Mining Frequent Subgraphs

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

- 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

Why Frequent patterns?

- Frequent pattern: a structure (a set of items, subsequences, 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?

The Apriori principle

- 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.

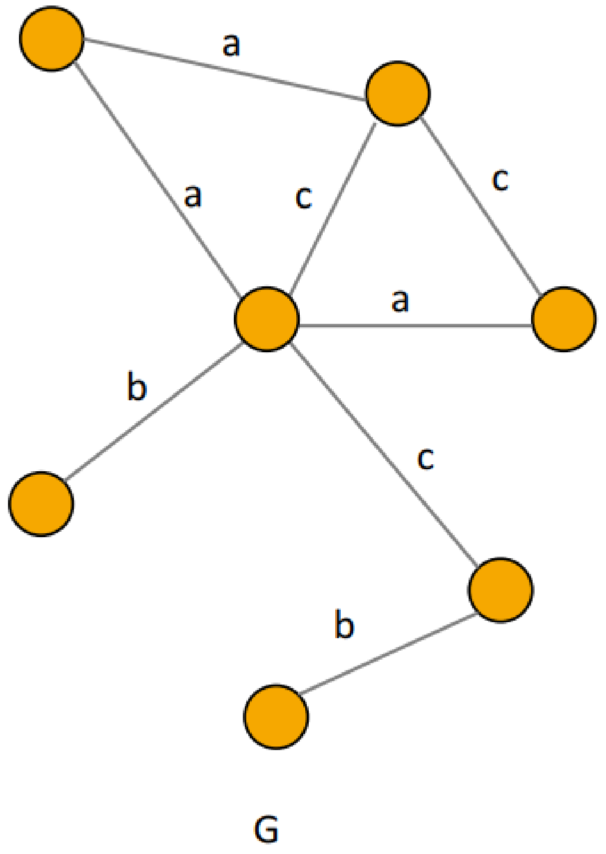
Graph Pattern Mining

- 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

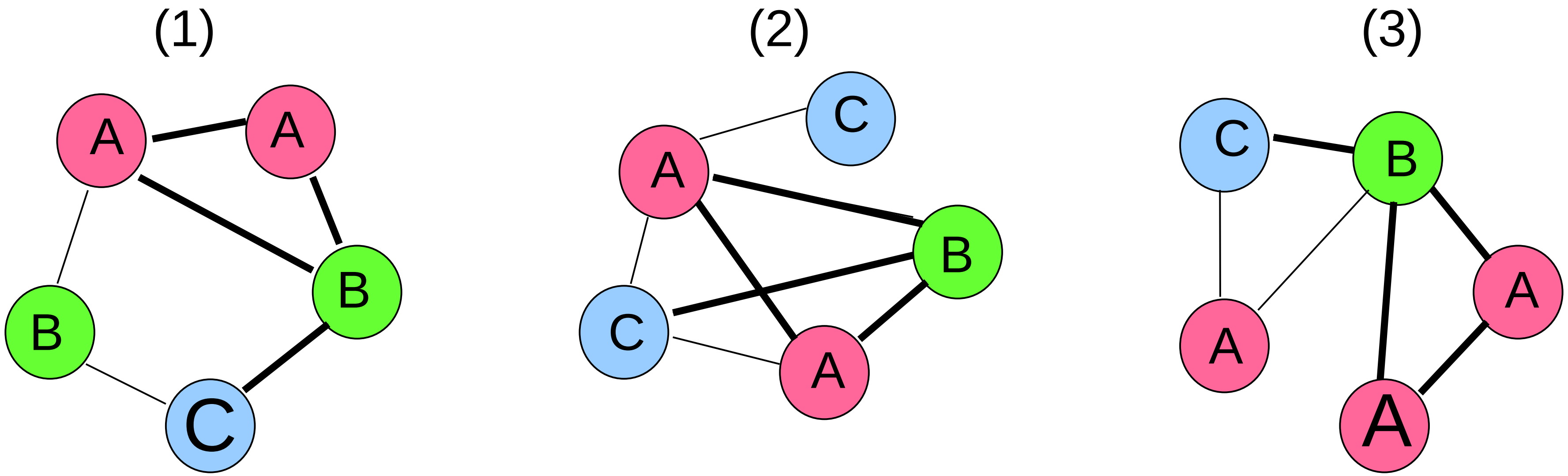
- Problem: Find all subgraphs of G that appear at least t times

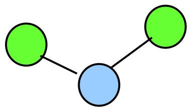
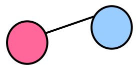
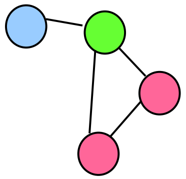
- Suppose t = 2, the frequent subgraphs are (only edge labels)
  - a, b, c
  - a-a, a-c, b-c, c-c
  - a-c-a …

- Exponential number of patterns!



Frequent Subgraph Example

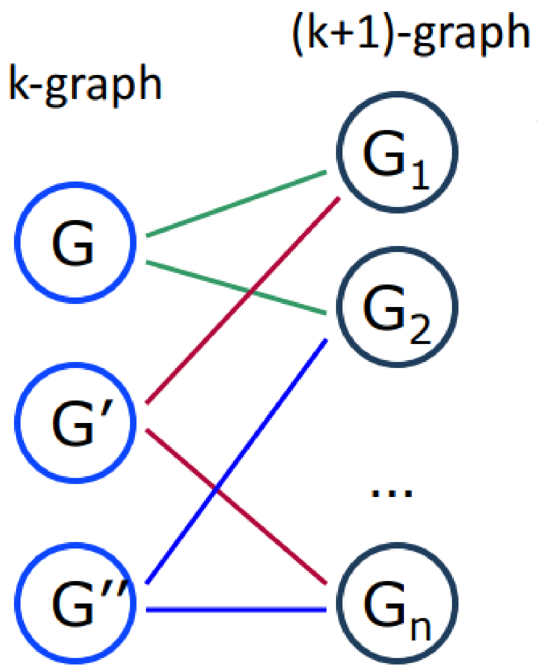


Support	1	3	3
Subgraph			

How to mine frequent subgraphs?

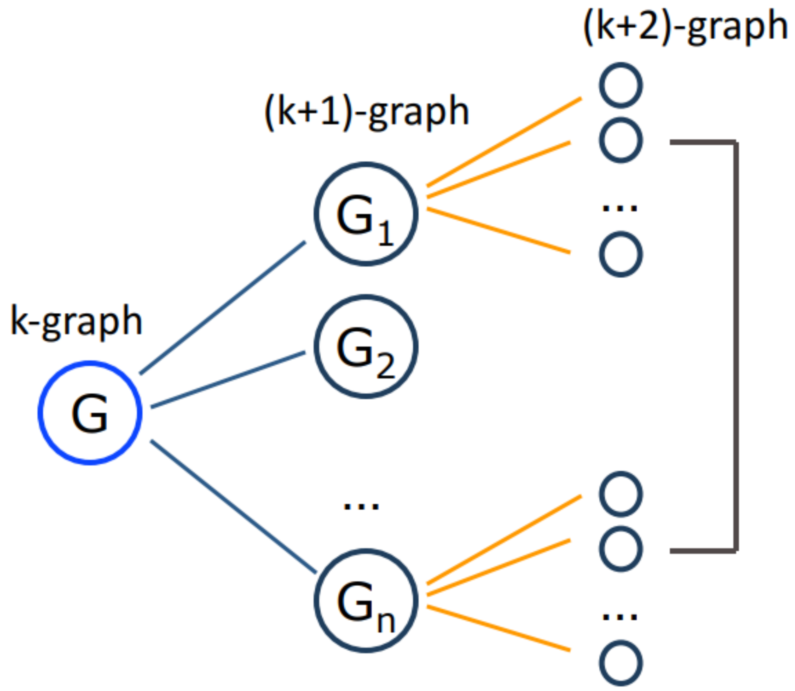
- Apriori-based approaches
  - Start with small-size subgraphs and proceeds in a bottom-up manner
  - Join two patterns to create bigger size patterns (through Apriori principle)
  - Several approaches
    - FSG
    - PATH#
- Pattern-growth approaches
  - Extends existing frequent graphs by adding one 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

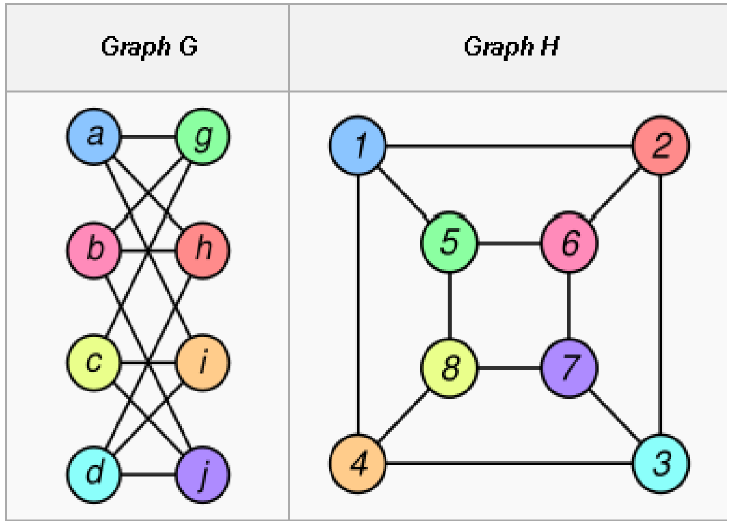
- Generate patterns expanding existing ones
- Extends existing frequent graphs by adding one edge



- Problems
  - Duplicate graphs

Key Challenges in Subgraph Mining

- Graph isomorphism
  - To detect if two graphs are identical in structure
- 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.
- Subgraph candidate generation
  - Generate candidate frequent subgraphs from datasets



➢ Graph isomorphism: sample code

➢ 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
```

➢ Graph isomorphism: sample code

➢ 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)
```

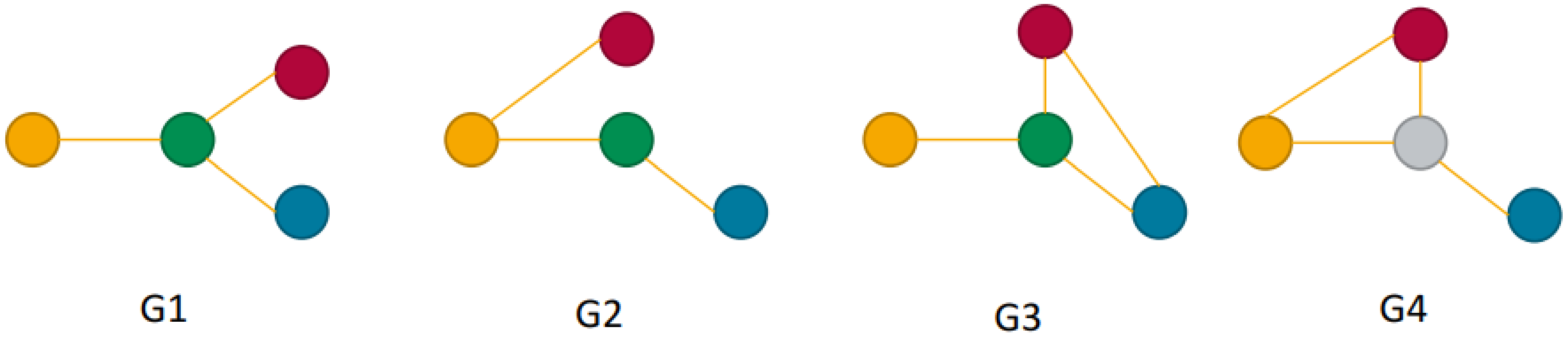
```
True
```

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

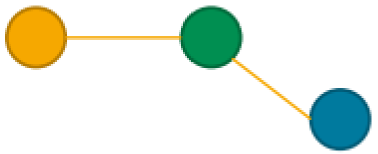
```
False
```

Graph Pattern Mining - Set of graphs

- Given a set of 4 graphs:

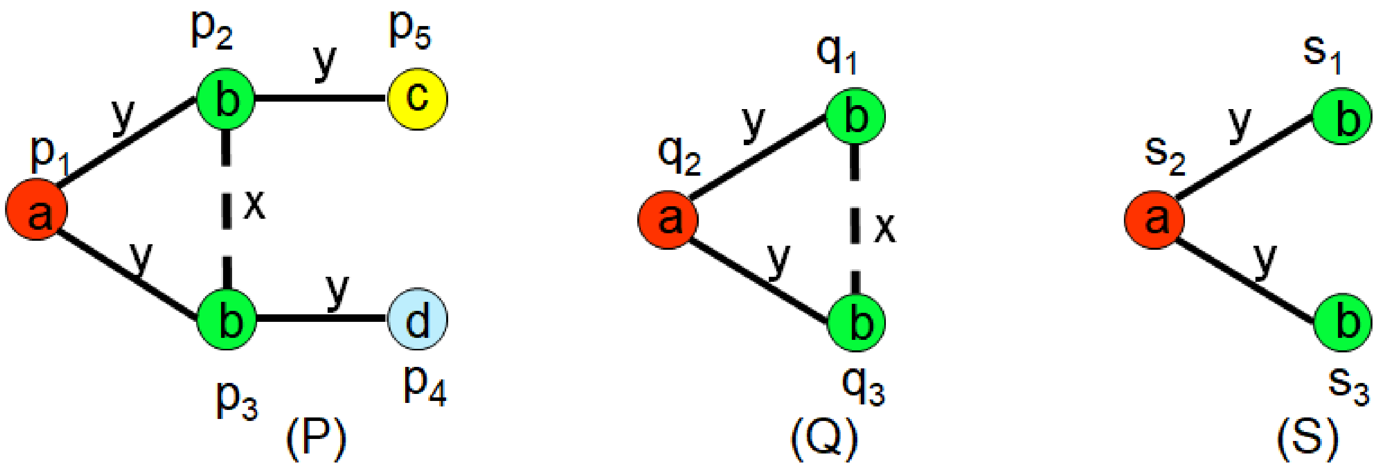


- Support: frequency of a subgraph appearing in a set of graphs
- Frequent subgraph Min support = 3/4



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,
  - Σ_V (Σ_E) are set of vertex (edge) labels,
  - δ is the labeling function: $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$ that maps vertices and edges to their labels.



Subgraph Mining problem

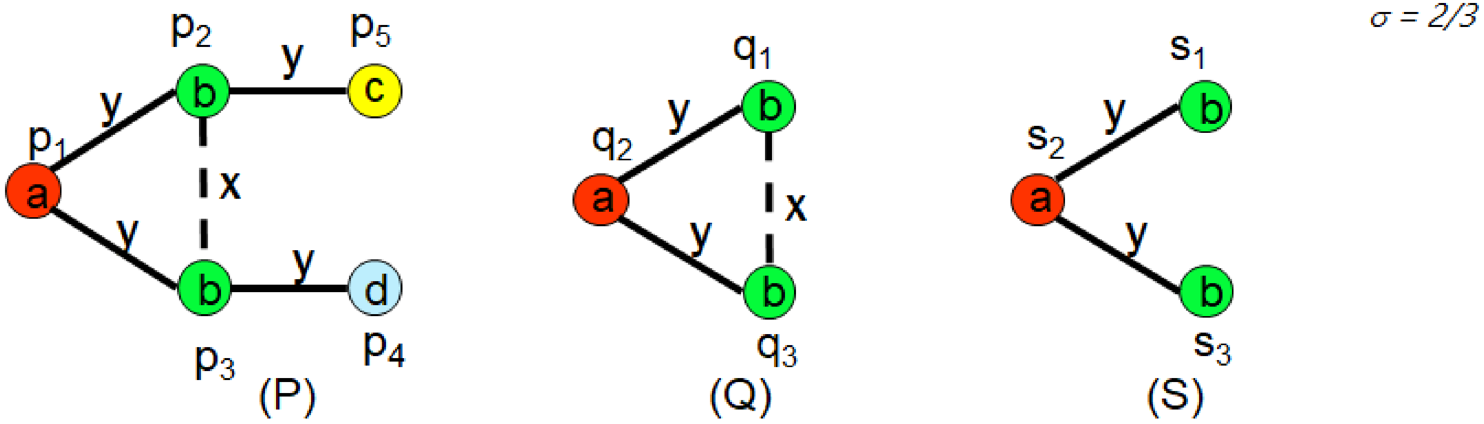
- Support: Given a set of labeled graphs:
  - D = $\{G_1, G_2, \ldots, G_n\}, G_i = \langle V_i, E_i, \ell_i \rangle$
  - A subgraph G.
- The supporting set of G is: $D_G = \{G_i | G \sqsubseteq G_i, G_i \in D\}$

  Where $G \sqsubseteq G_i$ indicates that G is subgraph isomorphic to Gi

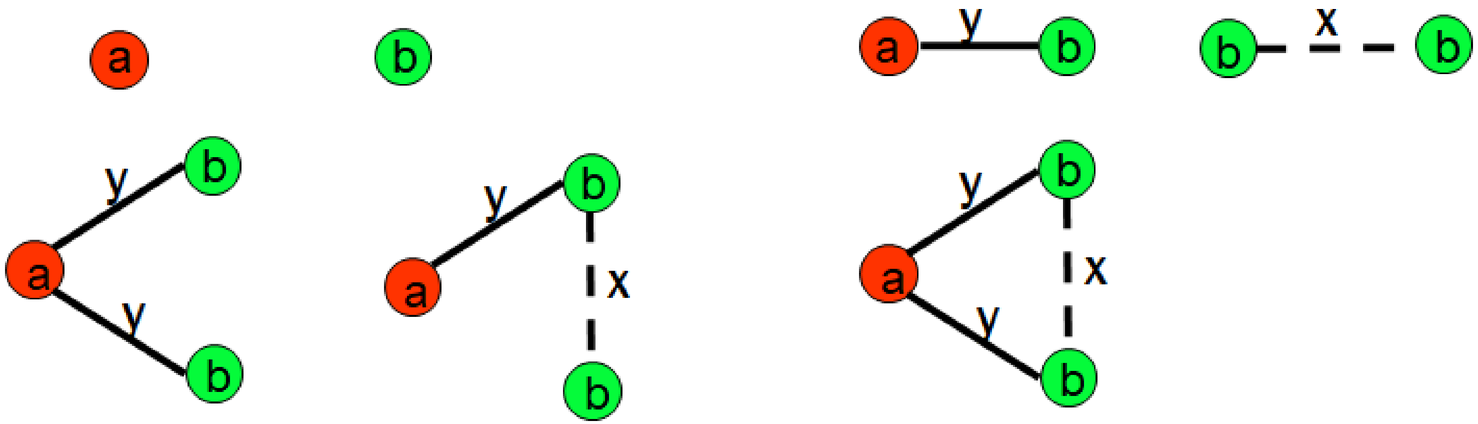
- The support is defined as:

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

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



- All frequent subgraphs (w. r. t. σ) from *Graph data* .



Subgraph Mining problem: Input and Output

- 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 min_sup
- Output:
  - A subgraph G is frequent if $\sigma(G) \geq \min\_sup$
  - Each subgraph is connected.

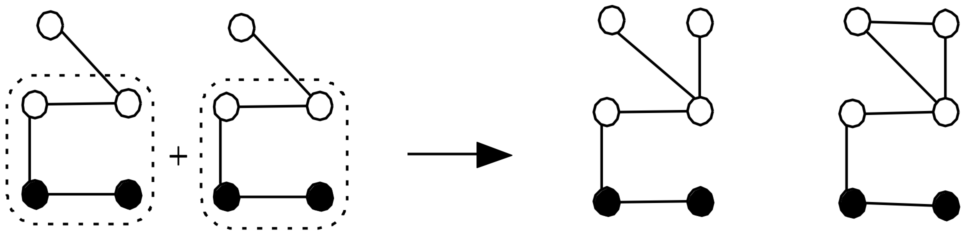
Mining approaches

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

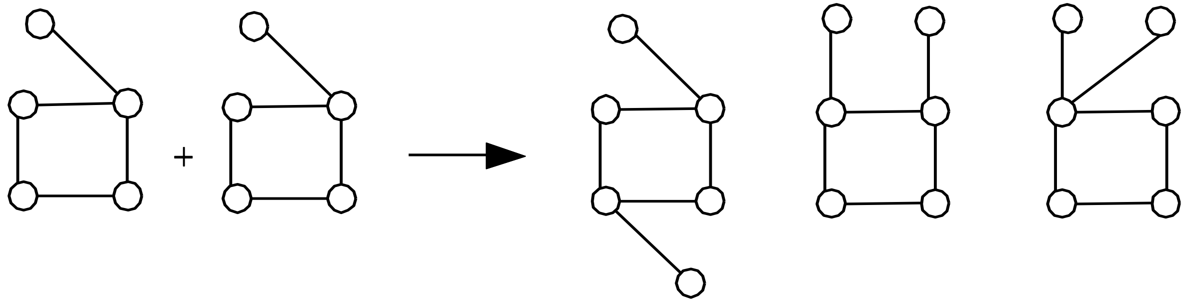
Apriori-Based, Breadth-First Search

Methodology: breadth-search, joining two graphs

- 1. Generates new graphs with one more node



- 2. Generates new graphs with one more edge



FSG Algorithm

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

FSG Algorithm

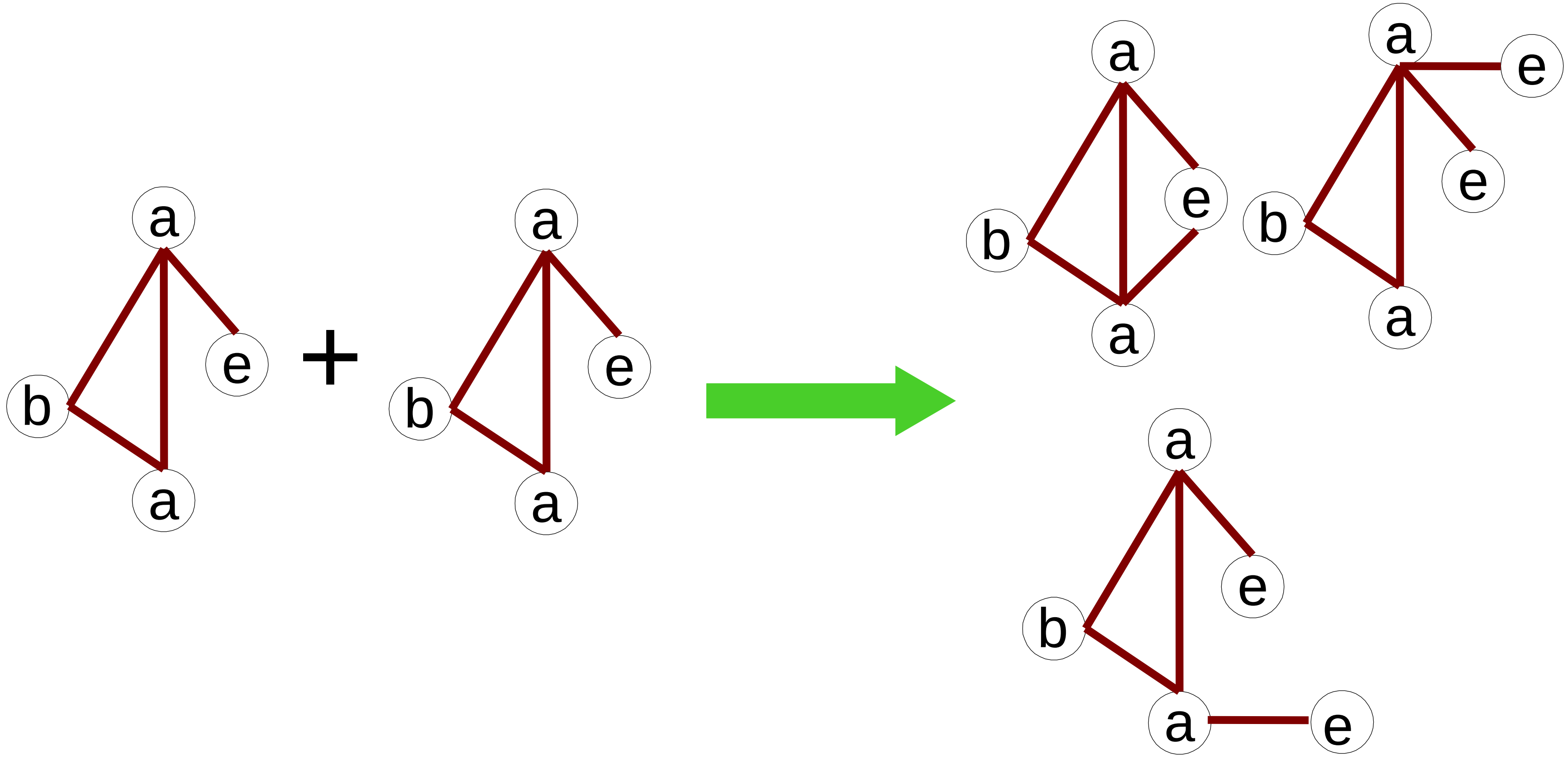
- $K = 1$
- F_1 = all frequent edges
- Repeat
  - $K = K + 1$;
  - C_K = join(F_{K-1})
  - F_K = frequent patterns in C_K
  - Until F_K is empty

Join: Key Operation

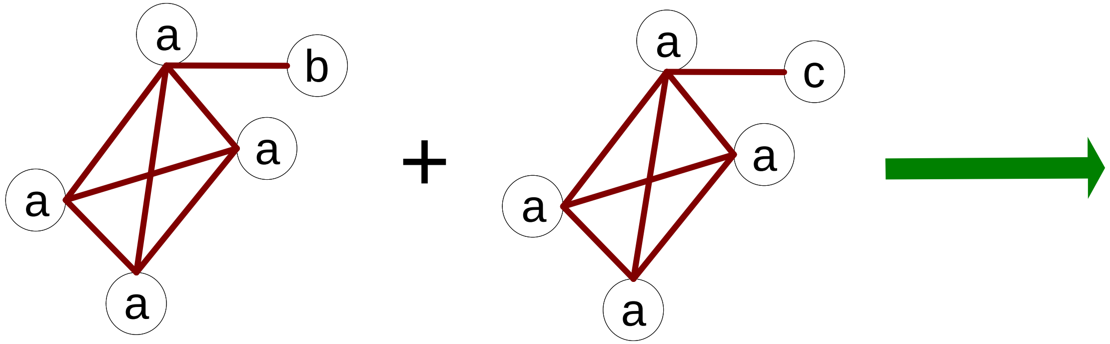
- Join(L) = $\cup$ join(P, Q) for all P, Q $\in$ L
- Join(P, Q) = {G | P, Q, $\subset$ G, |G| = |P| + 1, |P| = |Q|}
- Two graphs P and Q are ***joinable*** if the join of the two graphs produces an non-empty set
- Theorem: two graphs P and Q are joinable if P ∩ Q is a graph with size |P| -1 or share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

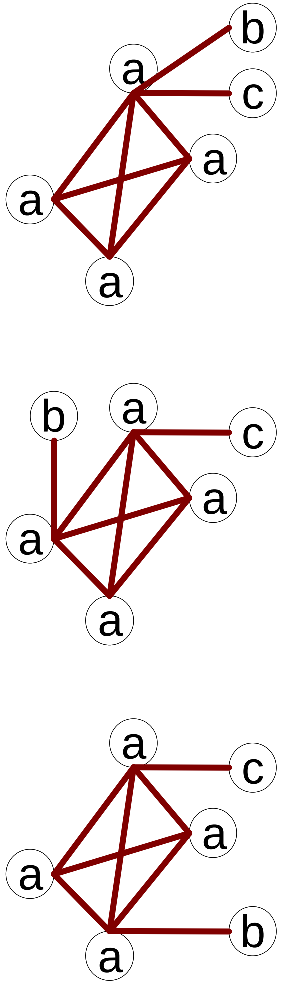
Multiplicity of Candidates

Case 1: identical node labels



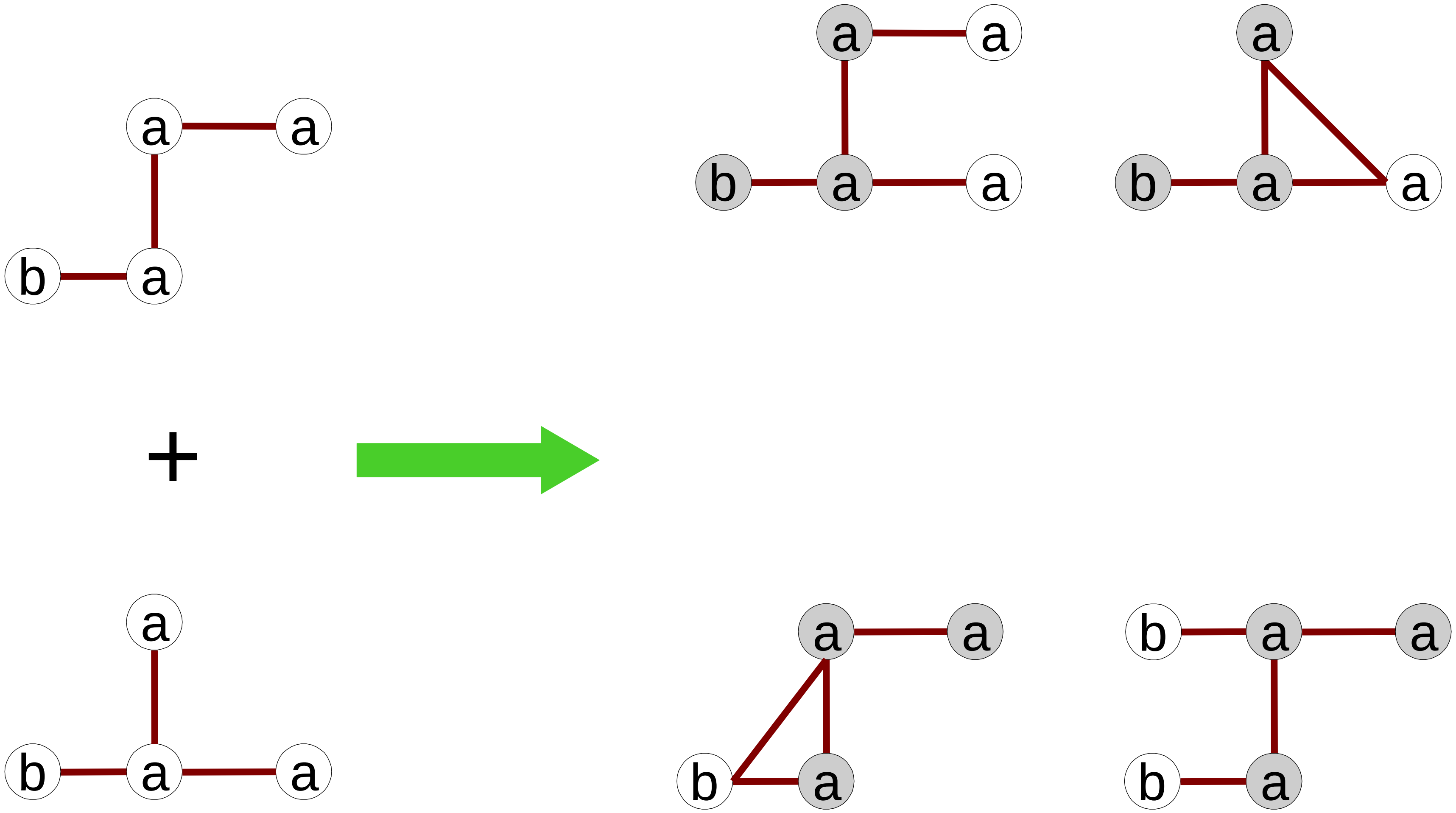
Case 2: Core contains identical labels





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

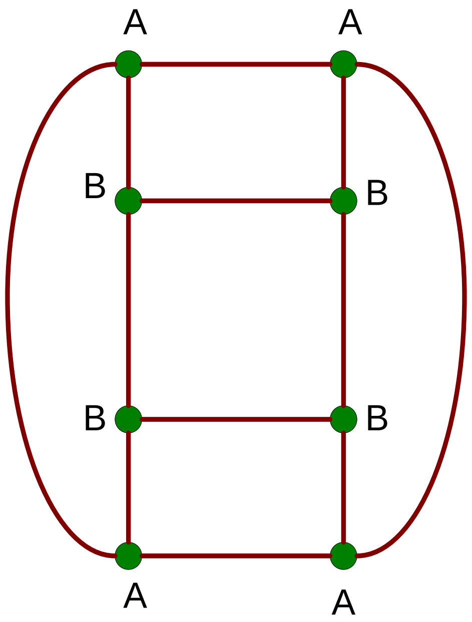
Case 3: Core multiplicity

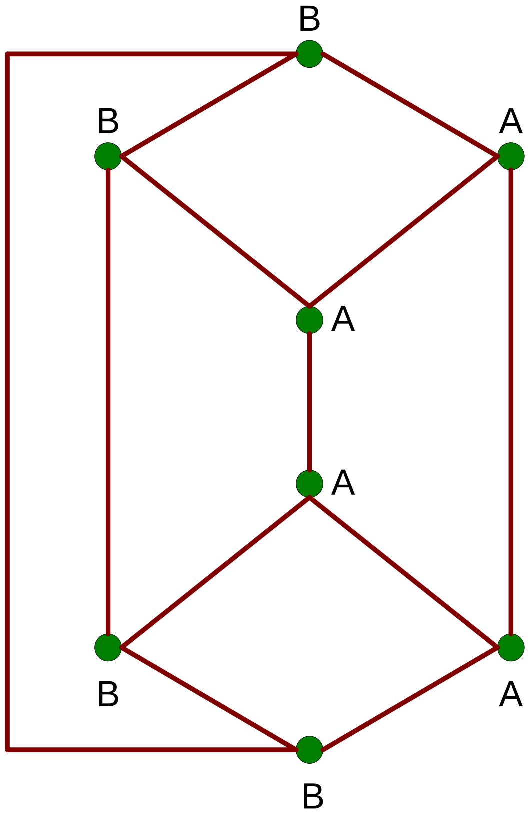


Challenges

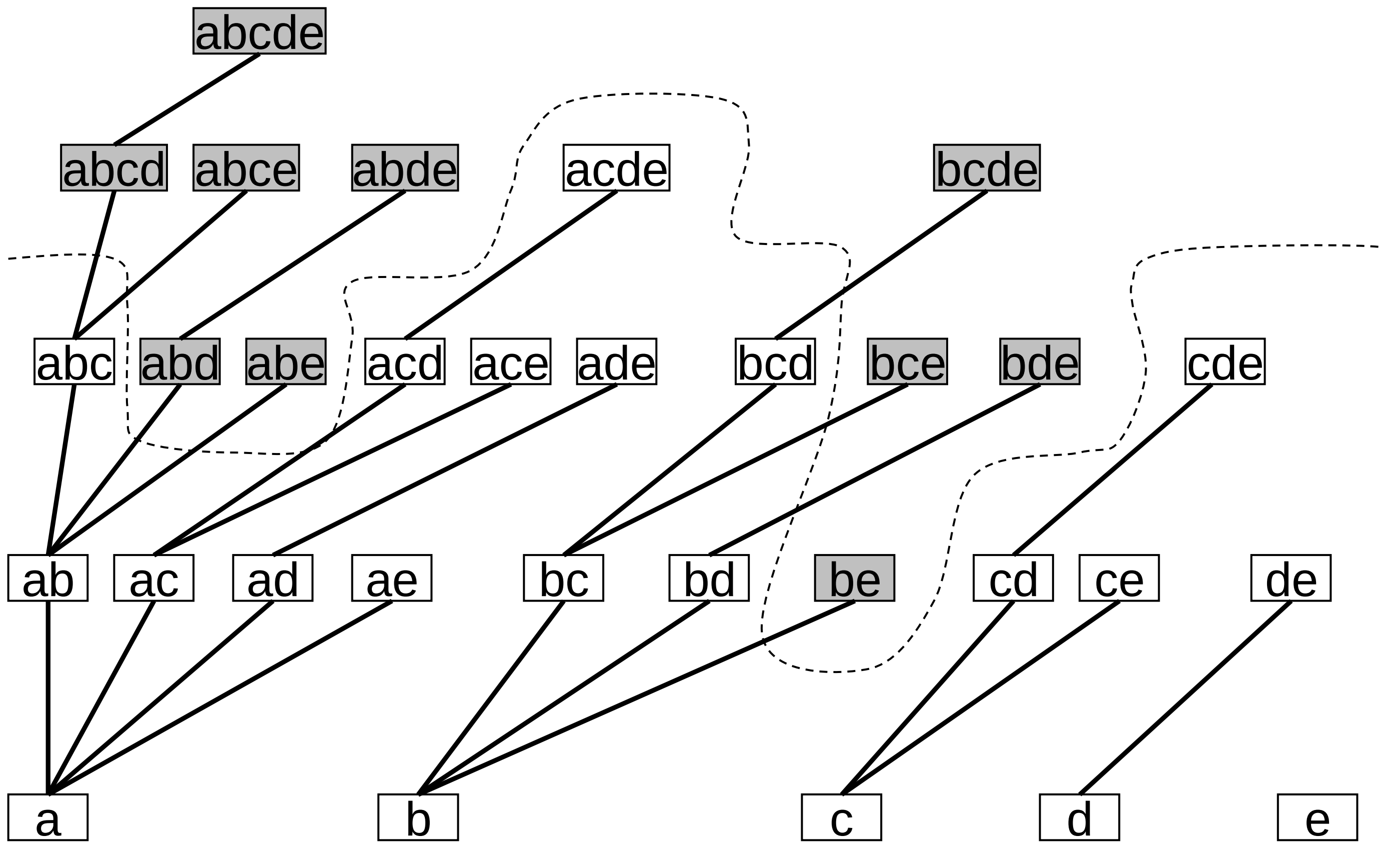
- Graph isomorphism
  - Two graphs may have the same topology though their layouts are different
- Subgraph isomorphism
  - How to compute the support value of a pattern

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





- Itemset search space – prefix based



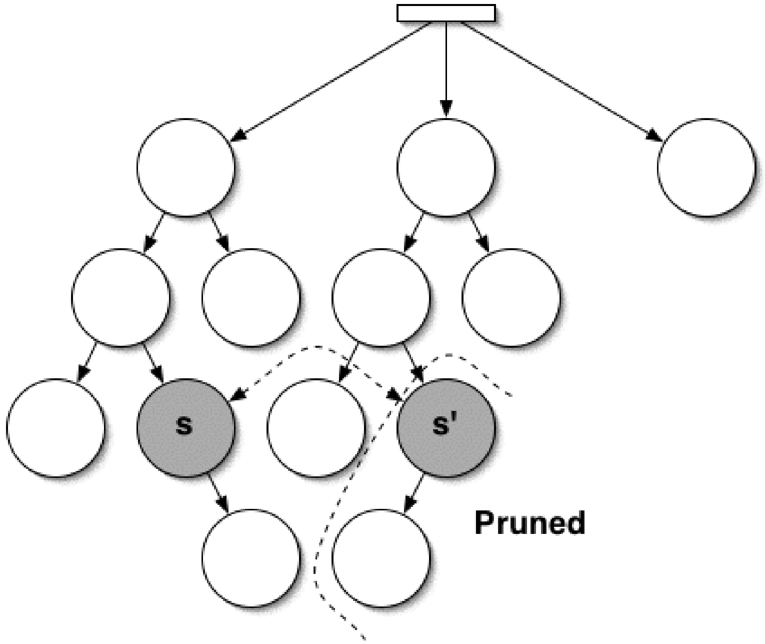
Motivation for Tree Search

- 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.

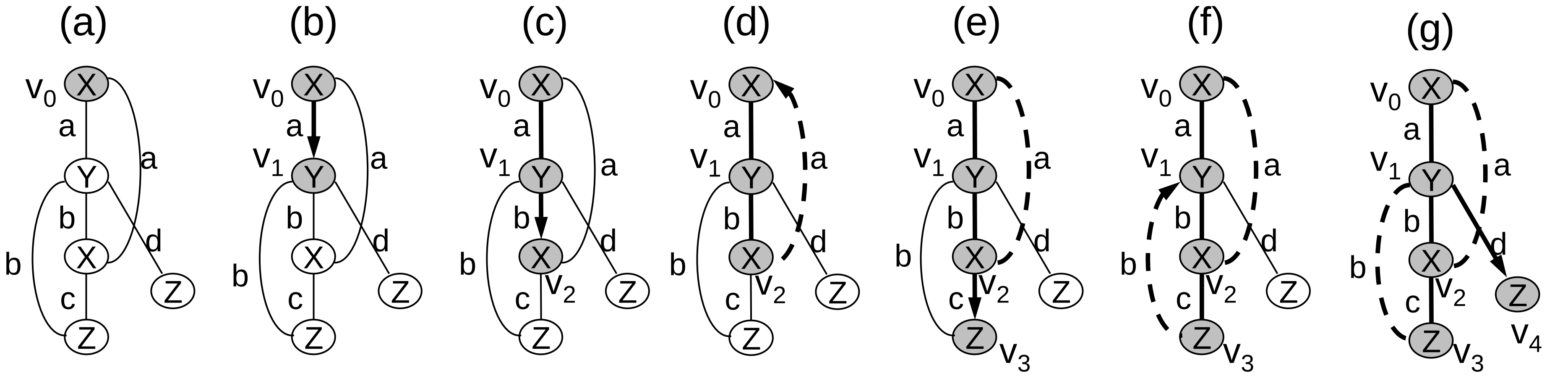
DFS Code Trees

- 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.



DFS Code construction

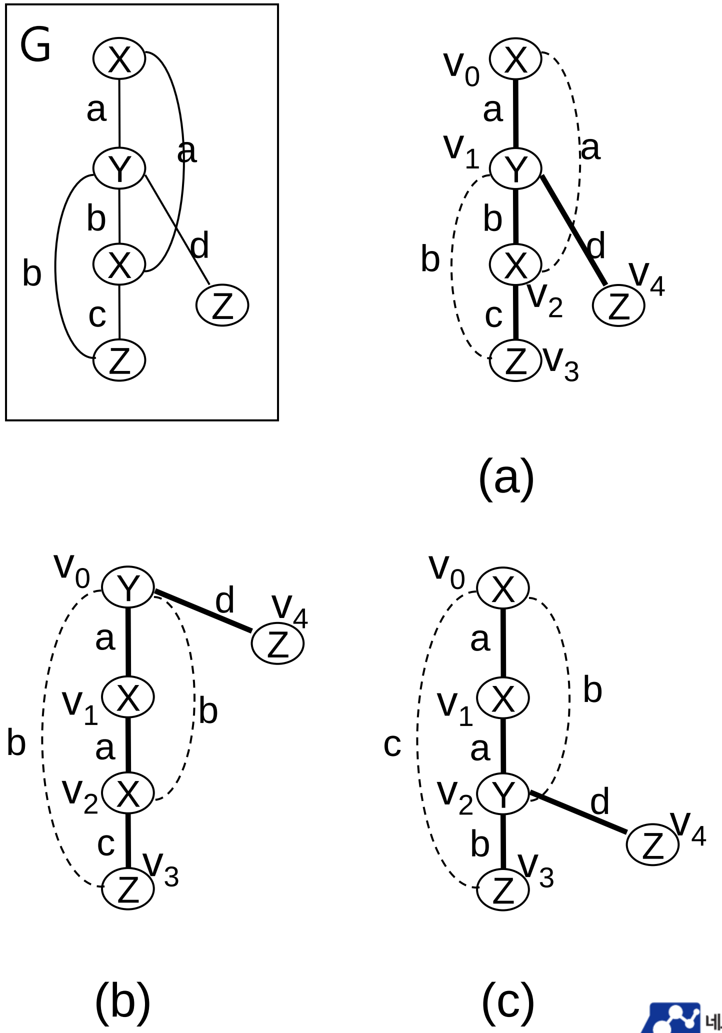
- Given a graph G. for each Depth First Search over graph G, construct the corresponding DFS-Code.



(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)

Single graph, several DFS-Codes

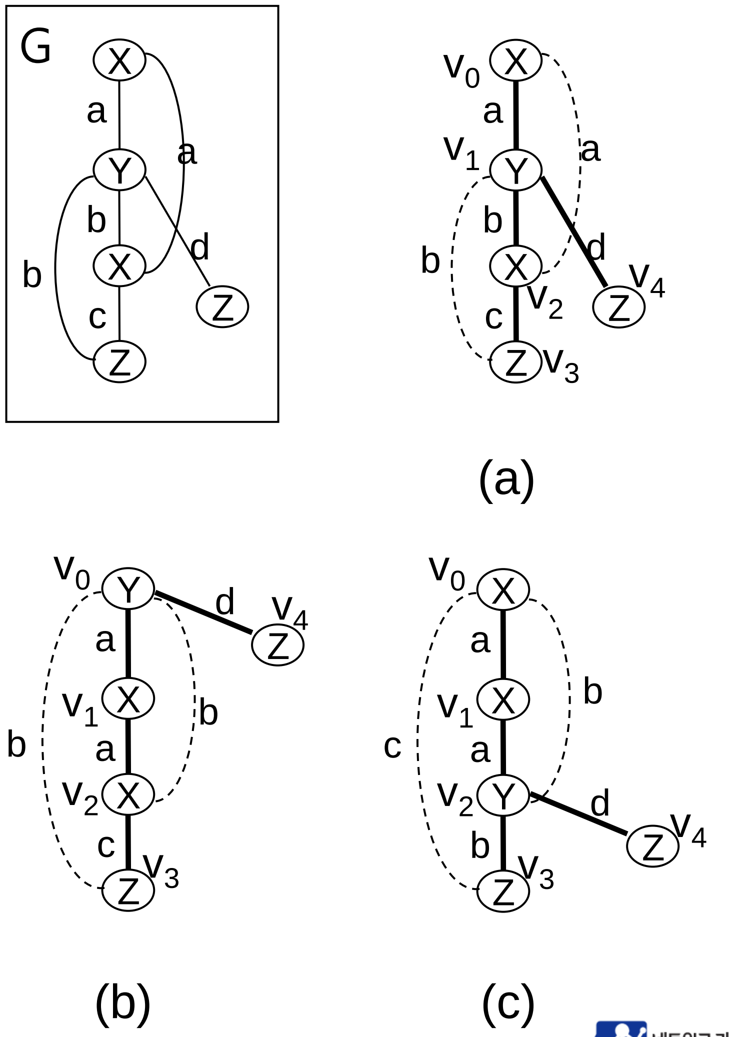
	(a)	(b)	(c)
1	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, X, a, X)
2	(1, 2, Y, b, X)	(1, 2, X, a, X)	(1, 2, X, a, Y)
3	(2, 0, X, a, X)	(2, 0, X, b, Y)	(2, 0, Y, b, X)
4	(2, 3, X, c, Z)	(2, 3, X, c, Z)	(2, 3, Y, b, Z)
5	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	(3, 0, Z, c, X)
6	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	(2, 4,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



Single graph - single Min DFS-code

Min 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)**



Minimum DFS-Code

- 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:

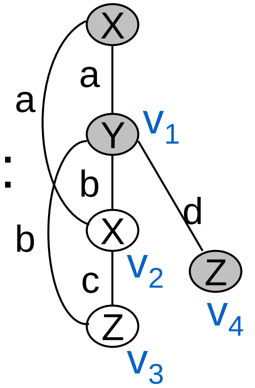
$$min(A) = min(B)$$

DFS-Code Tree: parent-child relation

- If $\min(G_1) = \{ a_0, a_1, \ldots.., a_n\}$
  and $\min(G_2) = \{ a_0, a_1, \ldots.., a_n, b\}$
  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).

DFS-Code Tree: parent-child relation

Graph G_1:

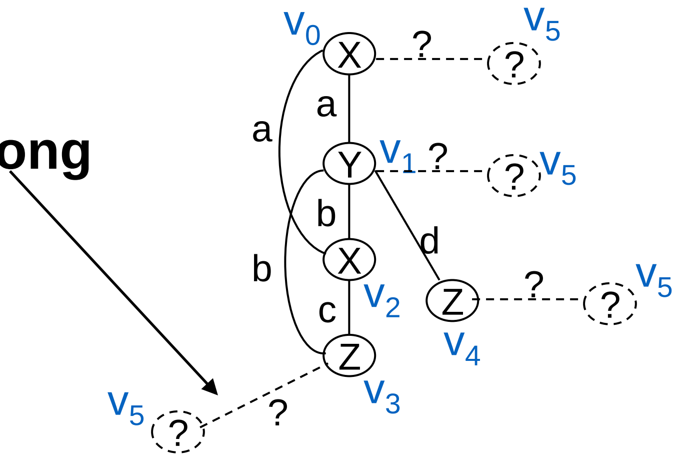


Min(g) = (0,1,X,a,Y) (1,2,Y,b,X) (2,0,X,a,X) (2,3,X,c,Z) (3,1,Z,b,Y) (1,4,Y,d,Z)

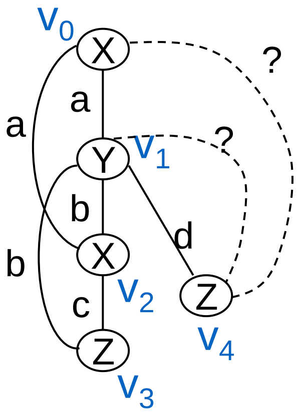
A child of graph G_1 *must grow edge from rightmost path of* G_1 (necessary condition)

Graph G_2:

wrong



Forward edge

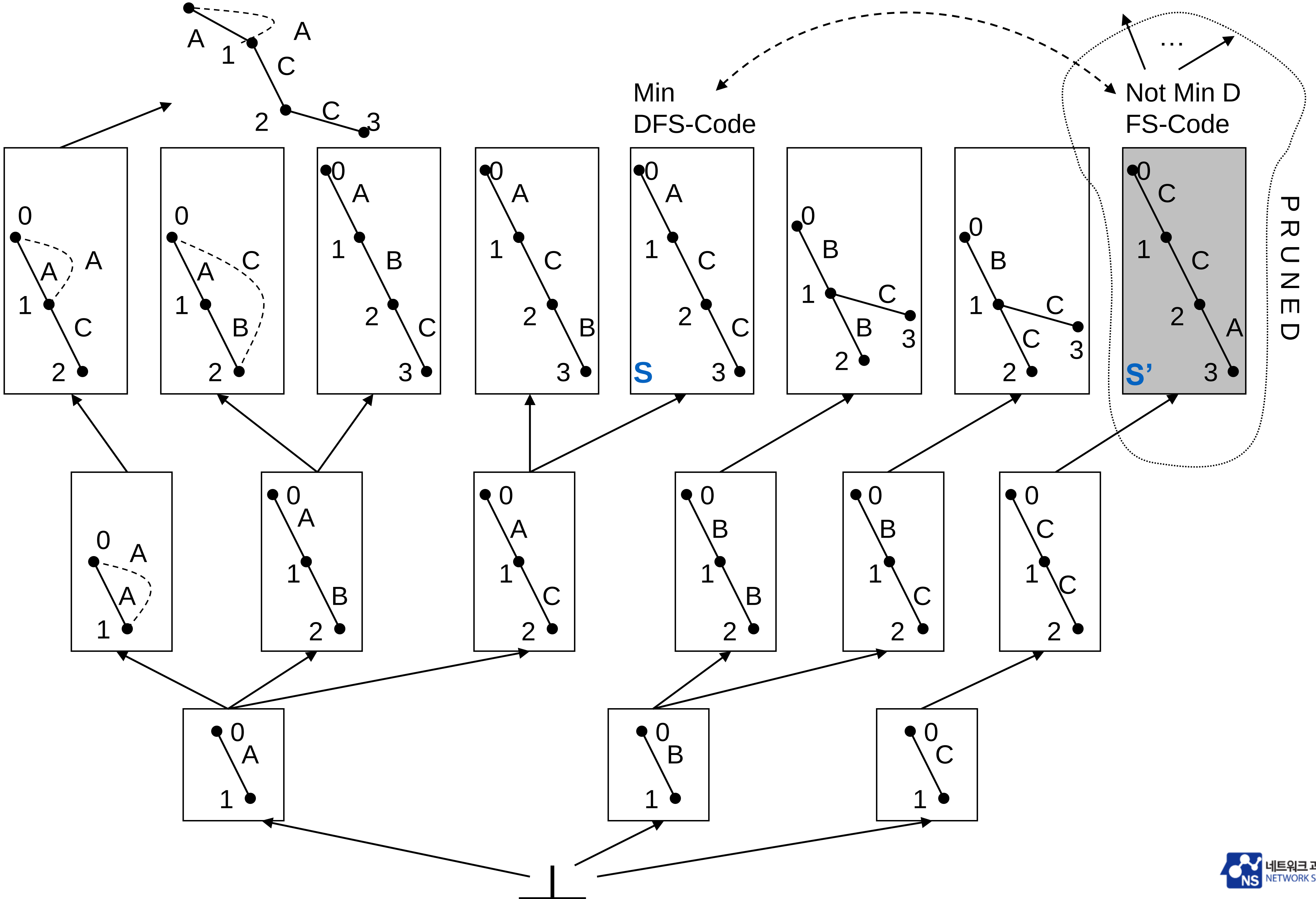


Backward edge

Search space: DFS code Tree

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

Search space: DFS code Tree



Tree pruning

- 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.

gSpan(D, F, g)

1: if $g \neq \min(g)$

return;

2: $F \leftarrow F \cup \{ g \}$

3: children(g) $\leftarrow$ [generate all g' potential children with one edge growth]

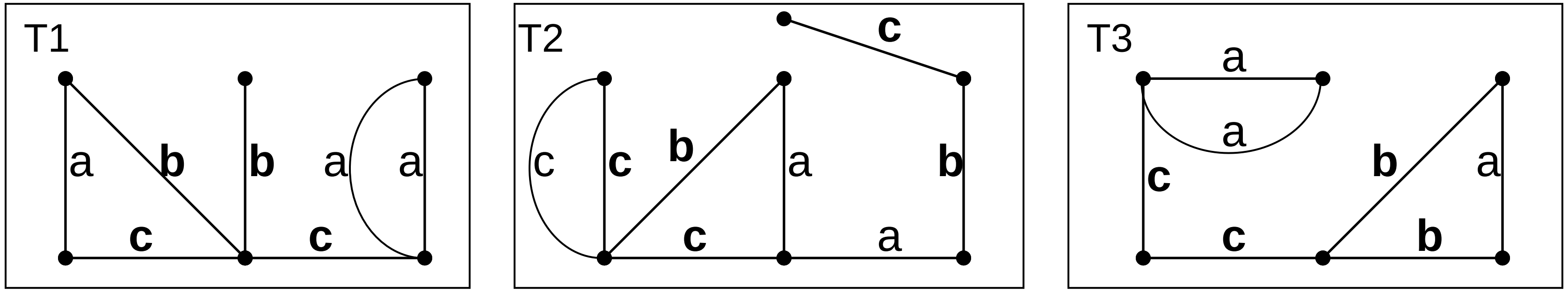
4: Enumerate(D, g, children(g))

5: for each $c \in$ children(g)

if support(c) $\geq$ #minSup

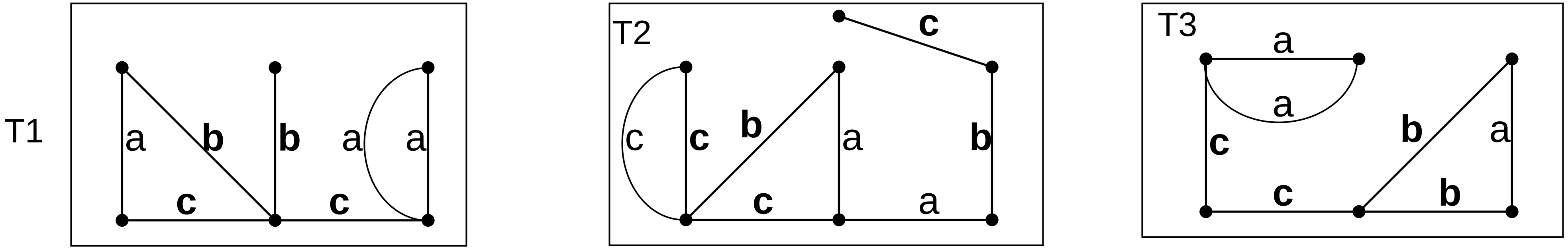
SubgraphMining (D, F, c)

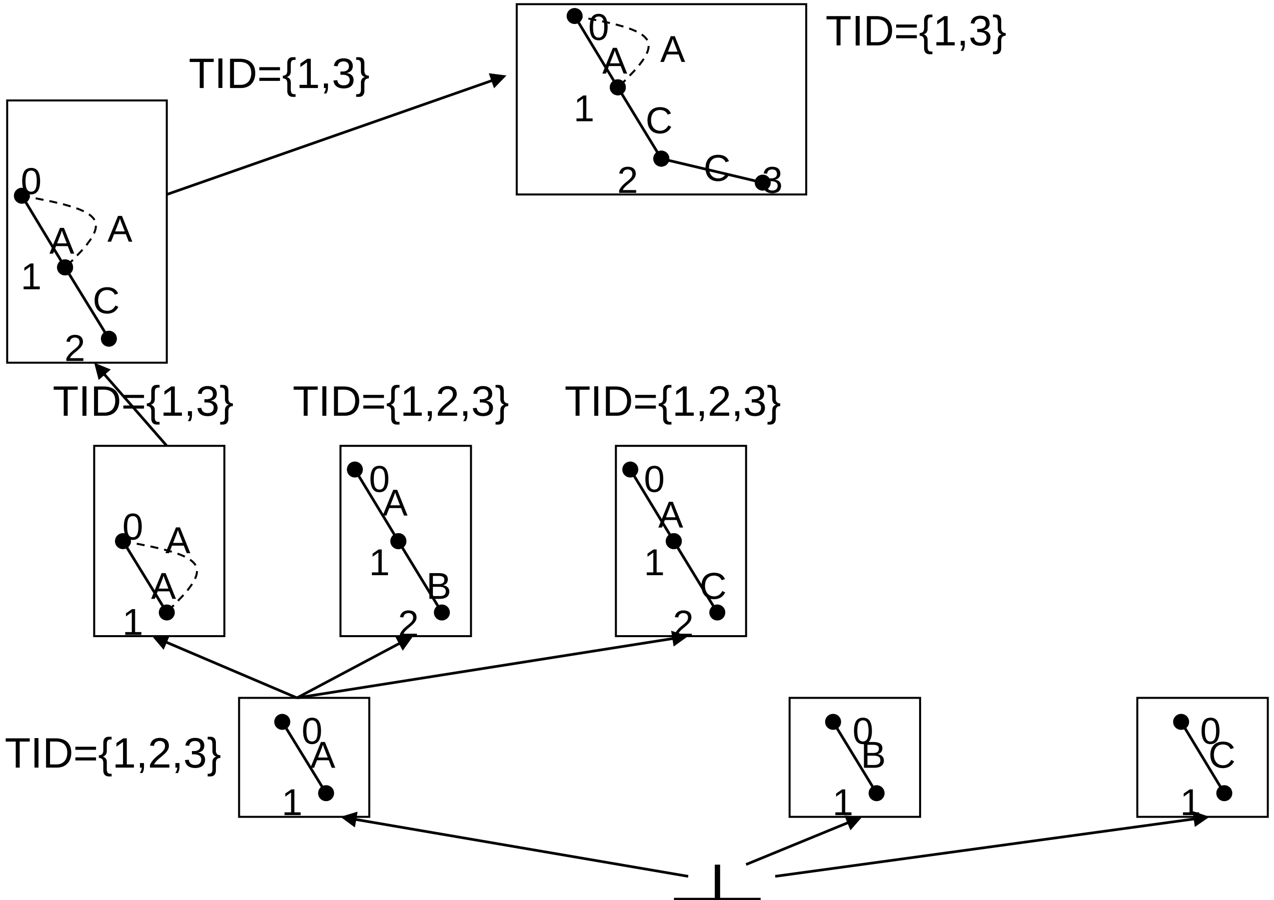
Given: database *D*



Task: Mine all frequent subgraphs with support ≥ 2 (#minSup)

Example





- Support: 2

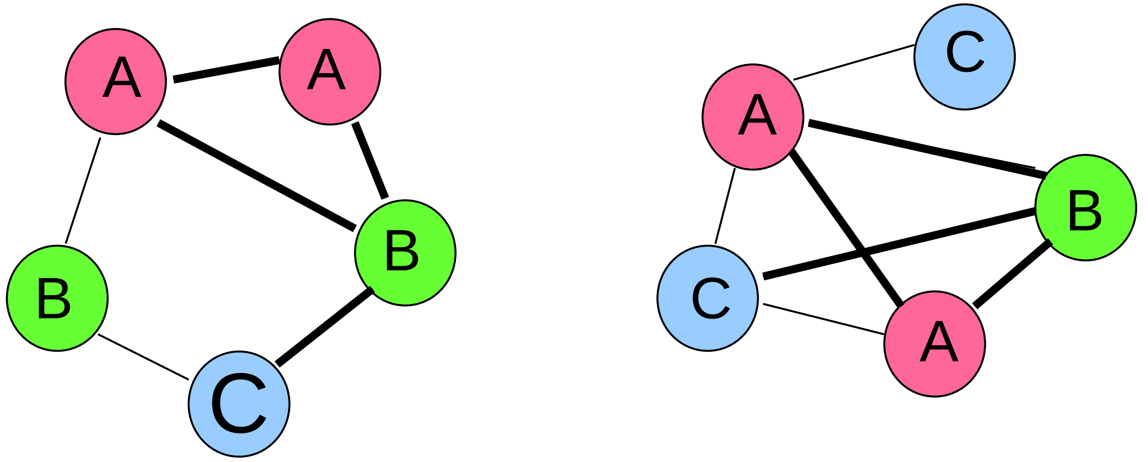
```
from gspan_mining.config import parser
from gspan_mining.main import main
```

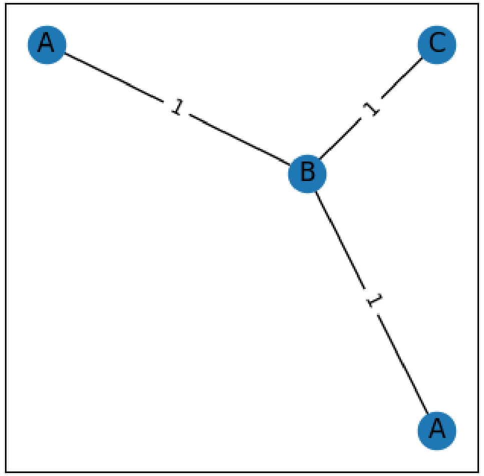
```
%pylab inline
```

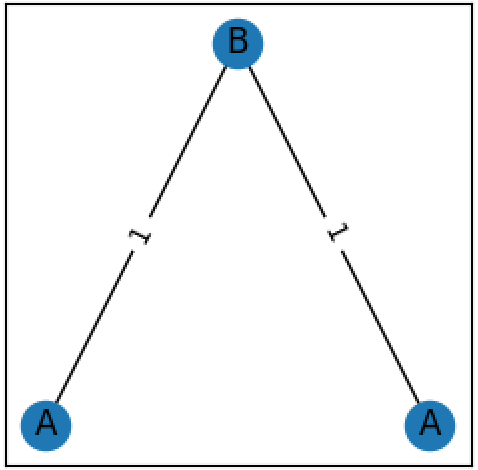
```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
```

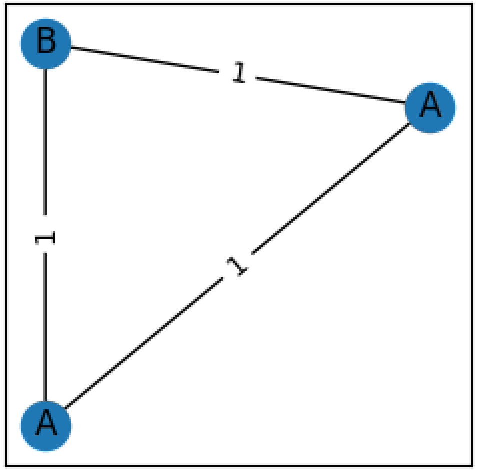
```
args_str = '-s 2 -l 3 -p True ./graphdata/sample_data3'
FLAGS, _ = parser.parse_known_args(args=args_str.split())
```

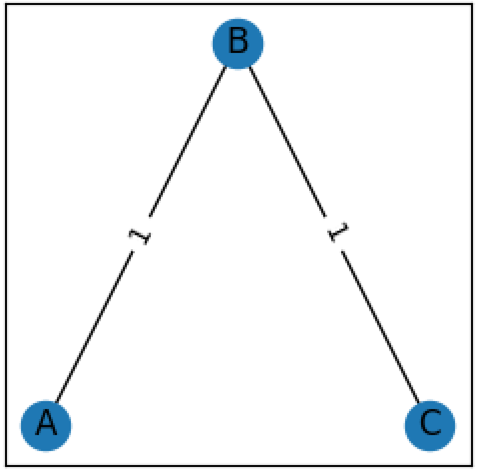
```
gs = main(FLAGS)
print(gs)
```

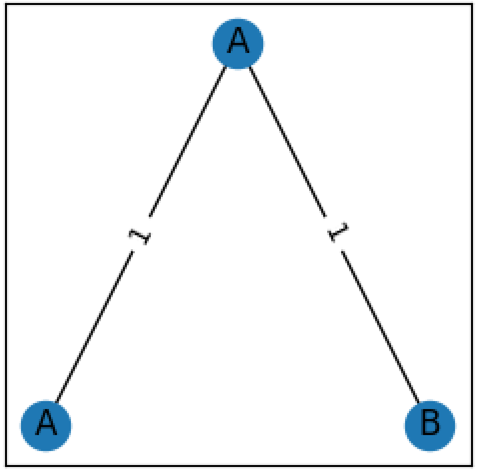












Subdue Algorithm

- 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.

Subdue Algorithm

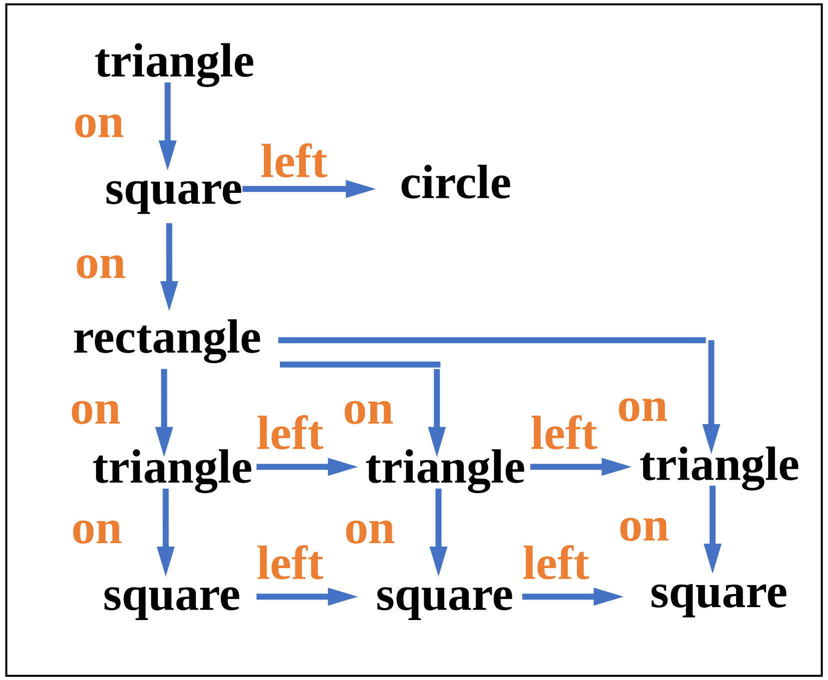
- It discovers substructures that compress the original data and represent structural concepts in the data.

- 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.

Subdue Algorithm: step 1

- Step 1: Create substructure for each unique vertex label

DB:

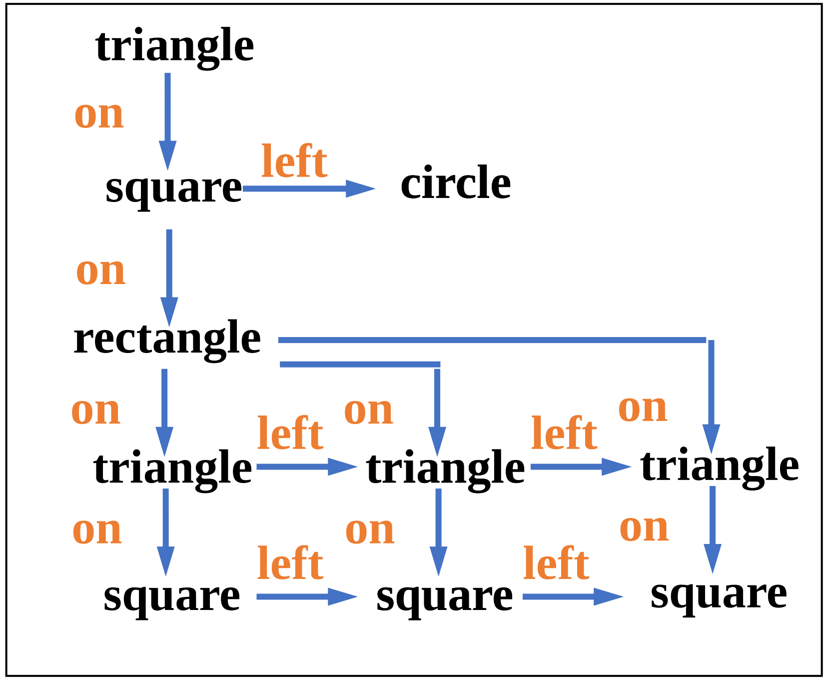


Substructures:

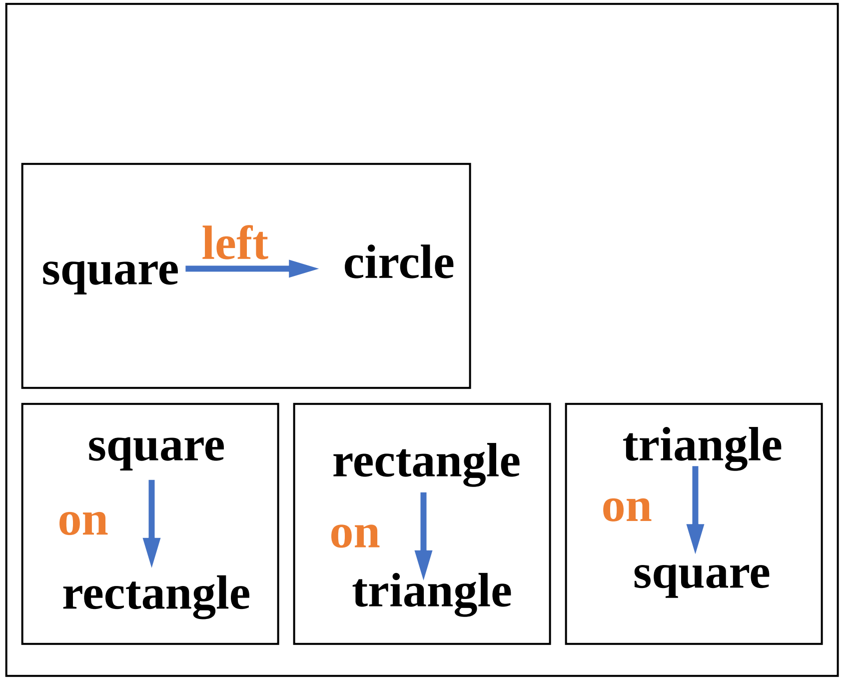
triangle (4)
square (4)
circle (1)
rectangle (1)

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

DB:



Substructures:



Subdue Algorithm: steps 3-5

- 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.

Q
A