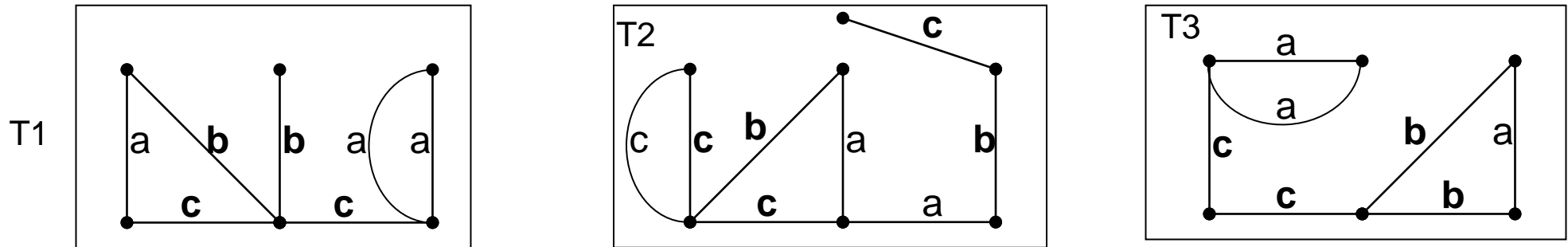
```
1: if  $g \neq \min(g)$   
    return;  
2:  $F \leftarrow F \cup \{ g \}$   
3:  $\text{children}(g) \leftarrow$  [generate all  $g'$  potential children with one edge growth]  
4: Enumerate( $D, g, \text{children}(g)$ )  
5: for each  $c \in \text{children}(g)$   
    if  $\text{support}(c) \geq \# \text{minSup}$   
        SubgraphMining ( $D, F, c$ )
```



Given: database  $D$



Task: Mine all frequent subgraphs with support  $\geq 2$  (#minSup)



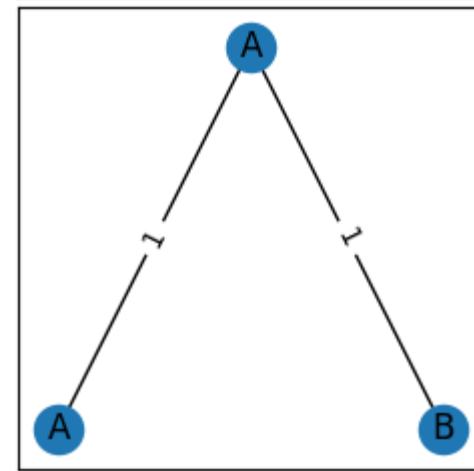
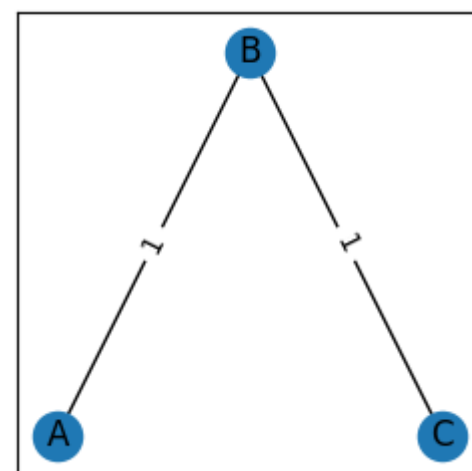
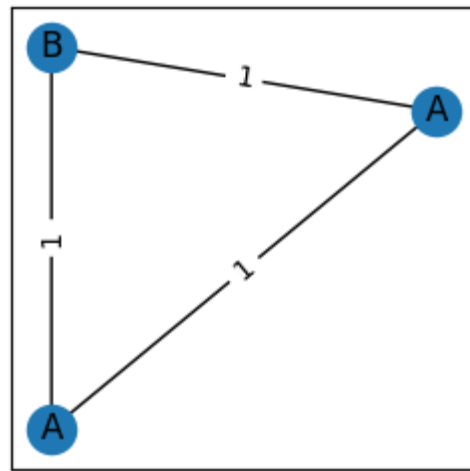
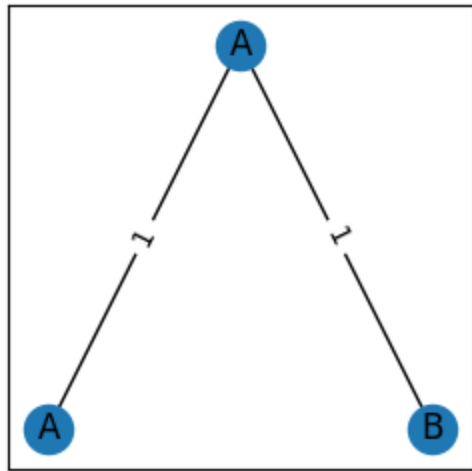
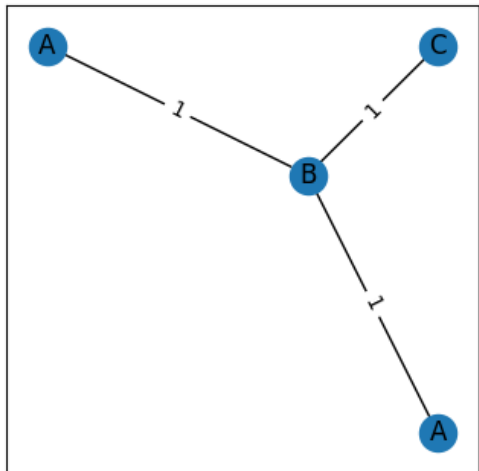
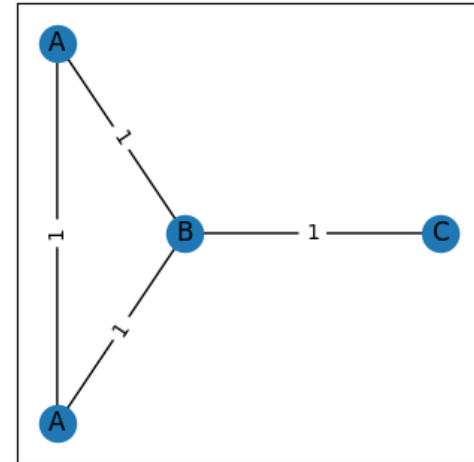
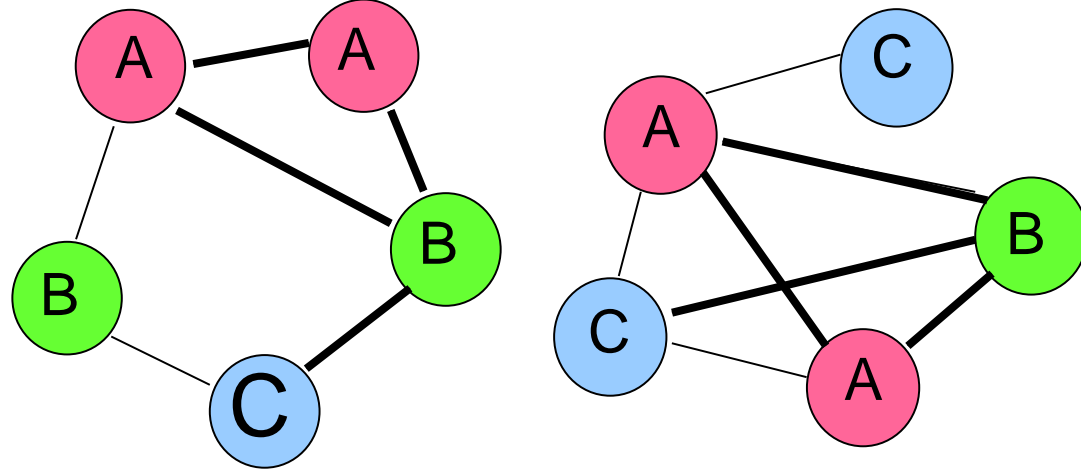
## ➤ Support: 2

```
from gspan_mining.config import parser
from gspan_mining.main import main
```

```
%matplotlib inline
```

```
args_str = '-s 2 -l 3 -p True graphdata/sample_data3'
FLAGS, _ = parser.parse_known_args(args=args_str.split())
```

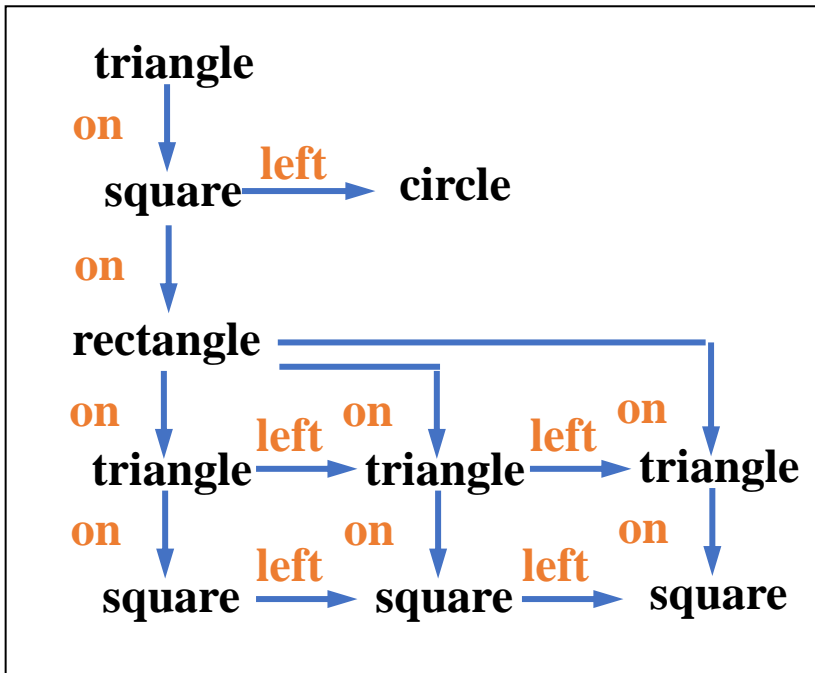
```
gs = main(FLAGS)
print(gs)
```



- A greedy algorithm for finding some of the most prevalent subgraphs.
- This method is not complete
  - i.e. it may not obtain all frequent subgraphs, although it pays in fast execution.
- It discovers substructures that compress the original data and represent structural concepts in the data.
- Based on *Beam Search* - like BFS it progresses level by level. Unlike BFS, however, beam search moves downward only through the best  $W$  nodes at each level. The other nodes are ignored.

- Step 1: Create substructure for each unique vertex label

DB:

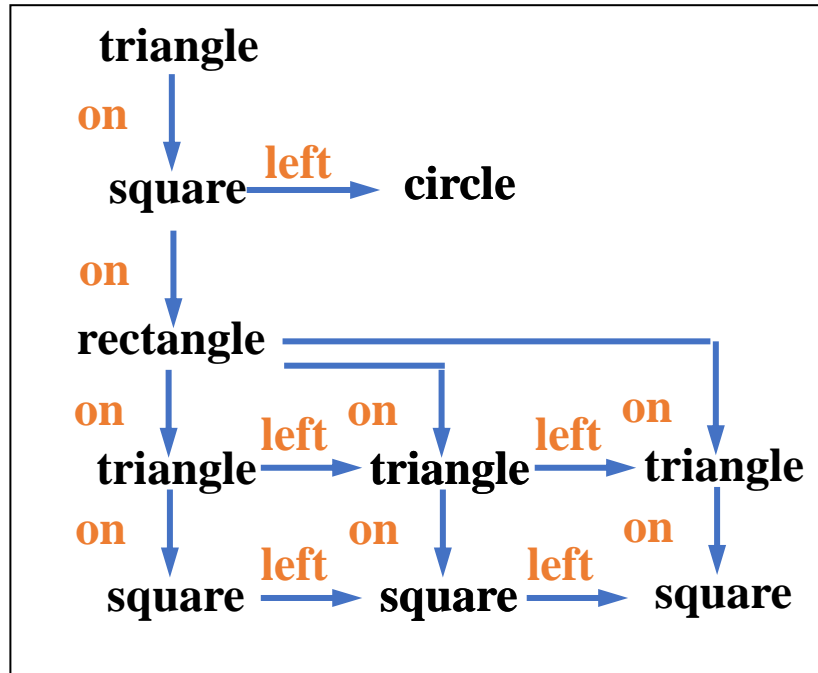


Substructures:

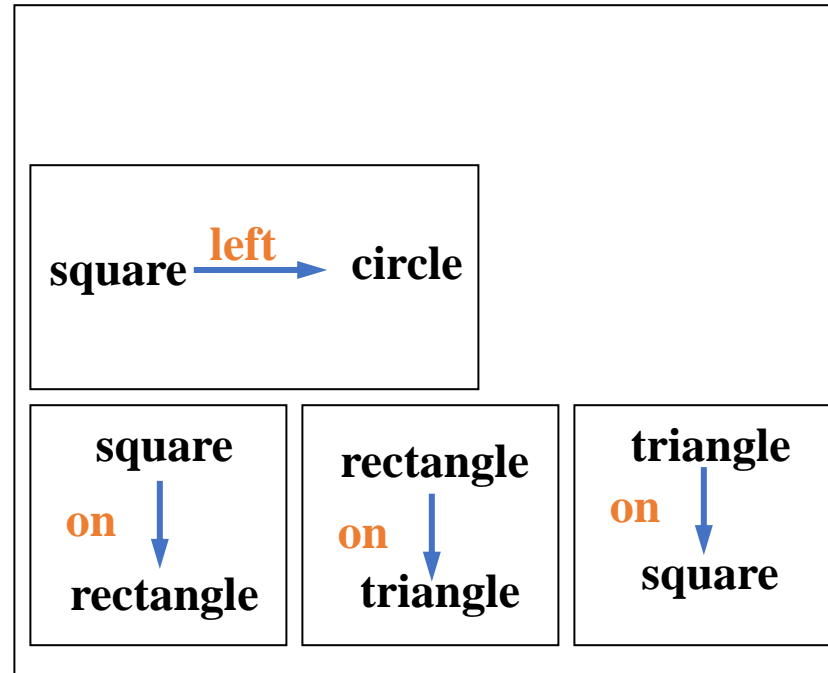
triangle (4)  
square (4)  
circle (1)  
rectangle (1)

- Step 2: Expand best substructure by an edge or edge and neighboring vertex

DB:



Substructures:



- Step 3: Keep only best substructures on queue (specified by beam width).
- Step 4: Terminate when queue is empty or when the number of discovered substructures is greater than or equal to the limit specified.
- Step 5: Compress graph and repeat to generate hierarchical description.

```
import os
import subprocess
import json
import re
import contextlib
import io
import sys
import networkx as nx

from Subdue_mining import Parameters
from Subdue_mining.Subdue import ReadGraph, Subdue, nx_subdue

subdue_example_path = './graphdata/inputgraph.json'
tolerance_pct = 0.1 # mainly due to non-deterministic nature of the algorithm

def subdue_json_to_undirected_nx_graph(subdue_json_path):
    with open(subdue_json_path, 'r') as subdue_json_file:
        subdue_format = json.load(subdue_json_file)

    graph = nx.Graph()
    for vertex_or_edge in subdue_format:
        if list(vertex_or_edge.keys()) == ['vertex']:
            node_attributes_loop = vertex_or_edge['vertex']['attributes']
            graph.add_node(
                vertex_or_edge['vertex']['id'],
                **node_attributes_loop,
            )
        elif list(vertex_or_edge.keys()) == ['edge']:
            edge_attributes_loop = vertex_or_edge['edge']['attributes']
            graph.add_edge(
                u_of_edge=vertex_or_edge['edge']['source'],
                v_of_edge=vertex_or_edge['edge']['target'],
                **edge_attributes_loop,
            )
        else:
            raise ValueError('Invalid entry type')

    return graph

# use networkx graph as an input
subdue_example_graph = subdue_json_to_undirected_nx_graph(subdue_example_path)
capture_prints = io.StringIO()
with contextlib.redirect_stdout(capture_prints):
    result = nx_subdue(graph=subdue_example_graph, verbose=True)
prints_nx_subdue = capture_prints.getvalue()
```

```
import matplotlib.pyplot as plt

# Create a new graph for each List and add nodes and edges
graphs = []
for graph_data in result:
    G = nx.Graph()
    for subgraph_data in graph_data:
        G.add_nodes_from(subgraph_data['nodes'])
        G.add_edges_from(subgraph_data['edges'])
    graphs.append(G)

# Draw the graphs
for i, G in enumerate(graphs):
    plt.figure(i)
    nx.draw(G, with_labels=True)
plt.show()
```







네트워크 과학연구실  
NETWORK SCIENCE LAB



가톨릭대학교  
THE CATHOLIC UNIVERSITY OF KOREA

