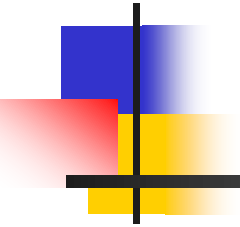



Graph and Web Mining - Motivation, Applications and Algorithms - Chapter 2



Prof. Ehud Gudes
Department of Computer Science
Ben-Gurion University, Israel



Outline

- Basic concepts of Data Mining and Association rules
 - Apriori algorithm
 - Sequence mining
- Motivation for Graph Mining
- Applications of Graph Mining
- Mining Frequent Subgraphs - Transactions 
 - BFS/Apriori Approach (FSG and others)
 - DFS Approach (gSpan and others)
 - Diagonal Approach
 - Constraint-based mining and new algorithms
- Mining Frequent Subgraphs – Single graph
 - The support issue
 - The Path-based algorithm



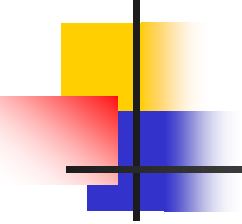
Problem Statement:

Transaction Setting

- **Input:** (D , minSup)
 - Set of labeled-graphs transactions $D = \{T_1, T_2, \dots, T_N\}$
 - Minimum support minSup
- **Output:** (All frequent subgraphs)
 - A subgraph is frequent if it is a subgraph of at least $\text{minSup} \cdot |D|$ (or $\# \text{minSup}$) different transactions in D
 - Each subgraph is connected
- **Notation:** $k\text{-subgraph}$ is a graph with k edges
- Note, the number of occurrences within a single graph is not important if it is > 0 !

Problem Statement

(single graph setting)



- ***Input:*** (D , $minSup$)
 - A single graph D (e.g., the Web or DBLP or an XML file)
 - Minimum support $minSup$
- ***Output:*** (All frequent subgraphs)
 - A subgraph is frequent if the support function of its occurrences in D is above an **admissible** support measure
- ***Definition of an admissible support measure?***
- ***The intuitive definition – number of occurrences is wrong! – we'll see later***



Graph Mining: Transaction Setting

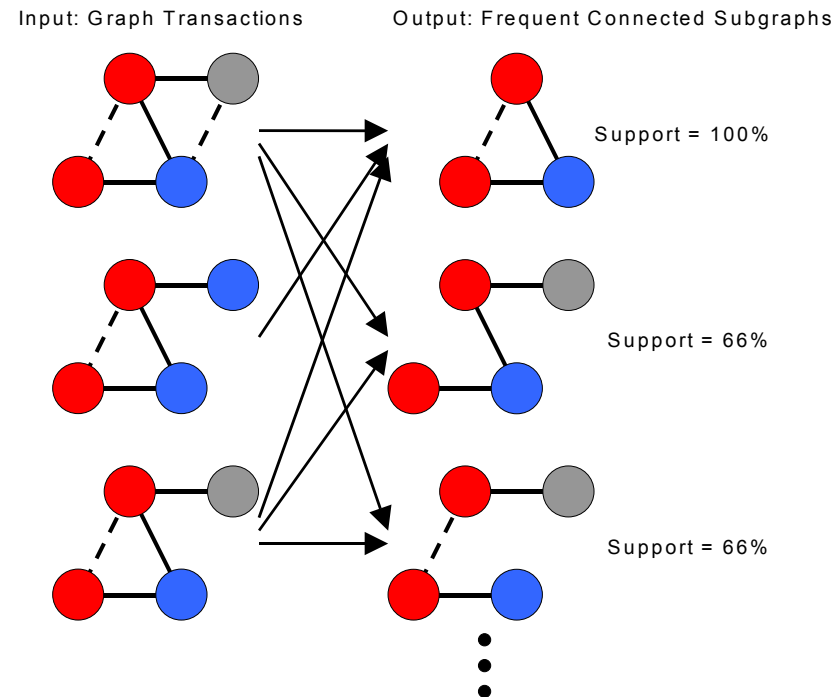
Finding Frequent Subgraphs: Input and Output

■ Input

- Database of graph transactions
- Undirected simple graph (no loops, no multiples edges)
- Each graph transaction has labels associated with its vertices and edges
- Transactions may not be connected
- Minimum support threshold σ

■ Output

- Frequent subgraphs that satisfy the minimum support constraint
- Each frequent subgraph is **connected**





The two Approaches

- At the core of any frequent subgraph mining algorithm are two computationally challenging problems
 - Subgraph isomorphism
 - Efficient enumeration of all frequent subgraphs
- Recent subgraph mining algorithms can be roughly classified into two categories
 - Use a level-wise search like Apriori to enumerate the recurring subgraphs, e.g. AGM, FSG
 - Use a depth-first search for finding candidate frequent subgraphs, e.g. gSpan, FFSM, MoFa, Gaston



Different Approaches for GM

- Apriori Approach
 - AGM
 - FSG 
 - Path Based
- DFS Approach
 - gSpan
 - FFSM
- Diagonal Approach
 - DSPM
- Greedy Approach
 - Subdue

Properties of Graph Mining Algorithms

- Search order
 - breadth vs. depth
- Generation of candidate subgraphs
 - apriori vs. pattern growth
- Elimination of duplicate subgraphs
 - passive vs. active
- Support calculation
 - embedding store or not
- Growing patterns by
 - Node → edge → path → tree → graph

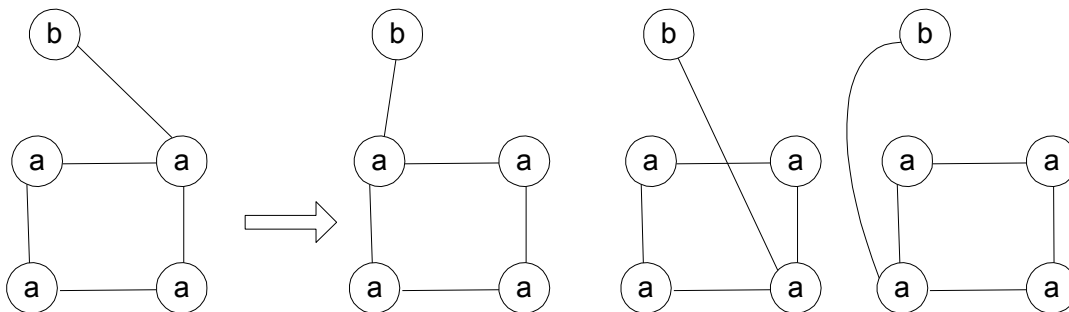


Problem Definition

- A labeled graph G is a 4-tuple (V, E, L, l)
 - V = set of vertices
 - E = set of edges, within $V \times V$
 - L = set of labels
 - l = label function, $V \cup E \rightarrow L$
- Undirected Graph G
 - Each edge is an unordered pair of vertices

Problem Complexity

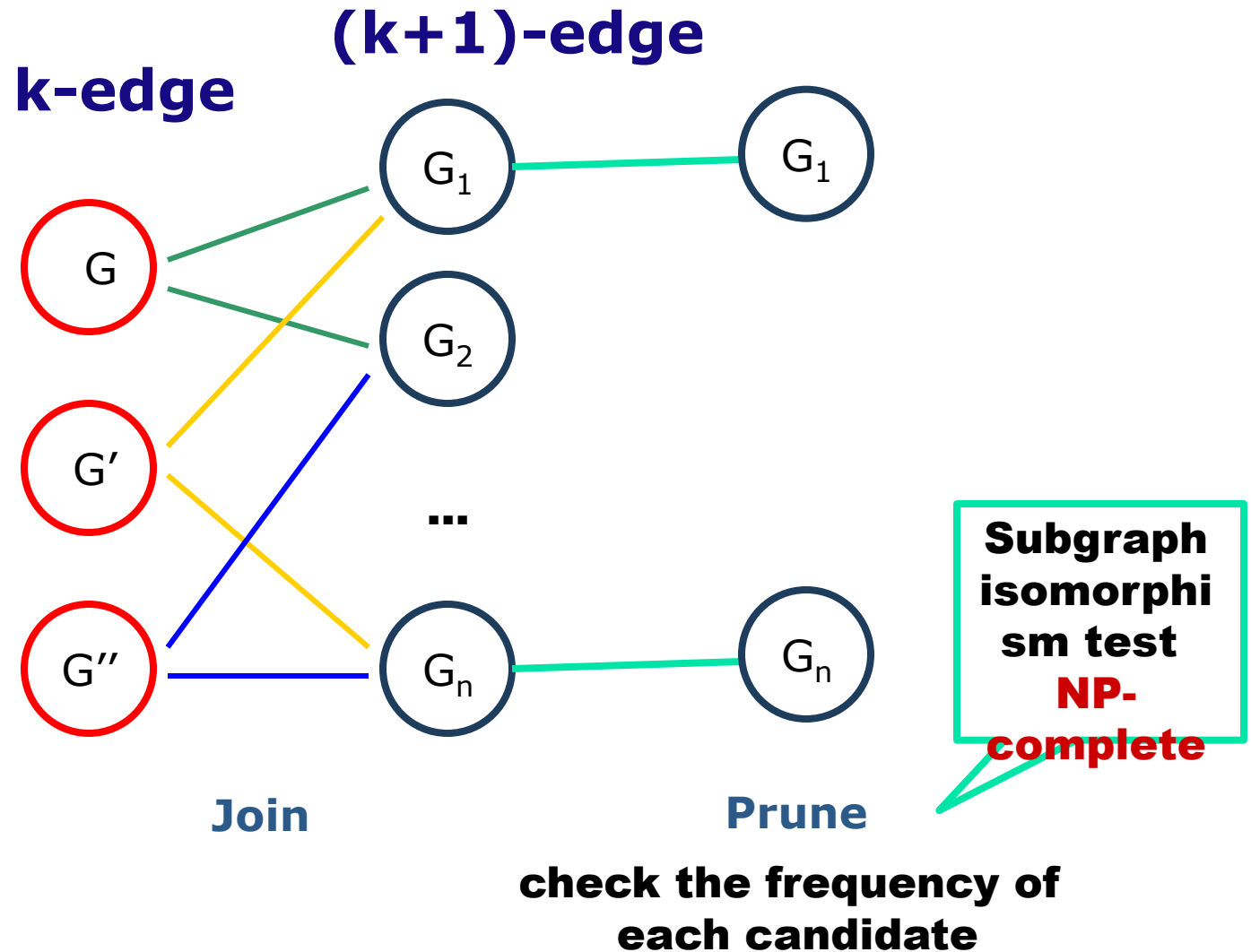
- **Isomorphism:** An isomorphism from G' to G is a function $f : V' \rightarrow V$, such that:
 1. For any vertex $u \in V'$
 - $f(u) \in V$ and $l'(u) = l(f(u))$
 2. For any edge $(u,v) \in E'$
 - $(f(u), f(v)) \in E$ and $l'(u,v) = l(f(u), f(v))$
- **Subgraph Isomorphism:** sub-graph isomorphism from G' to G is an isomorphism from G' to a sub-graph of G
- **Automorphism:** an automorphism of G is an isomorphism from G to *itself*
- *Examples for automorphism:*



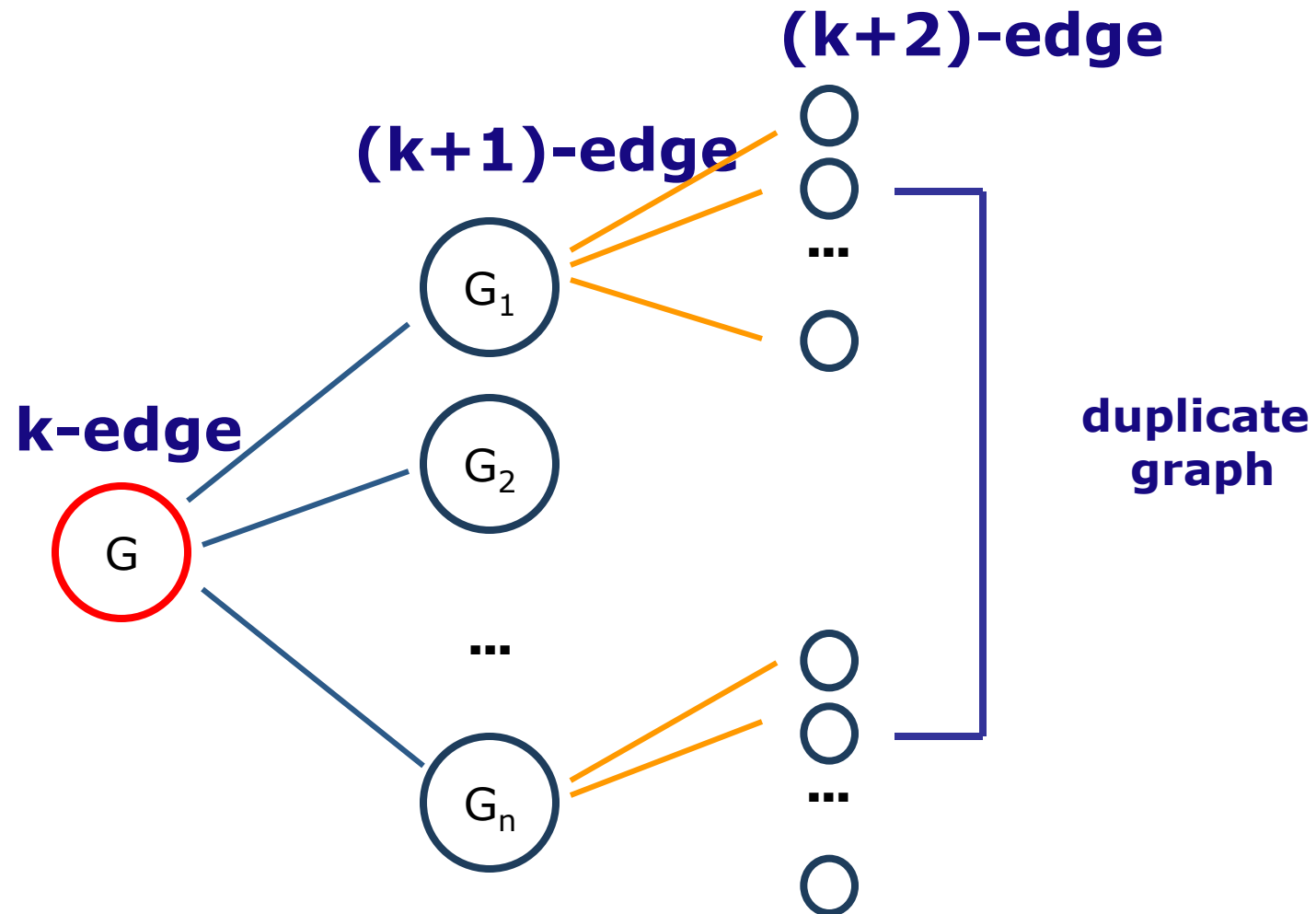
Problem Definition

- If each graph's vertices and edges have a unique label, then each graph can be modeled as a set of edges, and then use existing frequent itemset discovery algorithms to find all frequently occurring sub-graphs
- Since mapping of vertices and edges to labels is non-unique, frequent itemset solutions cannot be used - in this type of problem any frequent sub-graph discovery algorithm needs to solve many instances of sub-graph isomorphism problem, which is NP-complete
- Efficient frequent sub-graph mining algorithm tries to reduce the number of sub-graph isomorphism tests by reducing the search space

Apriori-Based Approach



Pattern Growth Method





Agenda

- Introduction
- Problem Definition
- FSG
- gSpan
- Scalable mining of large Disk-based Graph Databases



FSG Algorithm – Apriori based

Original version:

Kuramochi and G. Karypis. *Frequent subgraph discovery*.
[ICDM 2001]

Paper version: (with many optimizations)

M. Kuramochi, G. Karypis, "An Efficient Algorithm for
Discovering Frequent Subgraphs" IEEE TKDE,
September 2004 (vol. 16 no. 9)

FSG Algorithm

Init: Scan the transactions to find F_1 and F_2 the sets of all frequent 1-subgraphs and 2-subgraphs, together with their counts;

For ($k=3$; $F_{k-1} \neq \emptyset$; $k++$)

- 1) **Candidate Generation** - C_k , the set of candidate k -subgraphs, from F_{k-1} , the set of frequent $(k-1)$ -subgraphs found in the previous step;
- 2) **Candidates pruning** - a necessary condition of candidate to be frequent is that each of its $(k-1)$ -subgraphs is frequent.
- 3) **Frequency counting** - Scan the transactions to count the occurrences of subgraphs in C_k ;
- 4) $F_k = \{ c \in C_k \mid c \text{ has counts no less than } \#minSup \}$
Return $F_1 \cup F_2 \cup \dots \cup F_k$ ($= F$)

Frequent SubGraph Discovery

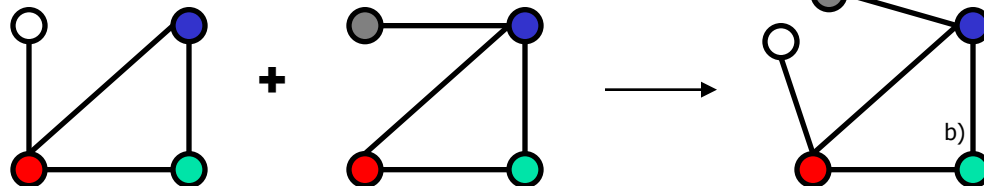
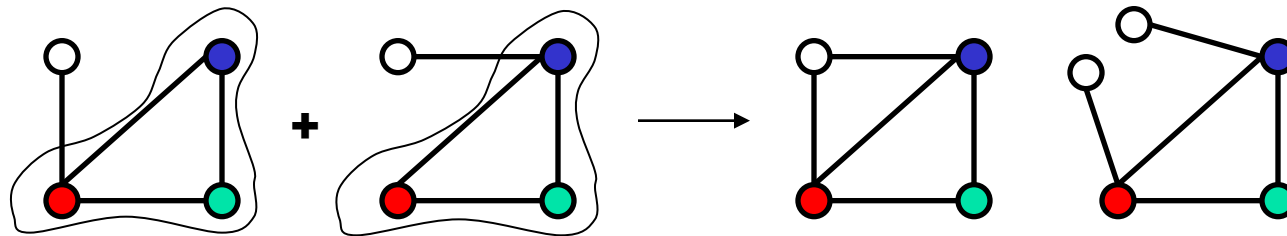
- Follows the level-by-level structure of the Apriori algorithm used for finding frequent itemsets
- FSG increase the size of frequent subgraphs by adding an **edge** one-by-one
 - Initially, enumerates all the frequent single and double edge graphs
 - During each iteration it first generates candidate subgraphs whose size is greater than the previous frequent ones by one edge
 - Candidates which do not satisfy the downward closure property are pruned
 - Next, it counts the frequency for each of these candidates, and prunes subgraphs that do not satisfy the support constraint



Trivial Operations Become Complicated with Graphs

- **Candidate generation**
 - To determine two candidates for joining, we need to perform **sub-graph isomorphism** (checking if the two graphs have the same “core”)
- **Candidate pruning**
 - To check downward closure property, we need **graph isomorphism**
- **Frequency counting**
 - **Sub-graph isomorphism** for checking containment of a frequent sub-graph within a graph

Candidates Generation Based on Core Detection

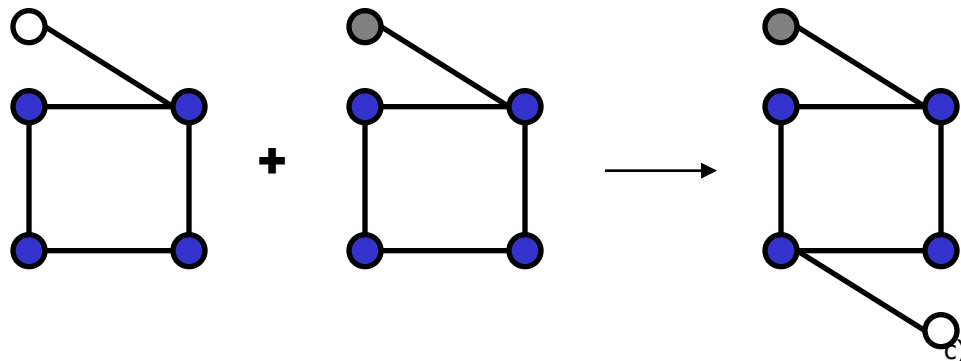


a)

the difference between the shared core and the two subgraphs can be a vertex that has the same or different label in both k -subgraphs

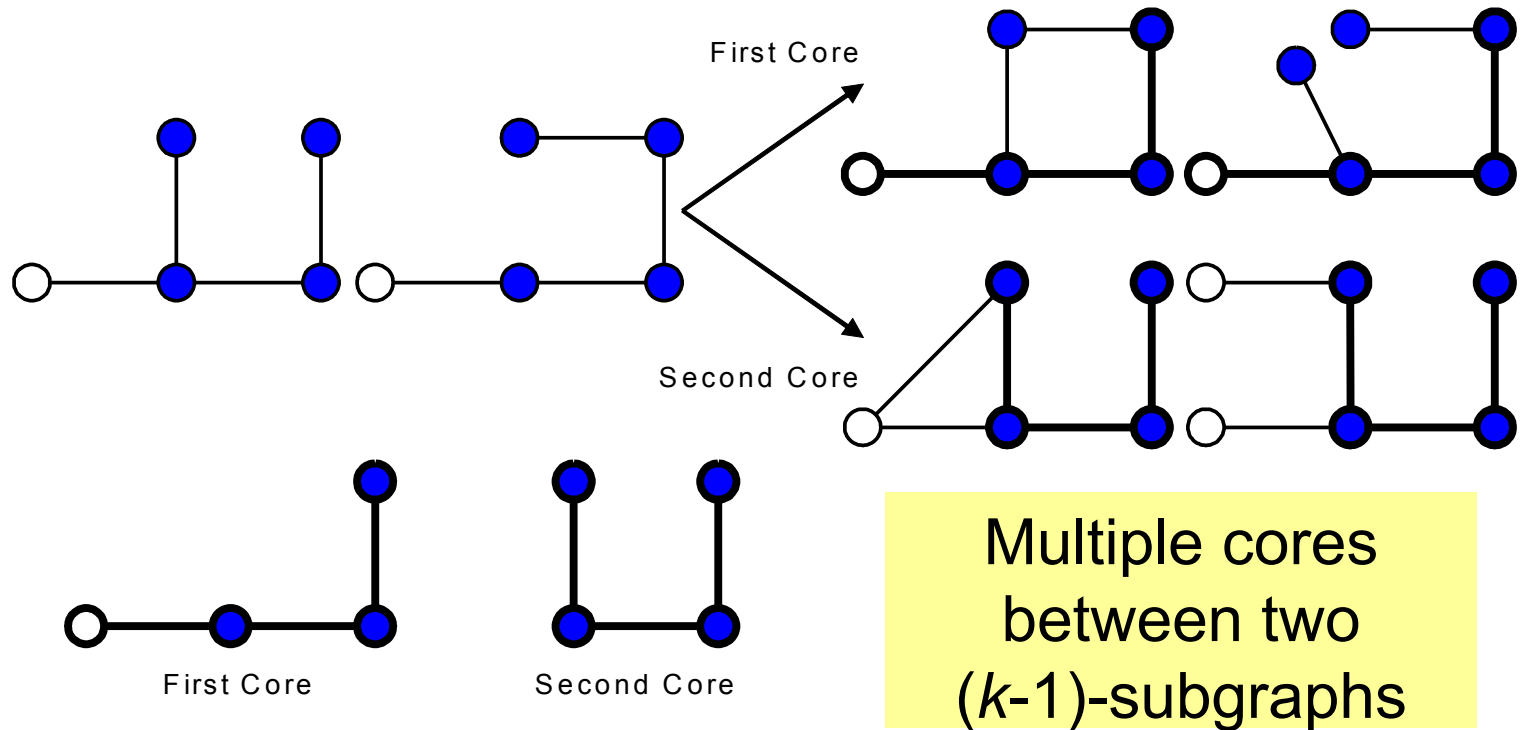
b)

the core itself may have multiple automorphisms. Each of them can lead to a different $(k+1)$ -candidate



two frequent subgraphs may have multiple cores

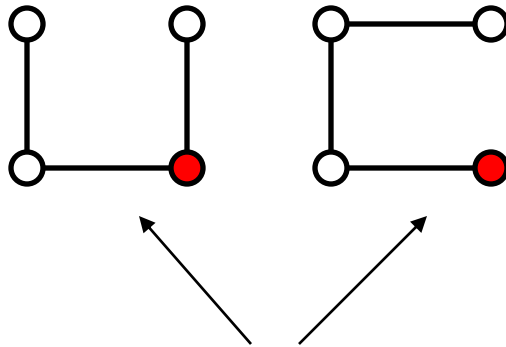
Candidate Generation Based On Core Detection (cont.)



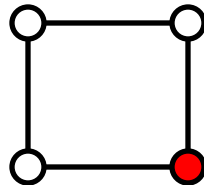
Candidate pruning:

Downward closure property

3-candidates:



4-candidates:



- Every $(k-1)$ -subgraph must be frequent
- For all the $(k-1)$ -subgraphs of a given k -candidate, check if downward closure property holds





Computation challenges

■ Candidate generation

- To determine if we can join two candidates, we need to perform subgraph isomorphism to determine if they have a common subgraph
- There is no obvious way to reduce the number of times that we generate the same sub-graph
- Need to perform graph isomorphism for redundancy checks (see canonical labeling...)
- The joining of two frequent sub-graphs can lead to multiple candidate sub-graphs

■ Candidate pruning

- To check downward closure property, we need sub-graph isomorphism

■ Frequency counting

- Sub-graph isomorphism for checking containment of a frequent sub-graph



FSG Optimizations

- Key to FSG's computational efficiency
 - Uses an efficient algorithm to determine a *canonical labeling* of a graph and use these “*strings*” to perform identity checks (simple comparison of strings!)
 - Uses a sophisticated candidate generation algorithm that reduces the number of times each candidate is generated
 - Uses an augmented TID-list based approach to speedup frequency counting



FSG Algorithm - details

Algorithm 1 $\text{fsg}(D, \sigma)$ (Frequent Subgraph)

```
1:  $F^1 \leftarrow$  detect all frequent 1-subgraphs in  $D$ 
2:  $F^2 \leftarrow$  detect all frequent 2-subgraphs in  $D$ 
3:  $k \leftarrow 3$ 
4: while  $F^{k-1} \neq \emptyset$  do
5:    $C^k \leftarrow \text{fsg-gen}(F^{k-1})$ 
6:   for each candidate  $G^k \in C^k$  do
7:      $G^k.\text{count} \leftarrow 0$ 
8:     for each transaction  $T \in D$  do
9:       if candidate  $G^k$  is included in transaction  $T$  then
10:         $G^k.\text{count} \leftarrow G^k.\text{count} + 1$ 
11:    $F^k \leftarrow \{G^k \in C^k \mid G^k.\text{count} \geq \sigma|D|\}$ 
12:    $k \leftarrow k + 1$ 
13: return  $F^1, F^2, \dots, F^{k-2}$ 
```

FSG Algorithm - Candidate Generation

Algorithm 2 fsg-gen(F^k) (Candidate Generation)

```
1:  $C^{k+1} \leftarrow \emptyset$ 
2: for each pair of  $G_i^k, G_j^k \in F^k, i \leq j$  such that  $\text{cl}(G_i^k) \leq \text{cl}(G_j^k)$  do
3:    $H^{k-1} \leftarrow \{H^{k-1} \mid \text{a core } H^{k-1} \text{ shared by } G_i^k \text{ and } G_j^k\}$ 
4:   for each core  $H^{k-1} \in H^{k-1}$  do
5:      $\{B^{k+1}$  is a set of tentative candidates $\}$ 
6:      $B^{k+1} \leftarrow \text{fsg-join}(G_i^k, G_j^k, H^{k-1})$ 
7:     for each  $G_j^{k+1} \in B^{k+1}$  do
8:       {test if the downward closure property holds}
9:       flag  $\leftarrow$  true
10:      for each edge  $e_l \in G_j^{k+1}$  do
11:         $H_l^k \leftarrow G_j^{k+1} - e_l$ 
12:        if  $H_l^k$  is connected and  $H_l^k \notin F^k$  then
13:          flag  $\leftarrow$  false
14:          break
15:      if flag = true then
16:         $C^{k+1} \leftarrow C^{k+1} \cup \{G_j^{k+1}\}$ 
17: return  $C^{k+1}$ 
```

For each pair of frequent -
subgraph(canonical labeling -cl)

Detect shared core

Generates all possible
candidates of size $k+1$

Test downward closure
property

Add to candidate set

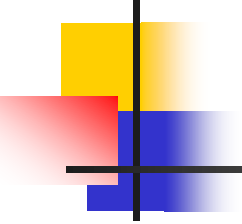


FSG - Candidate Generation(Cont.)

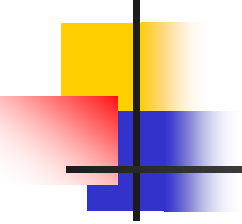
Algorithm 3 fsg-join(G_1^k, G_2^k, H^{k-1}) (Join)

- 1: $e_1 \leftarrow$ the edge appears only in G_1^k , not in H^{k-1}
 - 2: $e_2 \leftarrow$ the edge appears only in G_2^k , not in H^{k-1}
 - 3: $M \leftarrow$ generate all automorphisms of H^{k-1}
 - 4: $B^{k+1} = \emptyset$
 - 5: **for each** automorphism $\phi \in M$ **do**
 - 6: $B^{k+1} \leftarrow B^{k+1} \cup \{\text{all possible candidates of size } k+1 \text{ created from a set of } e_1, e_2, H^{k-1} \text{ and } \phi\}$
 - 7: **return** B^{k+1}
-

Candidate Generation - Core identification

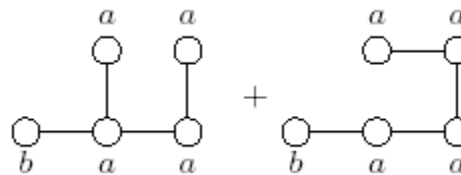
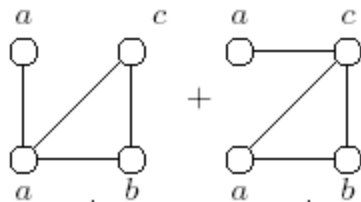
- 
- The key computational steps in candidate generation are:
 - Core identification
 - Joining
 - Using the downward closure property for pruning candidates
 - A straightforward way of performing these tasks:
 - A core between a pair of graphs G_i^k and G_j^k can be identified by creating each of the $(k-1)$ -subgraphs of G_i^k by removing each of the edges and checking whether this subgraph is also a subgraph of G_j^k
 - Join two size k -subgraph, to obtain size $(k+1)$ -candidates, by integrating two edges, one from each subgraph added to core
 - For a candidate of size $(k+1)$, generate each one of the k -size subgraphs by removing the edges and check if exists in F^k

Core identification (Cont.)

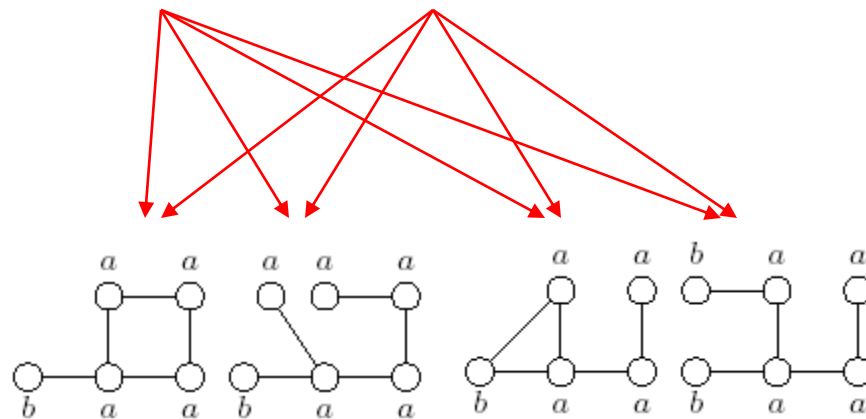
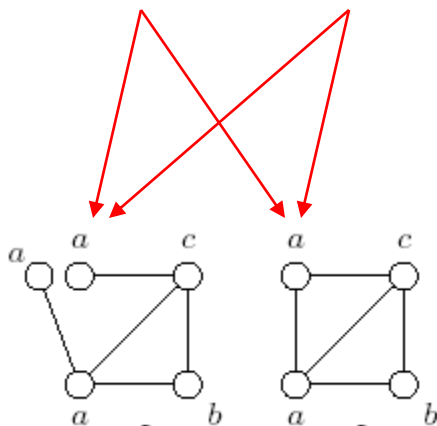
- 
- Using frequent subgraph lattice and canonical labeling to reduce complexity
 - Core identification:
 - Solution 1: for each frequent k -subgraph we store the canonical labels of its frequent $(k - 1)$ -subgraphs, then the cores between two frequent subgraphs can be determined by simply computing the intersection of these lists. *The complexity is quadratic on the number of frequent subgraphs of size k (i.e., $|F^k|$)*
 - Solution 2 - ***inverted indexing scheme*** - for each frequent subgraph of size $k - 1$, we maintain a list of child subgraphs of size k . Then, we only need to form every possible pair from the child list of every size $k - 1$ frequent subgraph.
This reduces the complexity of finding an appropriate pair of subgraphs to the square of the number of child subgraphs of size k

Candidate Generation

Frequent $k - 1$
subgraphs



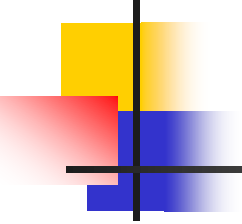
Frequent k
subgraphs



Solution 1: Each frequent k -subgraph stores the canonical labels of its frequent $(k - 1)$ -subgraphs

Solution 2: ***inverted indexing scheme*** - Each frequent subgraph of size $k - 1$ maintains a list of child subgraphs of size k

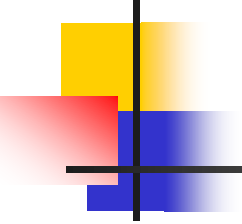
Candidate Generation -Optimization

- 
- Given a frequent sub-graph of size k – F_i , it contains at most k ($k-1$) sub-graphs. Order these sub-graphs by their canonical labels.
 - Call the smallest and second smallest sub-graphs H_{i1} and H_{i2} , define $P(F_i) = \{H_{i1}, H_{i2}\}$
 - An interesting property:
 - ***F_i and F_j can be joined only if the intersection of $P(F_i)$ and $P(F_j)$ is not empty!***

This dramatically reduces the number of possible joins!

Proof in Appendix of 2004 paper

Frequency Counting

- 
- For each frequent subgraph we keep a list of transaction identifiers that support it
 - When computing the frequency of G^{k+1} , we first compute the intersection of the TID lists of its frequent k -subgraphs.
 - If the size of the intersection is below the support, G^{k+1} is pruned
 - Otherwise we compute the frequency of G^{k+1} using subgraph isomorphism by limiting our search only to the set of transactions in the intersection of the TID lists

Another FSG Heuristic: Frequency Counting

Transactions

$$g^{k-1}_1, g^{k-1}_2 \subset T1$$

$$g^{k-1}_1 \subset T2$$

$$g^{k-1}_1, g^{k-1}_2 \subset T3$$

$$g^{k-1}_2 \subset T6$$

$$g^{k-1}_1 \subset T8$$

$$g^{k-1}_1, g^{k-1}_2 \subset T9$$

Frequent subgraphs

$$TID(g^{k-1}_1) = \{ 1, 2, 3, 8, 9 \}$$

$$TID(g^{k-1}_2) = \{ 1, 3, 6, 9 \}$$

Candidate

$$c^k = \text{join}(g^{k-1}_1, g^{k-1}_2)$$

$$TID(c^k) \subset TID(g^{k-1}_1) \cap TID(g^{k-1}_2)$$



$$TID(c^k) \subset \{ 1, 3, 9 \}$$

- Perform subgraph-iso to T1, T3 and T9 with c^k and determine $TID(c^k)$
- Note, TID lists require a lot of memory (but paper has some memory optimizations)

Canonical Labeling

- FSG relies on canonical labeling to efficiently perform a number of operations such as:
 - Checking whether or not a particular pattern satisfies the downward closure property of the support condition
 - Finding whether a particular candidate subgraph has already been generated or not
- Efficient canonical labeling is critical to ensure that FSG can scale to very large graph datasets
- Canonical label of a graph is a *code* that uniquely identifies the graph such that if two graphs are isomorphic to each other, they will be assigned the same code
- A simple way of assigning a code to a graph is to convert its adjacency matrix representation into a linear sequence of symbols. For example, by concatenating the rows or the columns of the graph's adjacency matrix one after another to obtain a sequence of zeros and ones or a sequence of vertex and edge labels

Canonical Labeling - Basics

- The code derived from adjacency matrix cannot be used as the graph canonical label since it depends on the order of the vertices
- One way to obtain isomorphism-invariant codes is to try every possible permutation of the vertices and its corresponding adjacency matrix, and to choose the ordering which gives lexicographically the largest, or the smallest code

		v_3	v_1	v_2	v_4	v_5	v_0
		0	0	0	0	0	0
v_3	0		1	1	1	1	
v_1	0	1		1			1
v_2	0	1	1				
v_4	0	1					
v_5	0	1					
v_0	0		1				

(a)

		v_1	v_0	v_2
		a	a	a
v_1	a		z	y
v_0	a	z		x
v_2	a	y	x	

(b)

Code: 000000111100100001000

Code: aaazyx

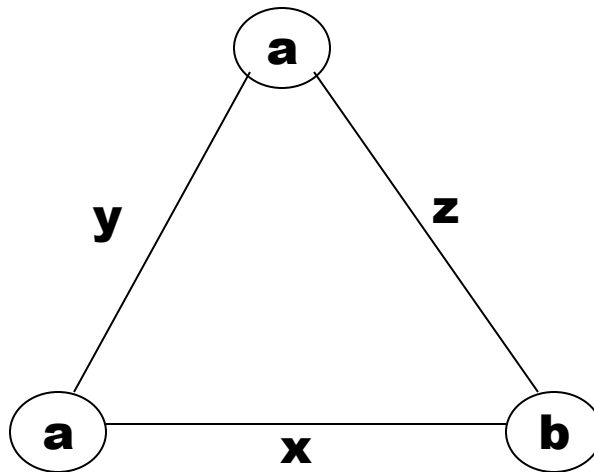
Time complexity: $O(|V|!)$ ■

FSG: Canonical Representation for graphs (based on adjacency Matrix)

Code(M_1) = “aabyzx”

Code(M_2) = “abaxyz”

Graph G:



M_1 :

	a	a	b
a		y	z
a	y		x
b	z	x	

M_2 :

	a	b	a
a		x	y
b	x		z
a	y	z	

Code(G) = min{ code(M) | M is adj. Matrix }

FSG: Finding the Canonical Labeling



- The problem is as complex as Graph Isomorphism (exponential?), (because we need to check all permutations) but
- FSG suggests some heuristics to speed it up, such as
 - Vertex invariants (e.g., degree)
 - Neighbor lists
 - Iterative partitioning
- Basically the heuristics allow to eliminate equivalent permutations



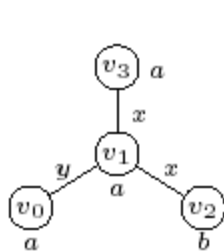
Canonical Labeling – Vertex Invariants

- Vertex invariants are properties assigned to a vertex which do not change across isomorphism mappings
- Vertex invariants is used to reduce the amount of time required to compute a canonical labeling, as follows:
 - Given a graph, the vertex invariants can be used to partition the vertices of the graph into equivalence classes such that all the vertices assigned to the same partition have the same values for the vertex invariants
 - maximize over those permutations that keep the vertices in each partition together
- Let m be the number of partitions created, containing p_1, p_2, \dots, p_m vertices, then the number of different permutations to consider is $\prod_{i=1}^m (p_i!)$ (instead of $(p_1 + p_2 + \dots + p_m)! \quad$)

Canonical Labeling – Vertex Invariants

Vertex Degrees and Labels:

- Vertices are partitioned into disjoint groups such that each partition contains vertices with the same label and the same degree
- Partitions are sorted by the vertex degree and label in each partition (e.g. V_0 and V_3)



(a)

	v_0	v_1	v_2	v_3
	a	a	b	a
v_0	a		y	
v_1	a	y		x
v_2	b		x	
v_3	a			x

(b)

	v_1	v_0	v_3	v_2
	a	a	a	b
v_1	a		x	y
v_0	a	x		
v_3	a	y		
v_2	b	x		
	p_0	p_1	p_2	

(c)

	v_1	v_3	v_0	v_2
	a	a	a	b
v_1	a		y	x
v_3	a	y		
v_0	a	x		
v_2	b	x		
	p_0	p_1	p_2	

(d)

- We can consider (x,y) and (y,x) for V_0 only...
- Only $1! \cdot 2! \cdot 1! = 2$ permutations, instead of $4! = 24$

Canonical Labeling – Vertex Invariants

Neighbor Lists:

- Incorporates information about the labels of the edges incident on each vertex, the degrees of the adjacent vertices, and their labels
- Adjacent vertex v is described by a tuple $(l(e), d(v), l(v))$:
 - $l(e)$ is the label of the incident edge e
 - $d(v)$ is the degree of the adjacent vertex v
 - $l(v)$ is its vertex label
- For each vertex u , construct its neighbor list $nl(u)$ that contains the tuples for each one of its adjacent vertices
- Partition the vertices into disjoint sets such that two vertices u and v will be in the same partition if and only if $nl(u) = nl(v)$

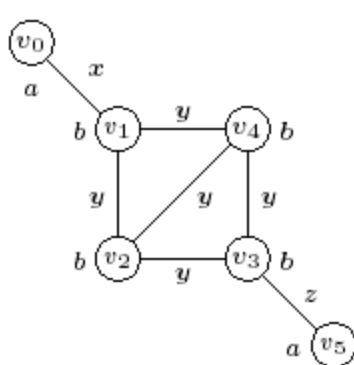
Canonical Labeling – Vertex Invariants

Neighbor Lists – continue:

- This partitioning is performed within the partitions already computed by the previous set of invariants (e.g. V_2 and V_4 have the same NL)

Vertex degrees and
labels partitioning

Neighbor lists
partitioning incorporated



(a)

	v_2	v_4	v_1	v_3	v_0	v_5
v_2	b		y	y	y	
v_4	b	y		y	y	
v_1	b	y	y			x
v_3	b	y	y			z
v_0	a			x		
v_5	a			z		
	p_0			p_1		

(b)

	v_2	v_4	v_1	v_3	v_0	v_5
v_2	b		y	y	y	
v_4	b	y		y	y	
v_1	b	y	y			x
v_3	b	y	y			z
v_0	a			x		
v_5	a			z		
	p_0		p_1	p_2	p_3	p_4

(c)

$(y, 3, b), (y, 3, b), (y, 3, b)$
 $(y, 3, b), (y, 3, b), (y, 3, b)$
 $(x, 1, a), (y, 3, b), (y, 3, b)$
 $(y, 3, b), (y, 3, b), (z, 1, a)$
 $(x, 3, b)$
 $(z, 3, b)$

(d)

Neighbor list

Search space reduced from $4! \cdot 2!$ to $2!$

Canonical Labeling – Vertex Invariants



Iterative Partitioning:

- Generalization of the idea of the neighbor lists, by incorporating the partition information
- See Paper



Canonical Labeling

Degree-based Partition Ordering

- Overall runtime of the canonical labeling can be further reduced by properly ordering the various partitions
- Partitions ordering may allow us to quickly determine whether a set of permutations can potentially lead to a code that is smaller than the current best code; thus, allowing us to prune large parts of the search space:
 - When we permute the rows and the columns of a particular partition, the code corresponding to the columns of the preceding partitions is not affected
 - If the code is smaller than the prefix of the currently best code, then the exploration of this set of permutations can be terminated
- Partitions are sorted in decreasing order of the degree of their vertices

Canonical Labeling - Degree-based Partition Ordering

Example

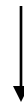
All vertices are labeled: a



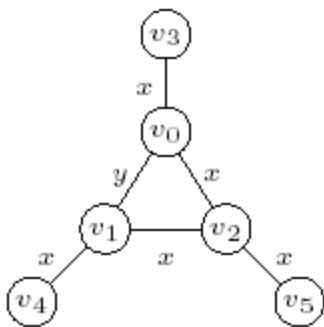
Partitions sorted by vertex degree in **ascending** order



Partitions sorted by vertex degree in **descending** order



Some permutation of p1 of (c), resulting with smaller prefix than (c) – saves us the permutations of p0



(a)

		v_3	v_4	v_5	v_0	v_1	v_2
		a	a	a	a	a	a
v_3	a				x		
v_4	a					x	
v_5	a						x
v_0	a	x				y	x
v_1	a		x			y	x
v_2	a			x	x	x	
		p_0			p_1		

(b)

		v_0	v_1	v_2	v_3	v_4	v_5
		a	a	a	a	a	a
v_0	a		y	x	x		
v_1	a		y		x		
v_2	a	x	x				x
v_3	a	x					
v_4	a		x				
v_5	a			x			
		p_1			p_0		

(c)

		v_2	v_0	v_1	v_3	v_4	v_5
		a	a	a	a	a	a
v_2	a		x	x			x
v_0	a	x		y	x		
v_1	a	x	y				x
v_3	a		x				
v_4	a						
v_5	a	x		x			
		p_1			p_0		

(d)

Experimental results

Comparison of various optimizations using the chemical compound dataset

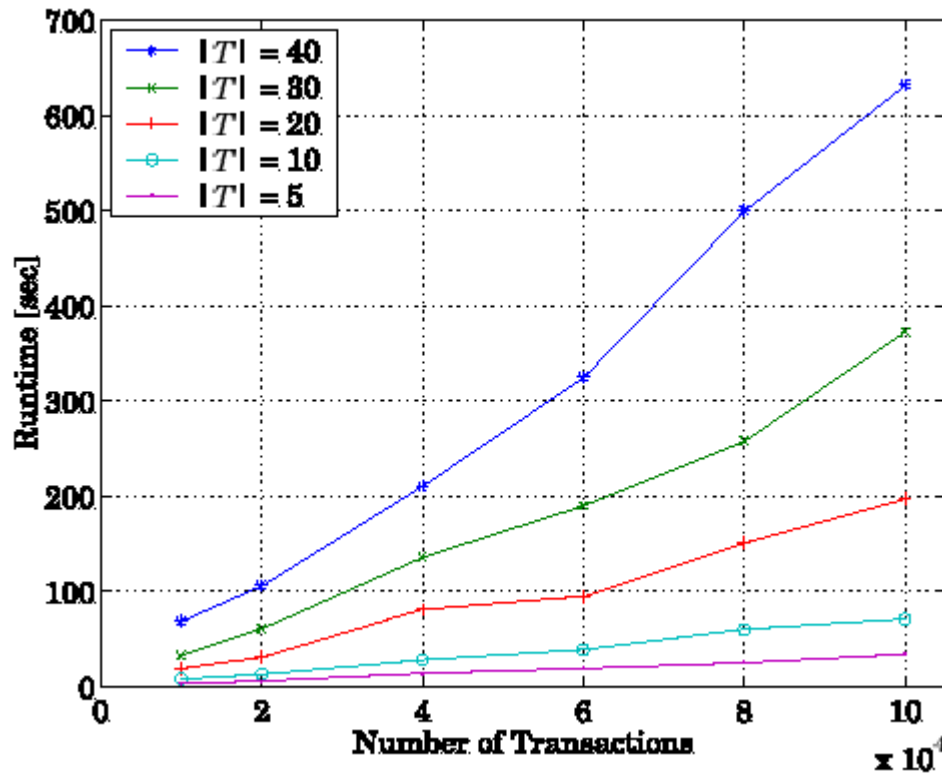
Note: Run-time with this and previous optimizations (left to right)

Support σ [%]	Running Time[s] with Optimizations					Largest Pattern Size k^*	Candidates \mathbf{C}	Frequent Patterns \mathbf{F}
	Degree-Label Partitioning	Inverted Index	Partition Ordering	Neighbor List	Iterative Partitioning			
10.0	6	4	3	3	3	11	970	844
9.0	8	6	4	3	4	11	1168	977
8.0	22	13	6	5	5	11	1602	1323
7.5	29	15	7	6	6	12	1869	1590
7.0	45	23	10	7	7	12	2065	1770
6.5	138	59	17	9	9	12	2229	1932
6.0	1853	675	56	13	11	13	2694	2326
5.5	5987	1691	112	18	14	13	3076	2692
5.0	24324	7377	879	33	22	14	4058	3608
4.5	—	55983	4196	40	35	15	5533	4984
4.0	—	—	12363	126	51	15	6546	5935
3.5	—	—	—	697	152	20	14838	13816
3.0	—	—	—	3097	317	22	24064	22758
2.5	—	—	—	9329	537	22	33660	31947
2.0	—	—	—	—	3492	25	139666	136927

Chemical compound dataset: 340 chemical compounds, 24 different element names, 66 different element types, 4 types of bonds

Experimental results

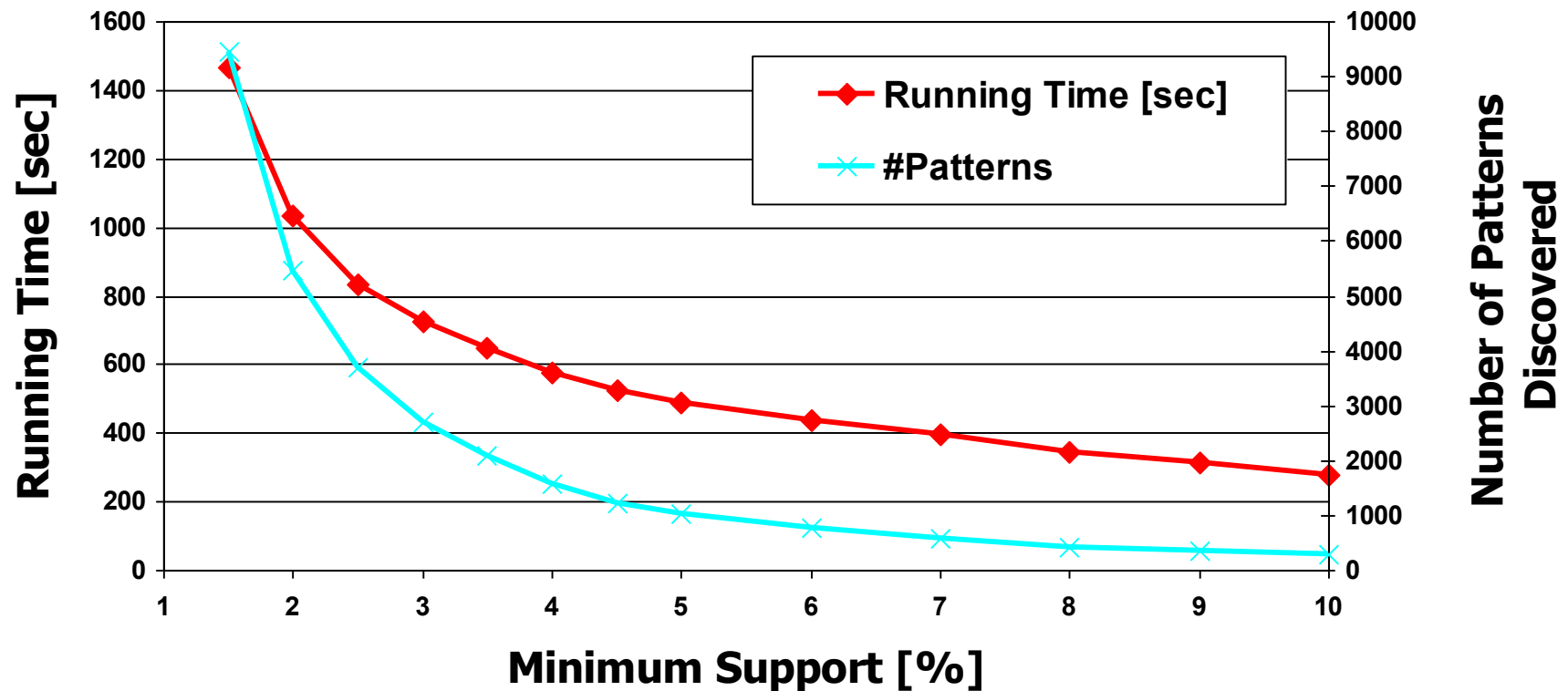
Database size scalability



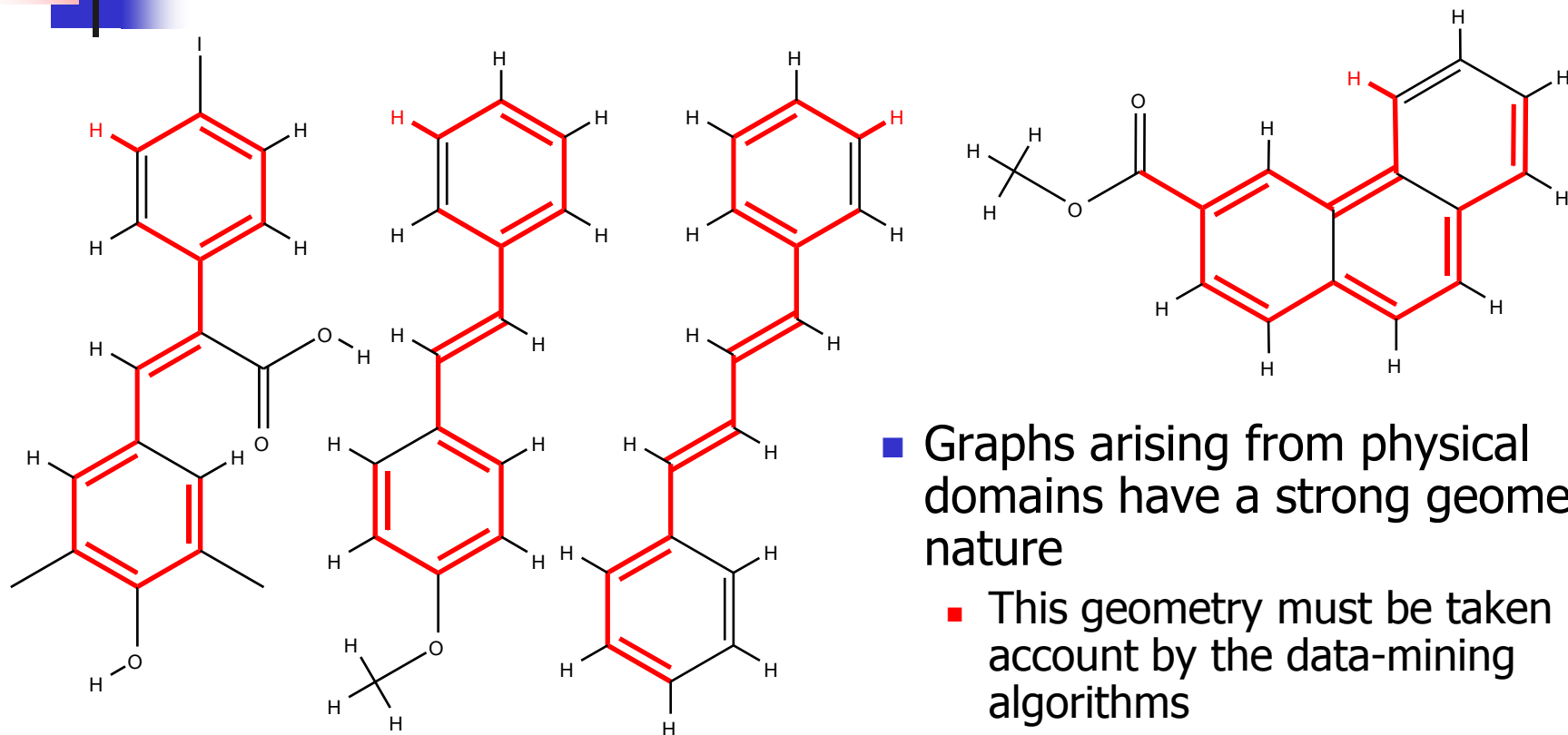
$|T|$ - average size of transactions (in terms of number of edges)

DTP Dataset (chemical compounds)

(Random 100K transactions)



FSG extension - Topology Is Not Enough (Sometimes)

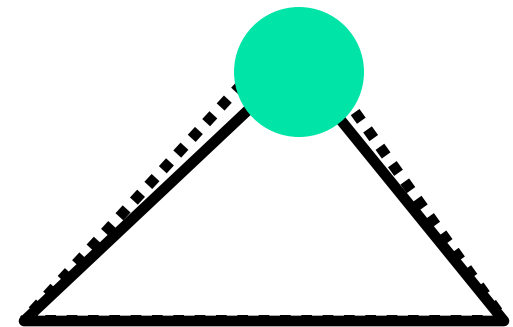
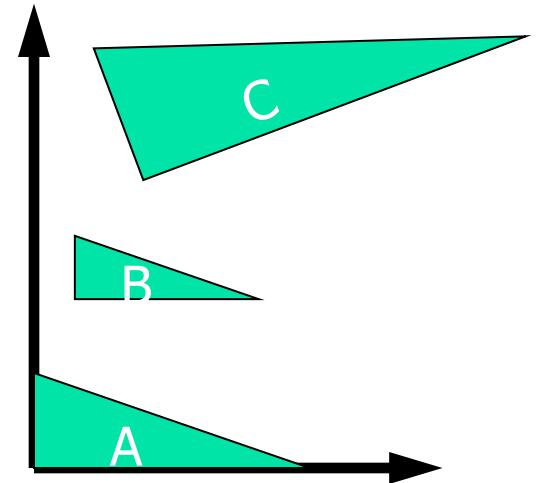


- Graphs arising from physical domains have a strong geometric nature
 - This geometry must be taken into account by the data-mining algorithms
- Geometric graphs
 - Vertices have physical 2D and 3D coordinates associated with them

gFSG—Geometric Extension Of FSG

(Kuramochi & Karypis ICDM 2002)

- Same input and same output as FSG
 - Finds frequent geometric connected subgraphs
- Geometric version of (sub)graph isomorphism
 - The mapping of vertices can be translation, rotation, and/or scaling invariant
 - The matching of coordinates can be inexact as long as they are within a tolerance radius of r
 - R -tolerant geometric isomorphism



Different Approaches for GM

- Apriori Approach
 - AGM
 - FSG
 - Path Based (later)
- DFS Approach
 - gSpan 
 - FFSM
- Diagonal Approach
 - DSPM
- Greedy Approach
 - Subdue

Y. Xifeng and H. Jiawei

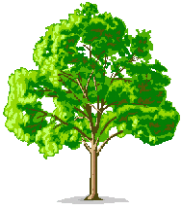
**gspan: Graph-Based
Substructure Pattern Mining**

ICDM, 2002



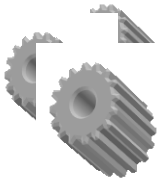
gSpan Outline

Part 1

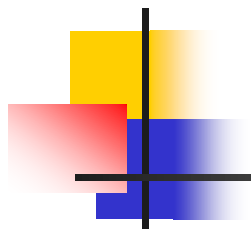


- Defines a canonical representation for graphs
- Defines Lexicographic order over the canonical representations
- Defines Tree Search Space (TSS) based on the lexicographic order

Part 2



- Discovers all frequent subgraphs by DFS exploration of TSS

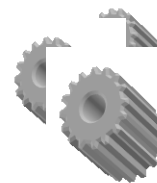


Part 1



Defining the Tree Search Space (TSS)

Part 2



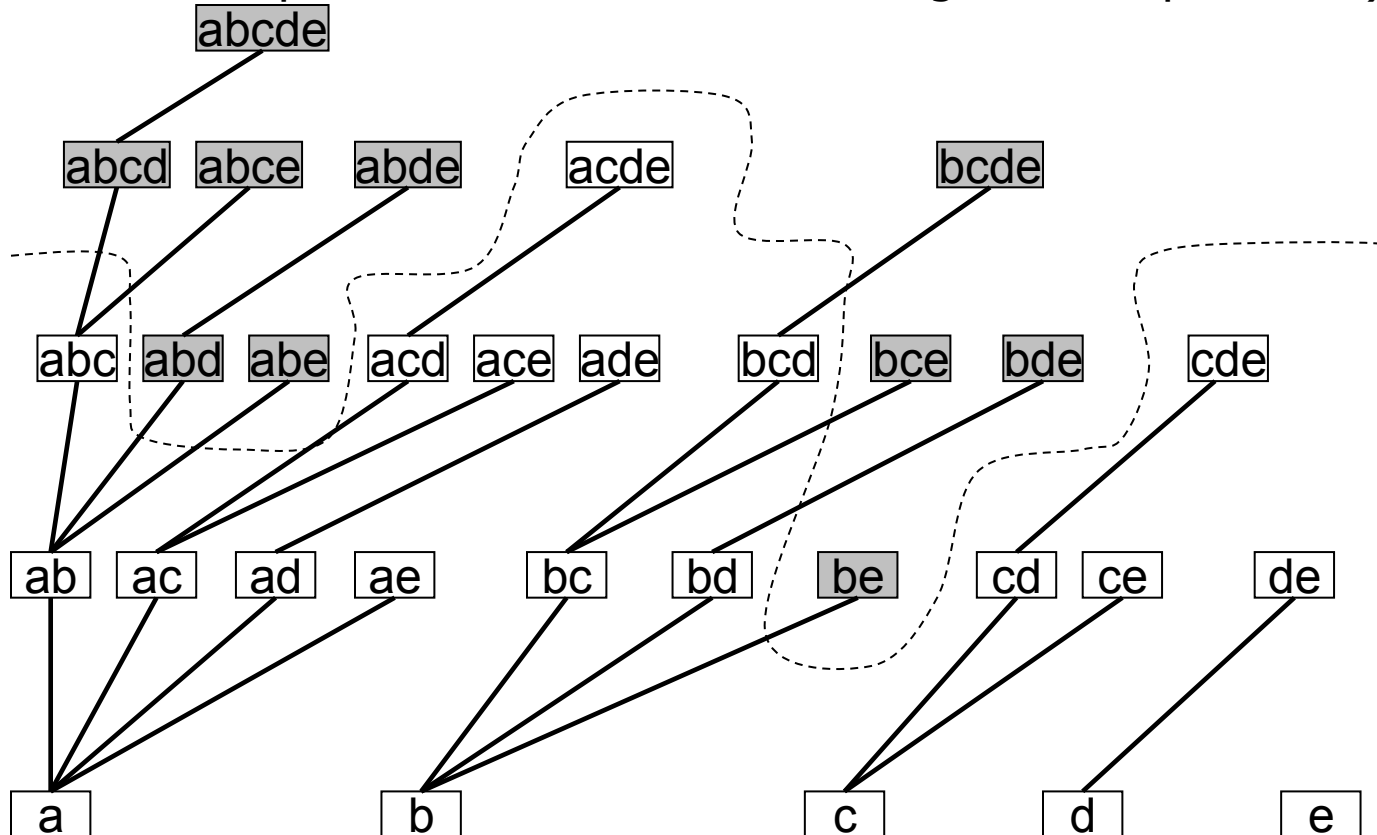
gSpan Finds all frequent graphs
by Exploring TSS

Motivation

DFS exploration vs. itemsets



Itemset Search space – **prefix based** (Note at the time we explore 'abe' we don't have enough info. to prune it...)



Motivation



Itemsets TSS properties

- **Canonical representation** of itemset is accepted 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



Targets



- Enumerating all frequent subgraphs by constructing a TSS, so
 - Completeness—There will be no duplications/omissions
 - A child (in tree) will be accepted from a parent, by extending the parent pattern
 - Correct pruning techniques

DFS Code representation

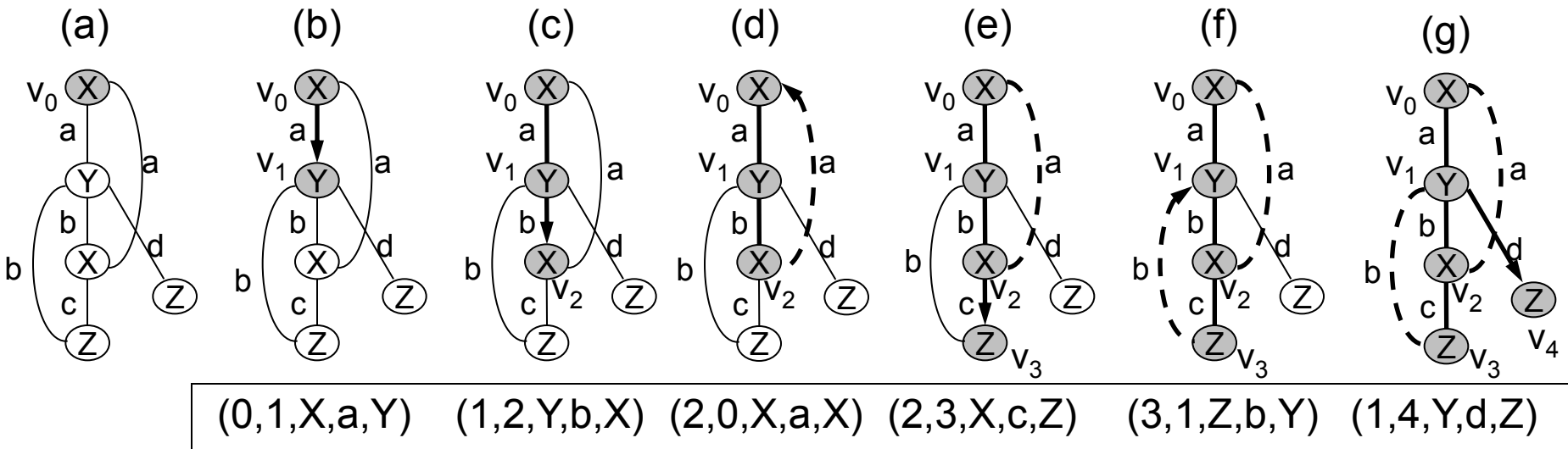


- Map each graph (2-Dim) to a sequential DFS Code (1-Dim)
- Lexicographically order the codes
- Construct TSS based on the **lexicographic order**

DFS-Code construction



- Given a graph G
- For each Depth First Search over graph G, construct a corresponding DFS-Code

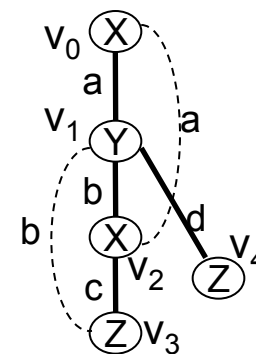
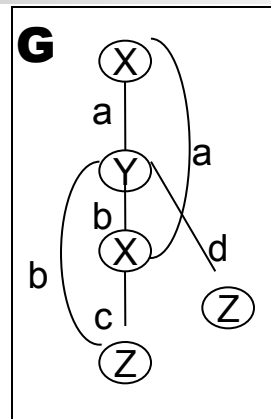


Dfs_Code(G, dfs) /*dfs - give some depth search over G*/

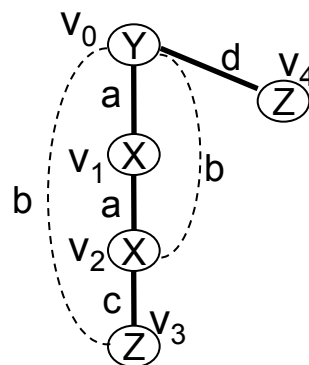


Single graph, Several DFS-Codes

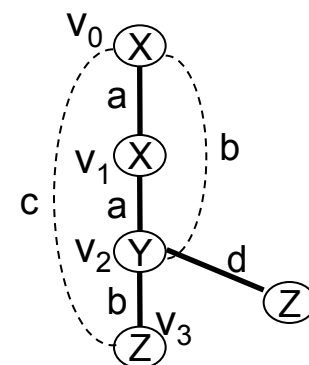
	(a)	(b)	(c)
1	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, X, a, X)
2	(1, 2, Y, b, X)	(1, 2, X, a, X)	(1, 2, X, a, Y)
3	(2, 0, X, a, X)	(2, 0, X, b, Y)	(2, 0, Y, b, X)
4	(2, 3, X, c, Z)	(2, 3, X, c, Z)	(2, 3, Y, b, Z)
5	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	(3, 0, Z, c, X)
6	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	(2, 4,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



(a)



(b)



(c)

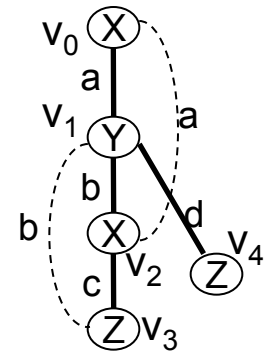
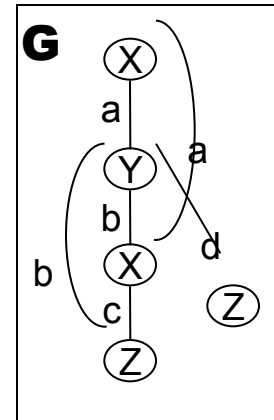
Single graph, Single Min DFS-Code!



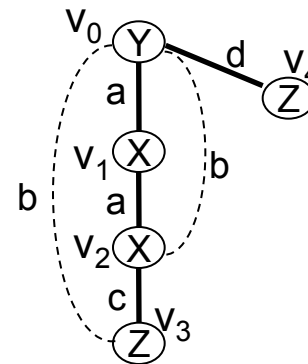
DFS code in column

**Min
DFS-Code**

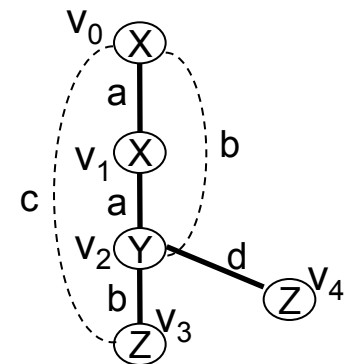
	(a)	(b)	(c)
1	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, X, a, X)
2	(1, 2, Y, b, X)	(1, 2, X, a, X)	(1, 2, X, a, Y)
3	(2, 0, X, a, X)	(2, 0, X, b, Y)	(2, 0, Y, b, X)
4	(2, 3, X, c, Z)	(2, 3, X, c, Z)	(2, 3, Y, b, Z)
5	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	(3, 0, Z, c, X)
6	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	(2, 4,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



(a)



(b)



(c)

DFS Lexicographic Order

- Let **Z** be the set of DFS codes of all graphs. Two DFS codes **a** and **b** have the relation **a ≤ b** (DFS Lexicographic Order in Z) if and only if one of the following conditions is true. Let

$$\mathbf{a} = (x_0, x_1, \dots, x_n) \text{ and}$$

$$\mathbf{b} = (y_0, y_1, \dots, y_n),$$

- (i) if there exists **t**, $0 \leq t \leq \min(m, n)$, $x_k = y_k$ for all **k**, s.t. $k < t$, and $x_t < y_t$
- (ii) $x_k = y_k$ for all **k**, s.t. $0 \leq k \leq m$ and $m \leq n$.



Minimum DFS-Code

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

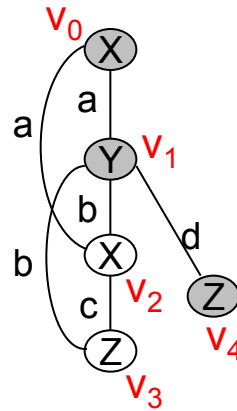
$$\min(A) = \min(B)$$

DFS-Code Tree: Parent-Child Relation



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

Graph G_1

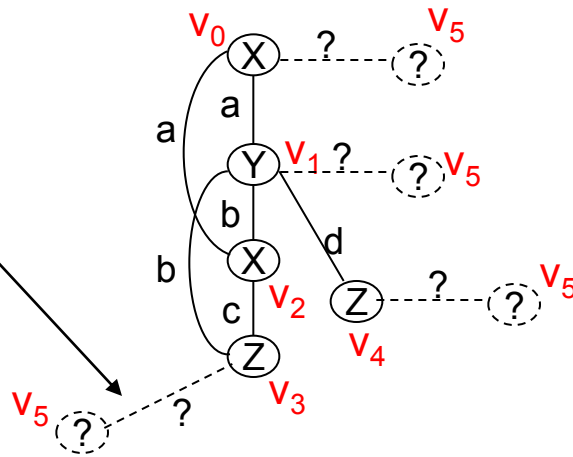


$\text{Min}(g) = (0, 1, X, a, Y) \quad (1, 2, Y, b, X) \quad (2, 0, X, a, X) \quad (2, 3, X, c, Z) \quad (3, 1, Z, b, Y) \quad (1, 4, Y, d, Z)$

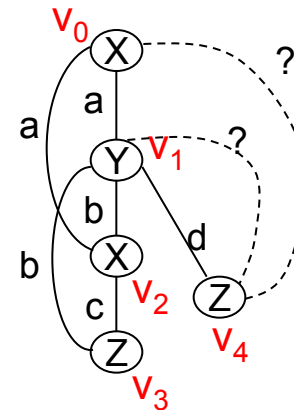
**A child of Graph g must grow edge from
right most path of G_1 (necessary condition)**

Graph G_2

wrong



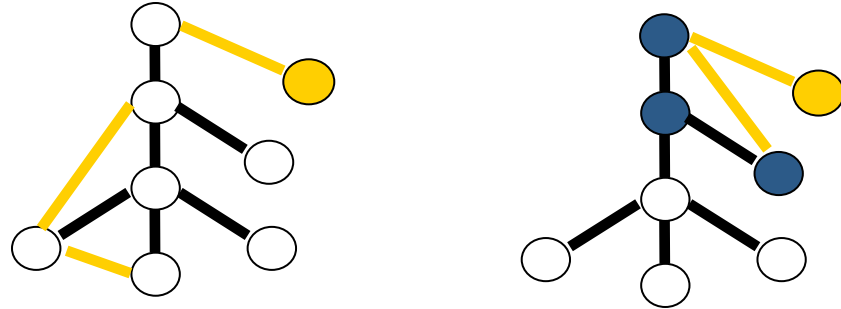
Forward EDGE



Backward EDGE

GSPAN (Yan and Han ICDM'02)

Right-Most Extension



Theorem: Completeness

**The Enumeration of Graphs
using Right-most Extension is
COMPLETE**

DFS Code Extension

- Let **a** be the minimum DFS code of a graph **G** and **b** be a non-minimum DFS code of **G**. For any DFS code **d** generated from **b** by one right-most extension,
 - (i) **d** is not a minimum DFS code,
 - (ii) $\text{min_dfs}(\mathbf{d})$ cannot be extended from **b**, and
 - (iii) $\text{min_dfs}(\mathbf{d})$ is either less than **a** or can be extended from **a**.

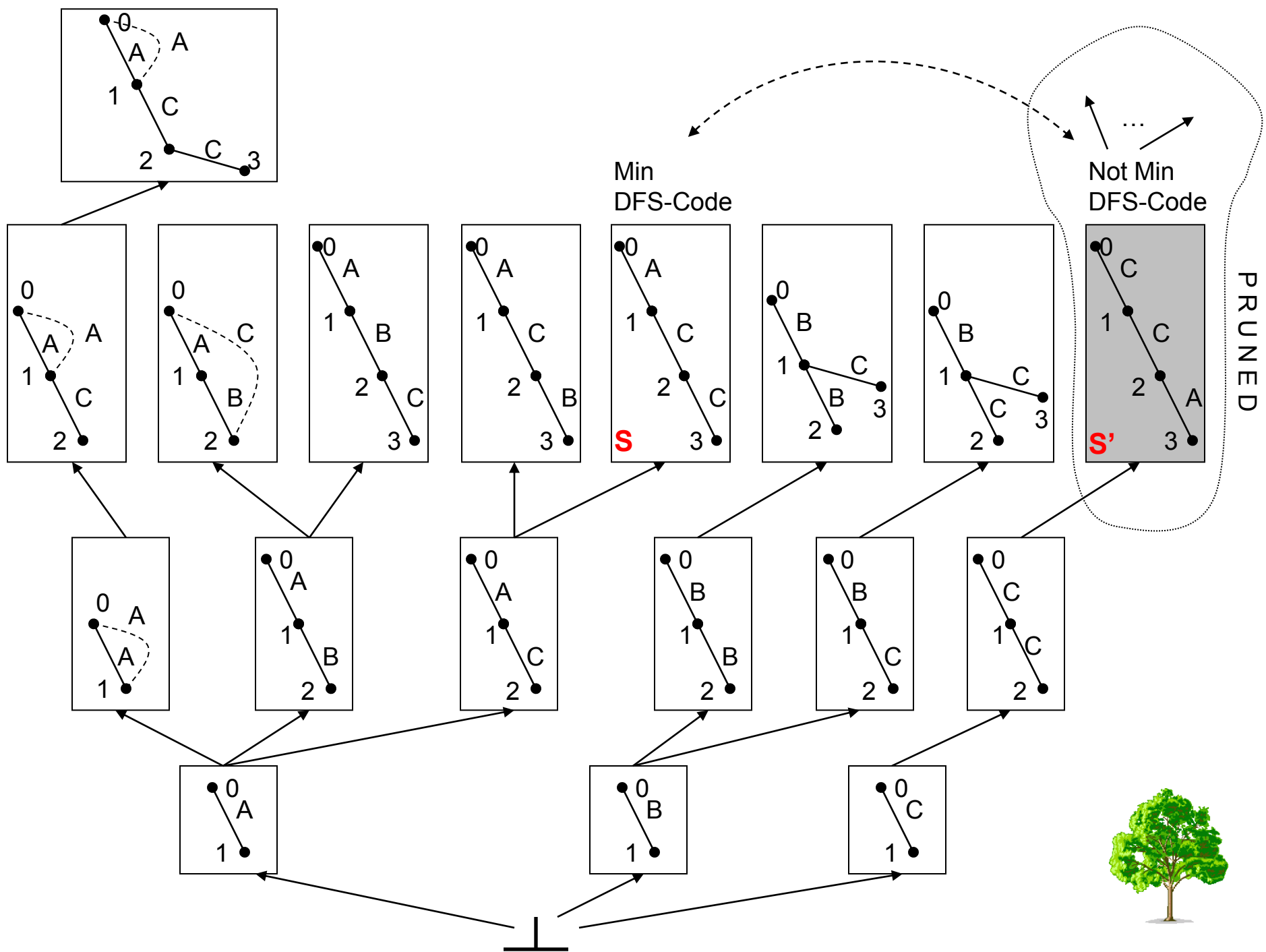
THEOREM [RIGHT-EXTENSION]

The DFS code of a graph extended from a Non-minimum DFS code is NOT MINIMUM

Search Space: DFS code Tree



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





Tree Pruning

- All of the descendants of infrequent node are infrequent also (just like with itemsets!)
- All of the descendants of a non min-DFS code are also non min-DFS code
- Therefore as soon as you discover a non min-DFS graph you can prune it!



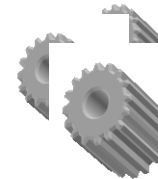
Part 1



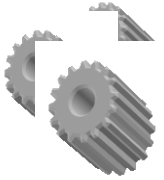
Defining the Tree Search Space (TSS)



Part 2



gSpan Finds all frequent graphs
by Exploring TSS




gSpan Algorithm

gSpan (D , F , g)

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

* gSpan improve this line

The gSpan Algorithm (details)

- 
- 1: sort labels of the vertices and edges in \mathbb{GS} by their frequency;
 - 2: remove infrequent vertices and edges;
 - 3: relabel the remaining vertices and edges in descending frequency;
 - 4: $S^1 \leftarrow$ all frequent 1-edge graphs in \mathbb{GS} ;
 - 5: sort S^1 in DFS lexicographic order;
 - 6: $S \leftarrow S^1$;
 - 7: **for each** edge $e \in S^1$ **do**
 - 8: initialize s with e , set $s.GS = \{g \mid \forall g \in \mathbb{GS}, e \in E(g)\}$; (only graph ID is recorded)
 - 9: Subgraph_Mining(\mathbb{GS}, S, s);
 - 10: $\mathbb{GS} \leftarrow \mathbb{GS} - e$; // Note with every iteration graph becomes smaller
 - 11: **if** $|\mathbb{GS}| < minSup$;
 - 12: **break**;

The gSpan Algorithm(Cont.)

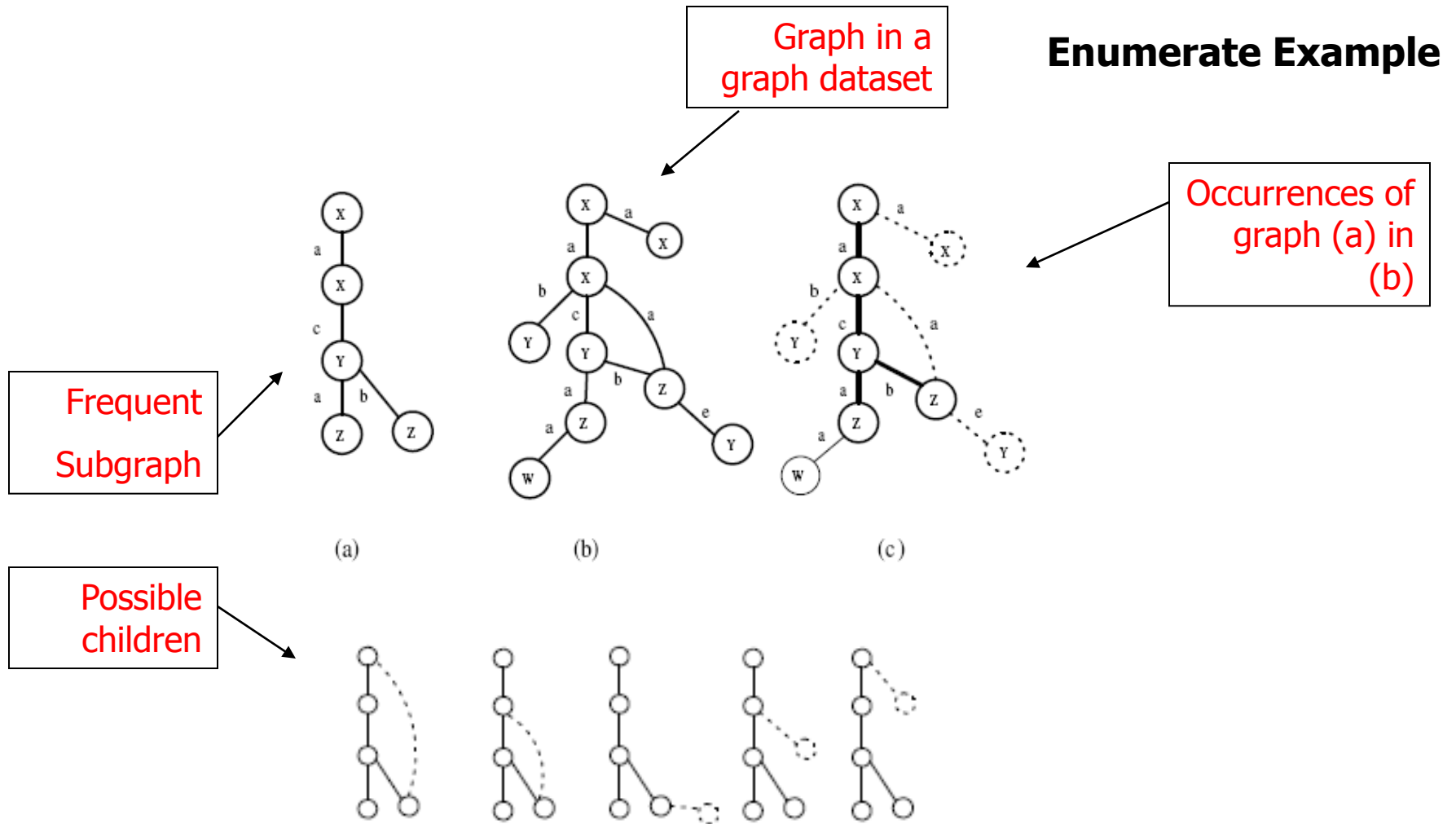
Subprocedure 1 Subgraph_Mining(\mathcal{GS} , \mathcal{S} , s).

```
1: if  $s \neq \min(s)$ 
2:   return;
3:  $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$ ;
4: generate all  $s$ ' potential children with one edge growth;†
5: Enumerate( $s$ );
6: for each  $c$ ,  $c$  is  $s$ ' child do
7:   if  $\text{support}(c) \geq \text{minSup}$ 
8:      $s \leftarrow c$ ;
9:     Subgraph_Mining( $\mathcal{GS}$ ,  $\mathcal{S}$ ,  $s$ );
```

Subprocedure 2 Enumerate(s).

```
1: for each  $g \in s.GS$  do
2:   enumerate the next occurrence of  $s$  in  $g$ ;
3:   for each  $c$ ,  $c$  is  $s$ ' child and occurs in  $g$  do
4:      $c.GS \leftarrow c.GS \cup \{g\}$ ;
5:   if  $g$  covers all children of  $s$  break;
```

The gSpan Algorithm - Enumerate children



The gSpan Algorithm - Pruning



The $s \neq \min(s)$ Pruning:

- $s \neq \min(s)$ prunes all DFS codes which are not minimum
- Significantly reduces unnecessary computation on duplicate subgraphs and their descendants
- Two ways for pruning
 - Pre-pruning: cutting off any child whose code is not minimum before counting frequency and after generating all potential children (after line 4 of [Subgraph Mining](#))
 - Post-pruning: pruning after the real counting
- First approach is costly since most of duplicate subgraphs are not even frequent, on the other hand counting duplicate frequent subgraphs is a waste
- Next: **Optimizations**

The gSpan Algorithm - Pruning

The $s \neq \min(s)$ Pruning (cont.):

A trade-off between pre-pruning and post-pruning: prune any discovered child in four stages:

If the first edge of s minimum DFS code is e_0 , then a potential child of s does not contain any edge smaller than e_0

example: minimum DFS code of (a) is

(0,1,x,a,x) e_0

(1,2,x,c,y)

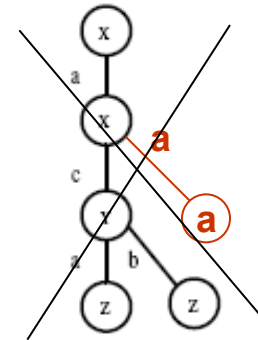
(2,3,y,a,z)

(2,4,y,b,z)

If a potential child of s could add the edge (x,a,a)

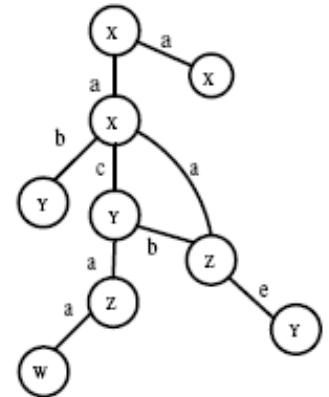
(x,a,a) < (x,a,x) \rightarrow s child pruned

Frequent subgraph



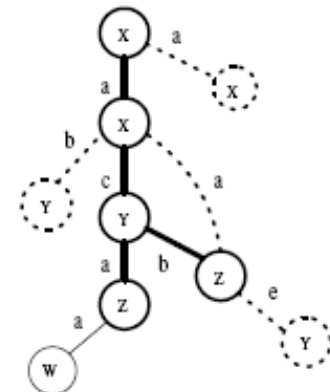
(a)

Database graph



(b)

potential children



(c)

The gSpan Algorithm - Pruning

The $s \neq \min(s)$ Pruning (cont.):

For any backward edge growth from s (v_i, v_j) $i > j$, this edge should be no smaller than any edge which is connected to v_j in s

example:

(a) min DFS

(0,1,x,a,x)

(1,2,x,c,y)

(2,3,y,a,z)

(2,4,y,b,z)

(a) growth

(0,1,x,a,x)

(1,2,x,c,y)

(2,3,y,a,z)

(2,4,y,b,z)

(4,1,z,a,x)

Growth min DFS

(0,1,x,a,x)

(1,2,x,a,z)

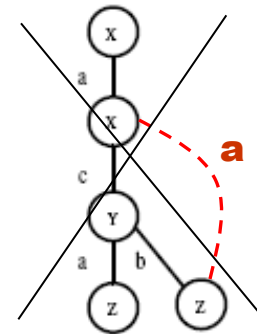
(2,3,z,b,y)

(3,1,y,c,z)

(3,4,y,a,z)

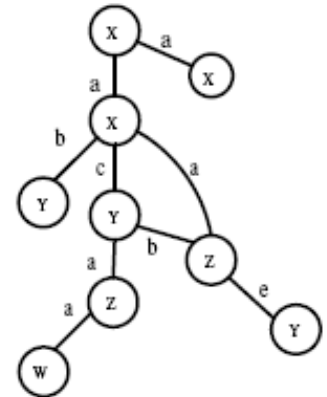
$S \neq \min(s)$

Frequent subgraph



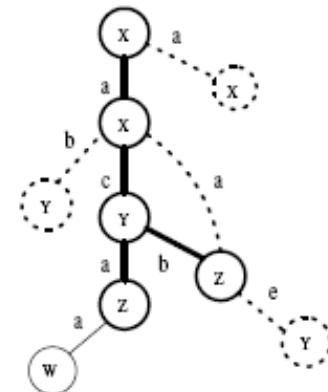
(a)

Database graph



(b)

potential children



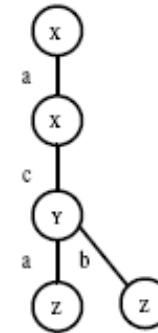
(c)

The gSpan Algorithm - Pruning

The $s \neq \min(s)$ Pruning (cont.):

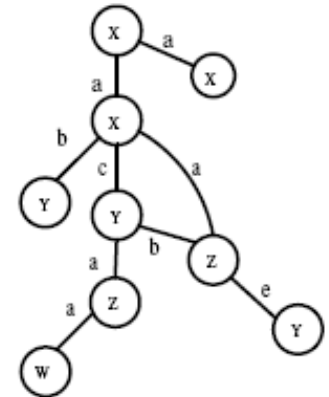
- 3) Edges which grow from other than the rightmost path are pruned
example: edge (z,a,w) is pruned
- 4) Post-pruning is applied to the remaining unpruned nodes

Frequent subgraph



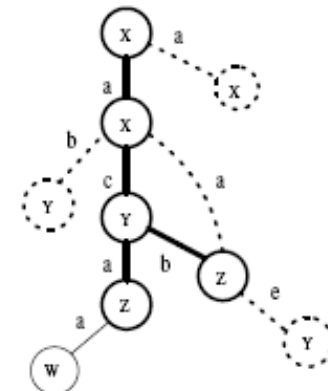
(a)

Database graph



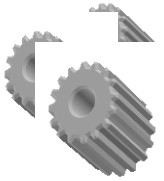
(b)

potential children

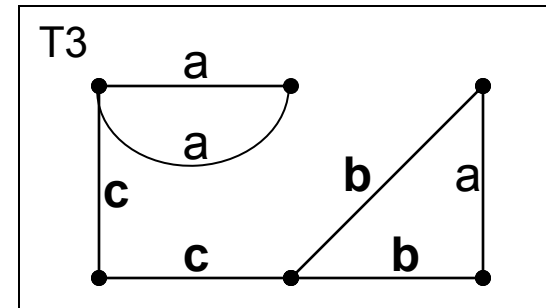
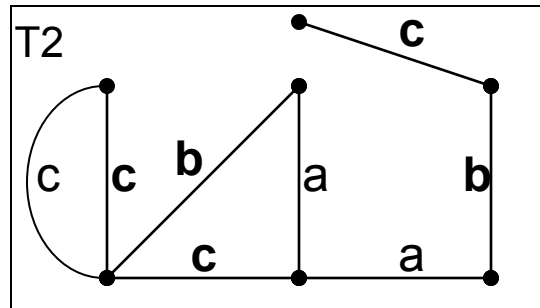
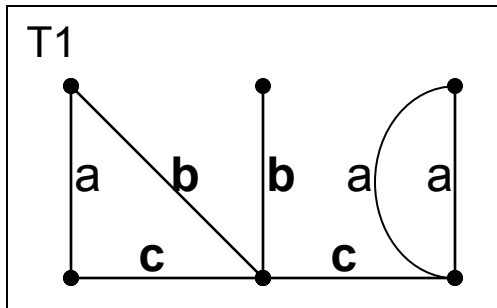


(c)

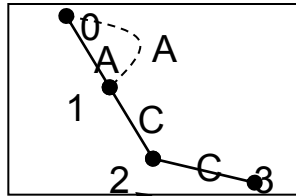
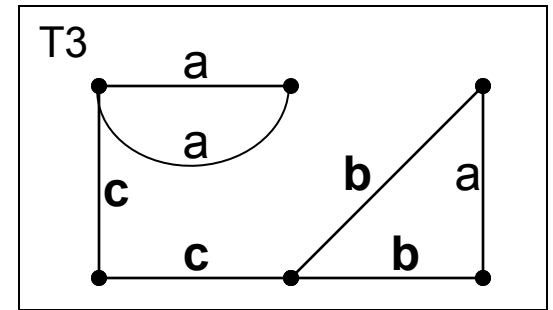
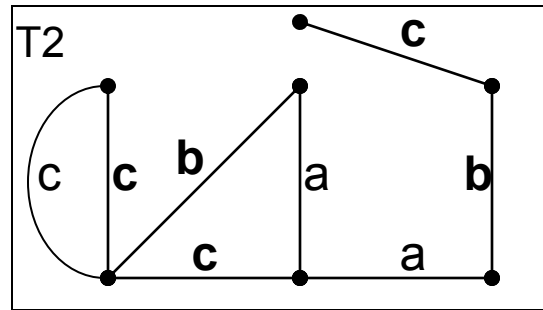
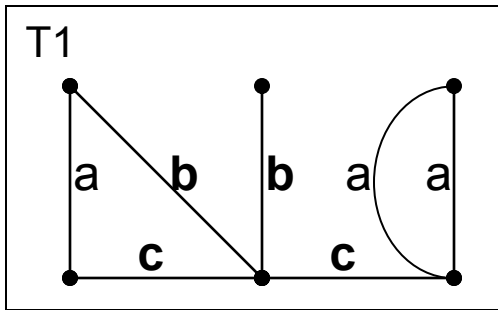
Another Example



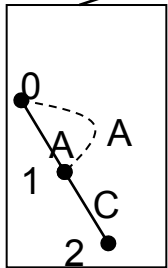
Given database ***D***



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



TID={1,3}

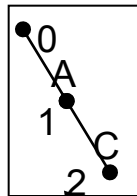
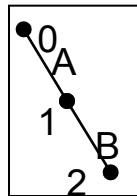
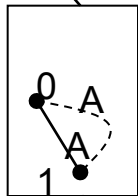


TID={1,3}

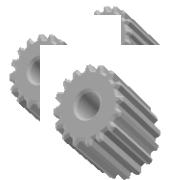
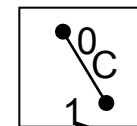
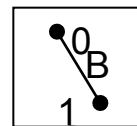
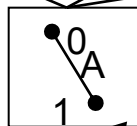
TID={1,3}

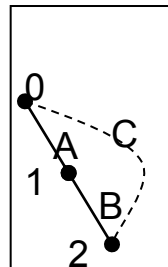
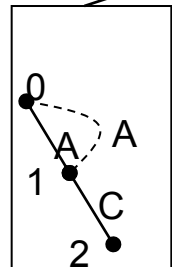
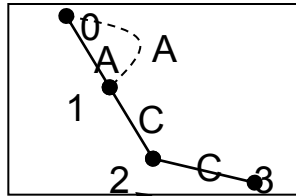
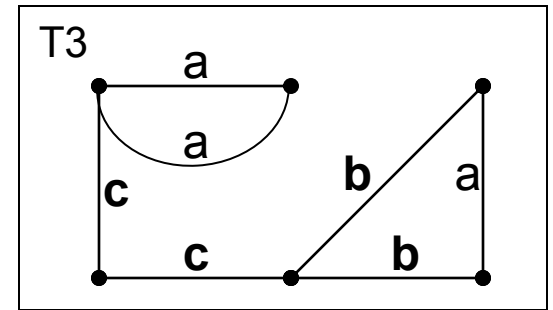
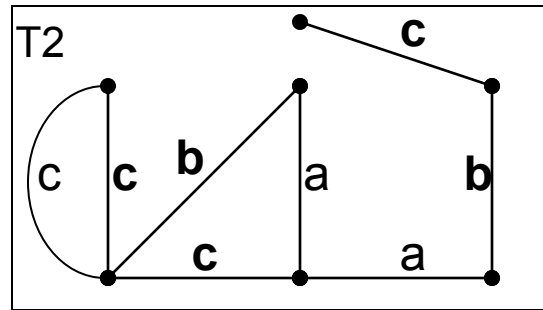
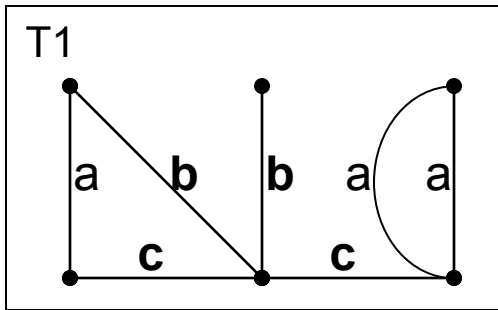
TID={1,2,3}

TID={1,2,3}



TID={1,2,3}

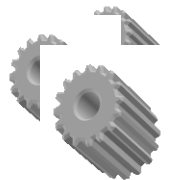
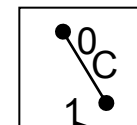
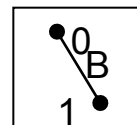
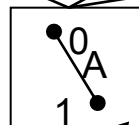
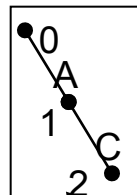
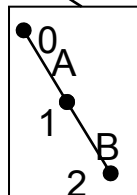
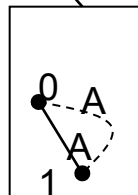


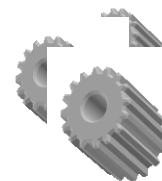
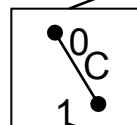
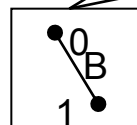
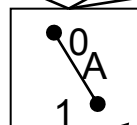
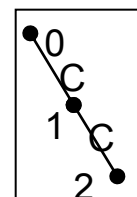
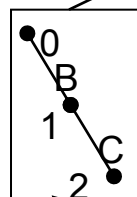
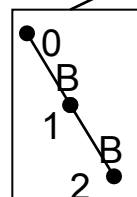
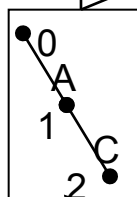
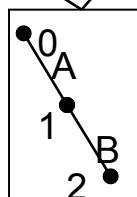
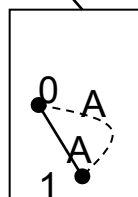
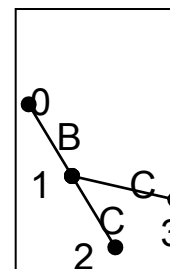
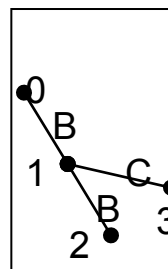
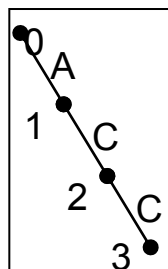
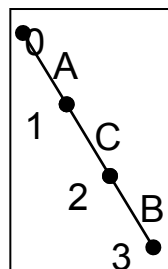
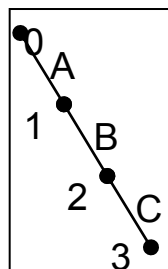
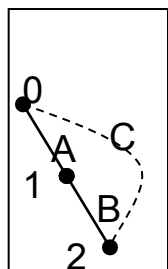
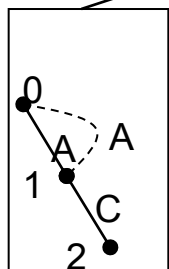
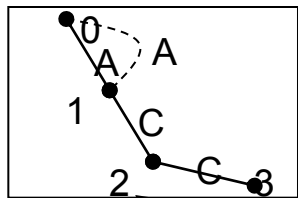
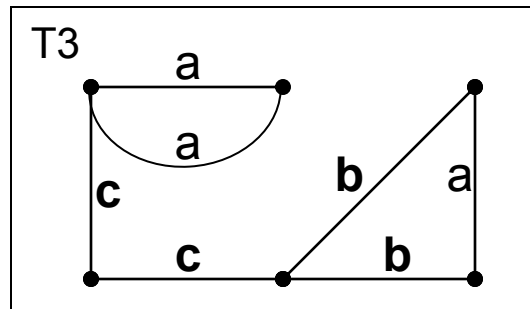
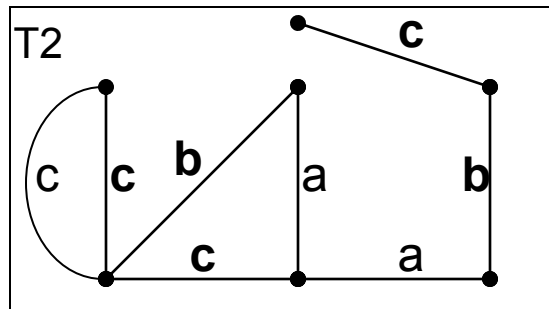
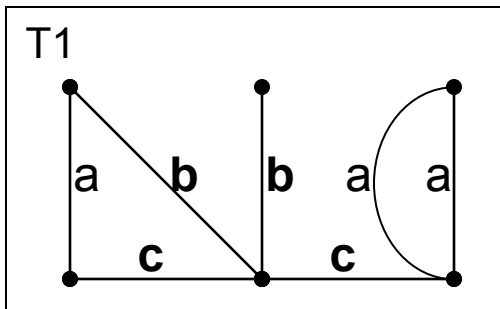


TID={1,2}

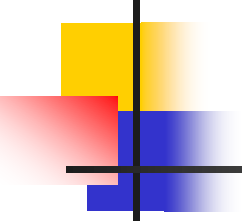
TID={1,2,3}

TID={1,2,3}





gSpan - Analysis

- 
- **No Candidate Generation and False Test** – the frequent $(k + 1)$ -edge subgraphs grow from k -edge frequent subgraphs directly
 - **Space Saving from Depth-First Search** – gSpan is a DFS algorithm, while Apriori-like ones adopt BFS strategy and suffers from much higher I/O and memory usage
 - **Quickly Shrunk Graph Dataset** – at each iteration the mining procedure is performed in such a way that the whole graph dataset is shrunk to the one containing a smaller set of graphs, with each having less edges and vertices

gSpan – Analysis(cont.)

- gSpan runtime measured by the number of subgraph and/or graph isomorphism (which is an NP-complete problem) tests:

$$O(kFS + rF)$$

↑ [bounds the maximum number of $s \neq \min(s)$ operations]
↑ [bounds the number of isomorphism tests that should be done]

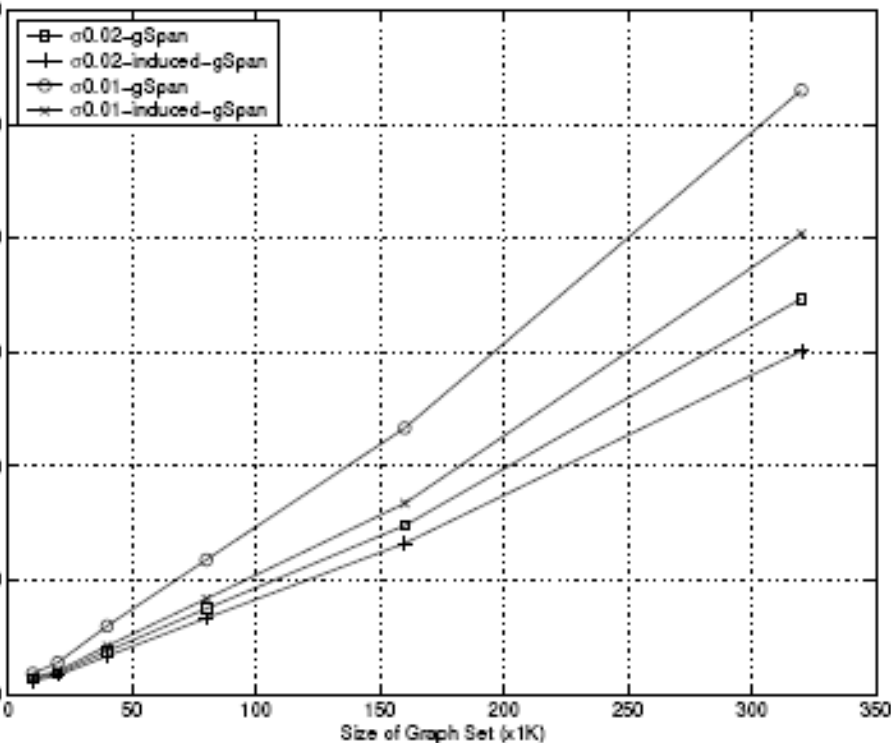
k – the maximum number of subgraph isomorphisms existing between a frequent subgraph and a graph in the dataset

F – the number of frequent subgraphs

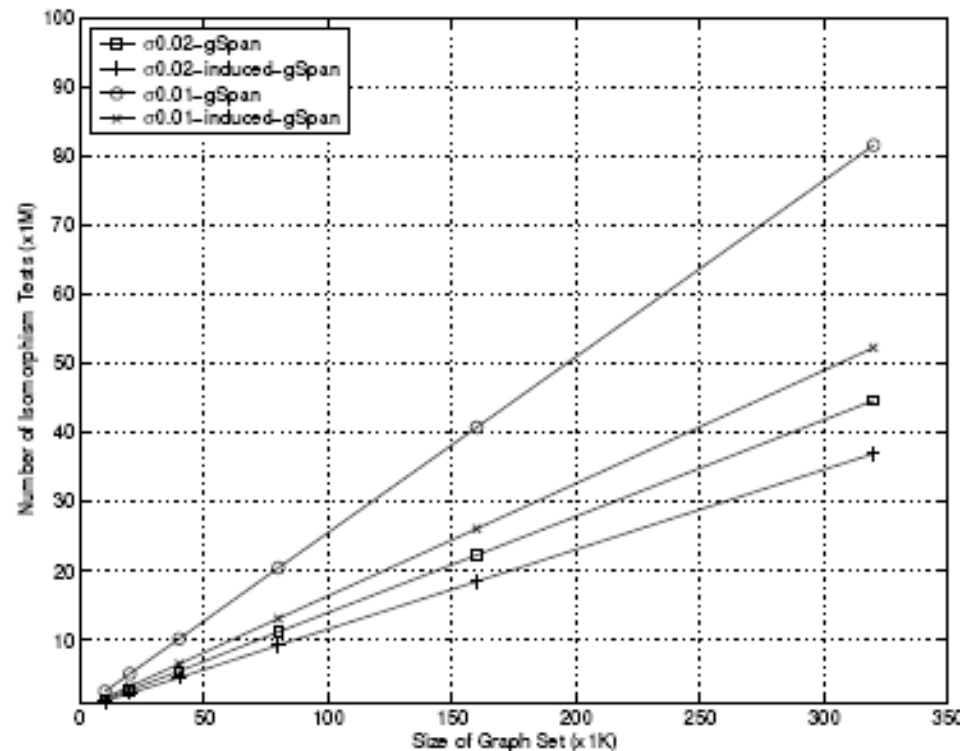
S – the dataset size

r – the maximum number of duplicate codes of a frequent subgraph that grow from other minimum codes

gSpan Experiments



(a) runtime

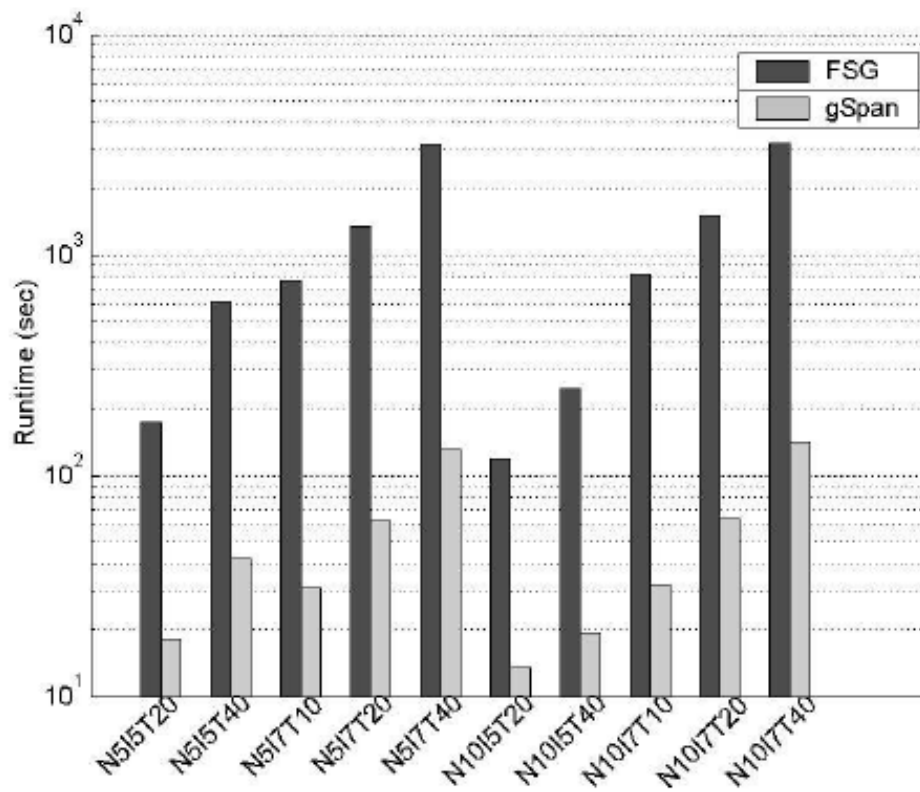


(b) # of isomorphism tests

Figure 6: Scalability on the Size of Graph Set (Synthetic Dataset)

gSpan Experiments

gSpan vs. FSG



(a) runtime on several data sets



gSpan Performance

- On Synthetic datasets it was 6-10 times faster than FSG
- On Chemical compounds datasets it was 15-100 times faster!
- But this was comparing to OLD versions of FSG!



GASTON (Nijssen and Kok, KDD'04)

- Extend graphs directly
- Store embeddings
- Separate the discovery of different types of graphs
 - path \rightarrow tree \rightarrow graph
 - Simple structures are easier to mine and duplication detection is much simpler



Different Approaches for GM

- Apriori Approach
 - AGM
 - FSG
 - Path Based (later)
- DFS Approach
 - gSpan
 - FFSM
- Diagonal Approach
 - DSPM 
- Greedy Approach
 - Subdue

Moti Cohen, Ehud Gudes

Diagonally Subgraphs Pattern Mining.

DMKD 2004, pages 51-58, 2004

Diagonal Approach & DSPM Algorithm



- Diagonal Approach is a general scheme for frequent pattern mining
- DSPM is an algorithm for mining frequent graphs which is based on the Diagonal Approach
- The algorithm combines ideas from Apriori & DFS approaches and also introduces several new ones



DSPM – Hybrid Algorithm

Operation	Similar to
Candidates Generation	BFS
Candidates Pruning	BFS
Search Space exploration	DFS
Enumerating Subgraphs	DFS

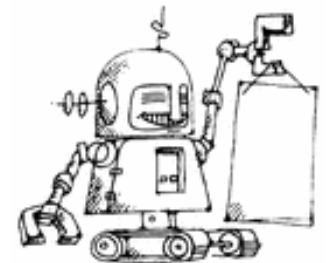
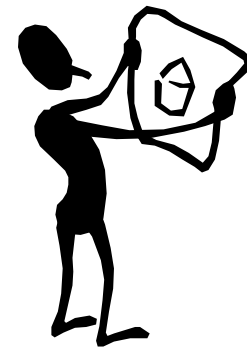
Concepts / Outline

Diagonal Approach

- Prefix based Lattice
- Reverse Depth Exploration

DSPM Algorithm

- Fast Candidate Generation & Frequency Anti-Monotone (FAM) Pruning
- Deep Depth Exploration
- Mass Support Counting





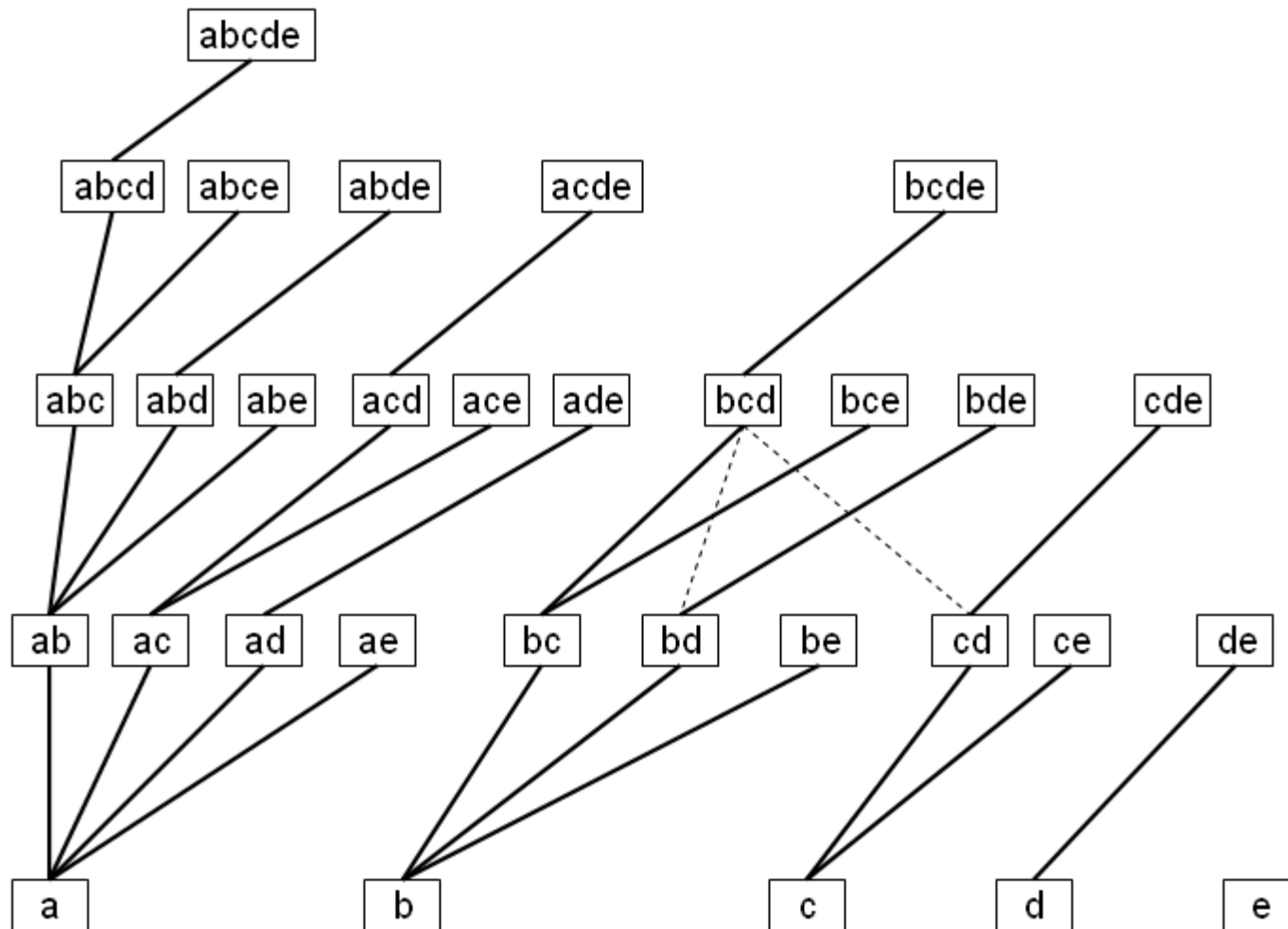
Definition: Prefix Based Lattice

- **Let** $\tau \in \{\text{itemsets, sequences, trees, graphs}\}$ be a frequent pattern problem
- τ -order is a complete order over the patterns
- τ -space is a search space of the τ problem which has a tree shape

Notation $\text{subpatterns}(p^k) = \{ p^{k-1} \mid p^{k-1} \text{ is a subpattern of } p^k \}$

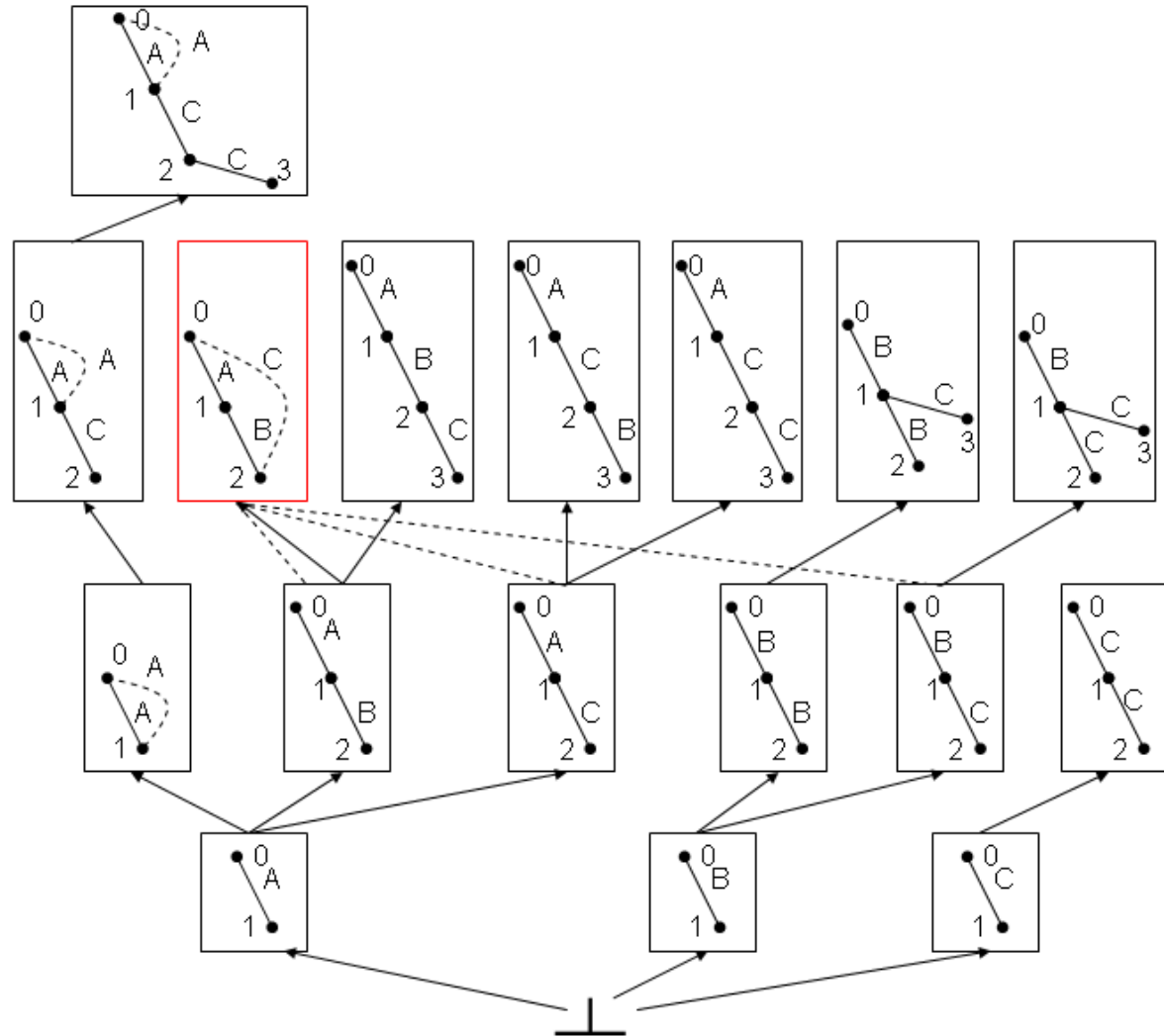
- **Then**, a τ -space is Prefix Based Lattice of τ if
- The parent of each pattern p^k , $k > 1$, is the minimum τ -order pattern from the set $\text{subpatterns}(p^k)$
- An in-order search over τ -space follows ascending τ -order
- The search space is complete

Example: Prefix Based Lattice (Itemsets)



Example: Prefix Based Lattice (Subgraphs)

[gSpan Algorithm of X. Yan, J. Han – an instance of PBL]





Reverse Depth Exploration

- Depth search over τ -space explores the sons of each visited node (pattern) in a descending τ -order

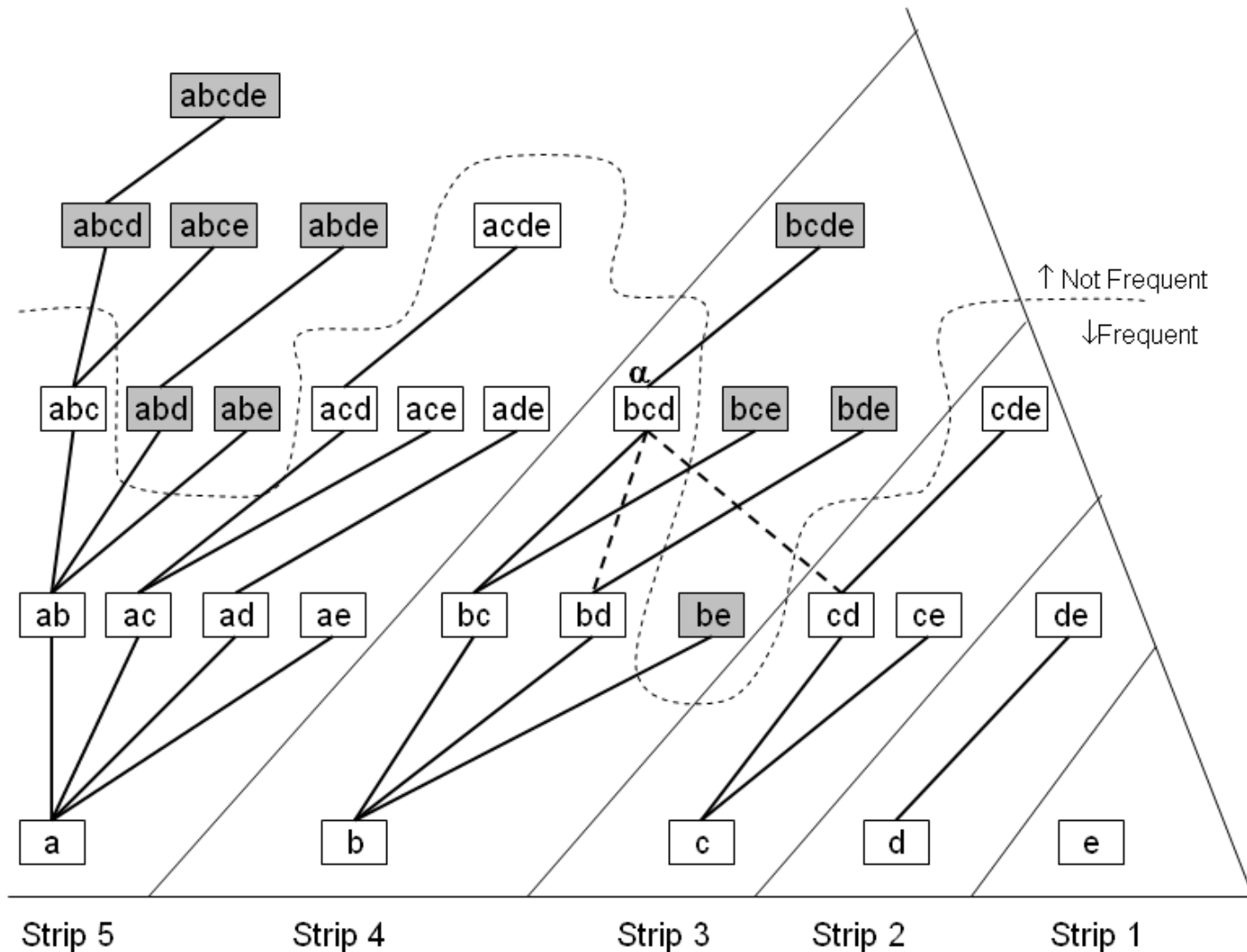


Observation

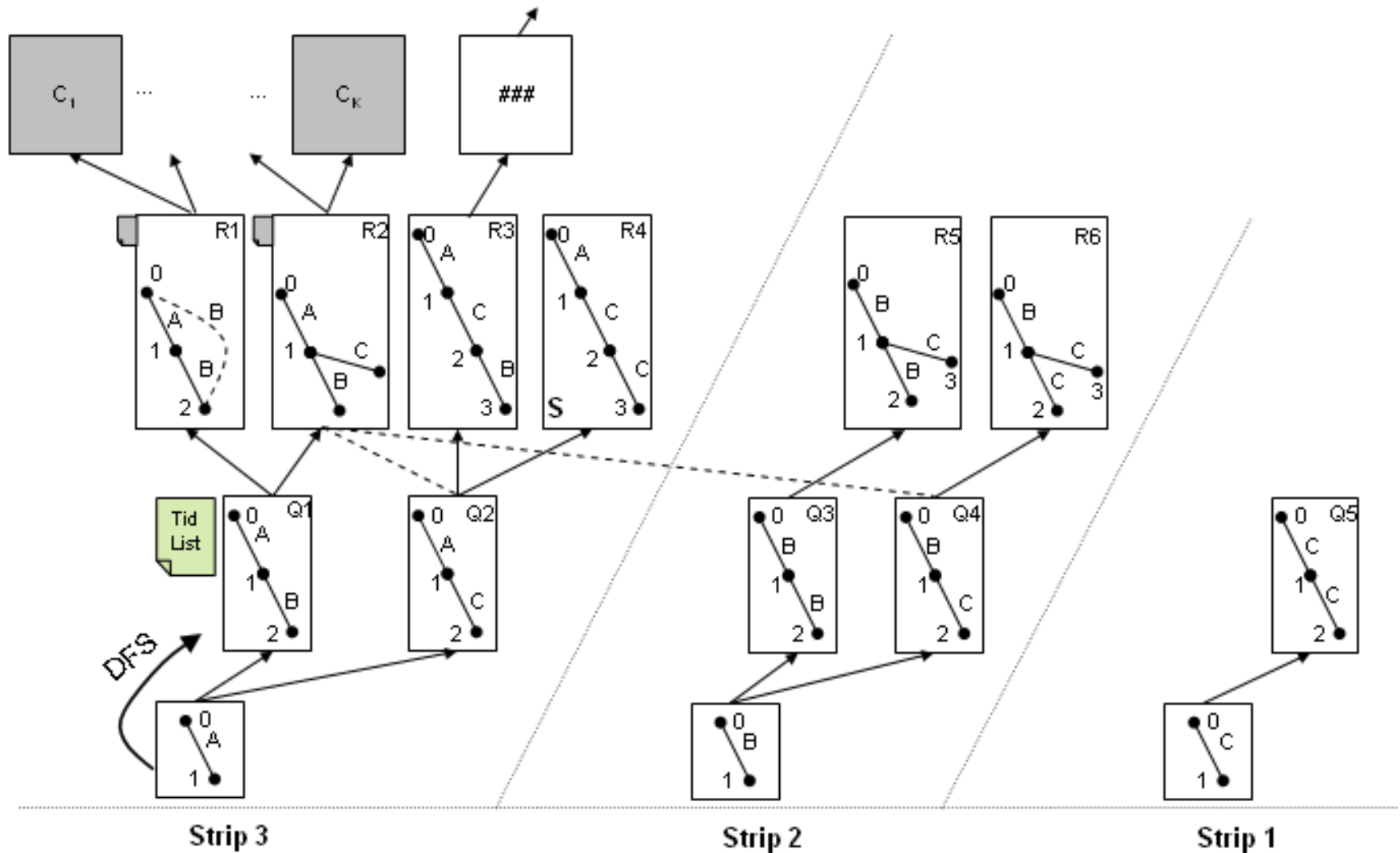
- Exploring prefixed based τ -space in reverse depth search enables checking Frequency Anti-Monotone (FAM) property for each explored pattern, if all previous mined patterns are kept.

Reverse Depth exploration + FAM Pruning

(Intuition wrt. Itemset)



Reverse Depth exploration + FAM Pruning



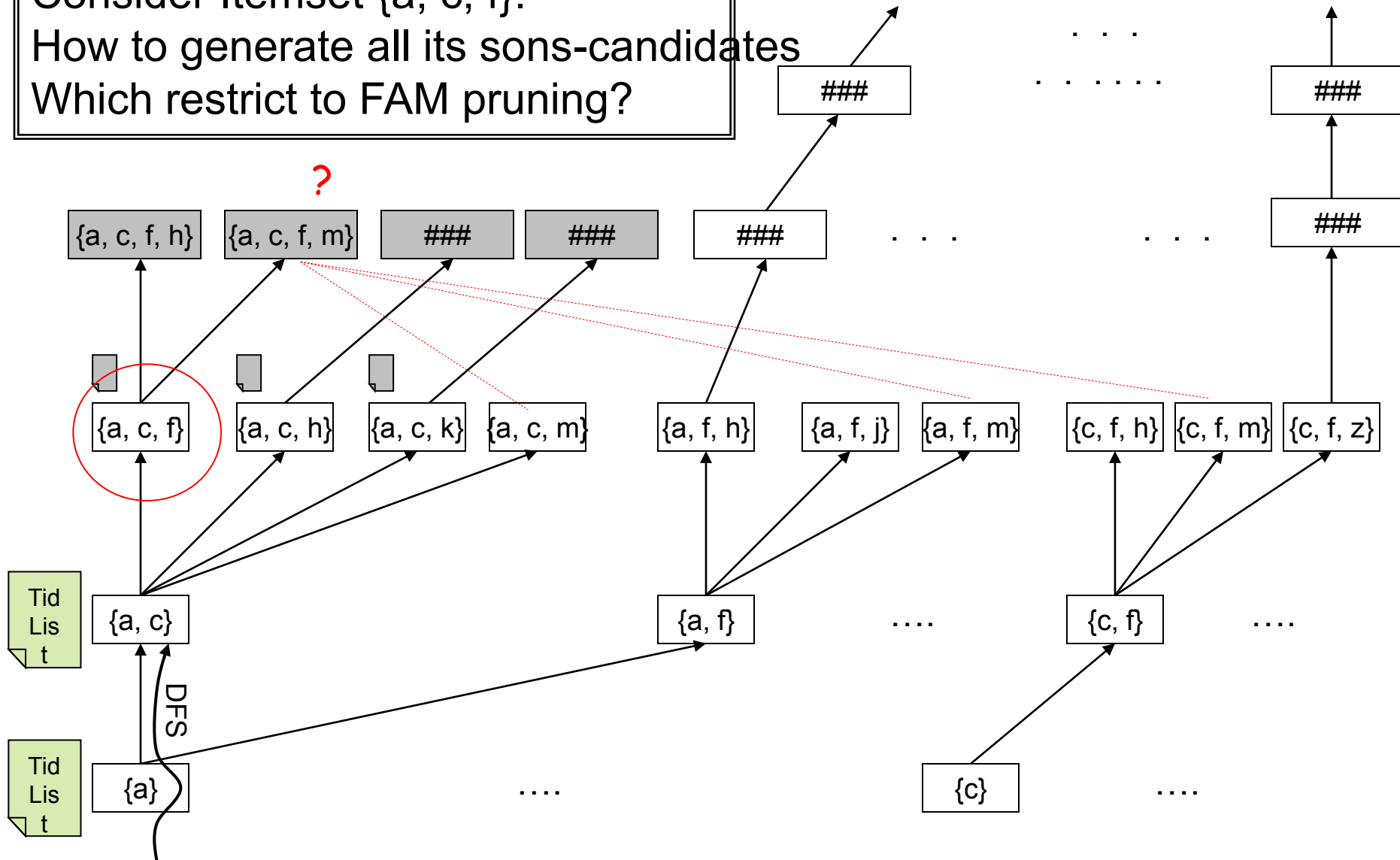
Fast Candidate Generation & FAM Pruning

(The idea wrt. Itemset)

Consider Itemset $\{a, c, f\}$.

How to generate all its sons-candidates

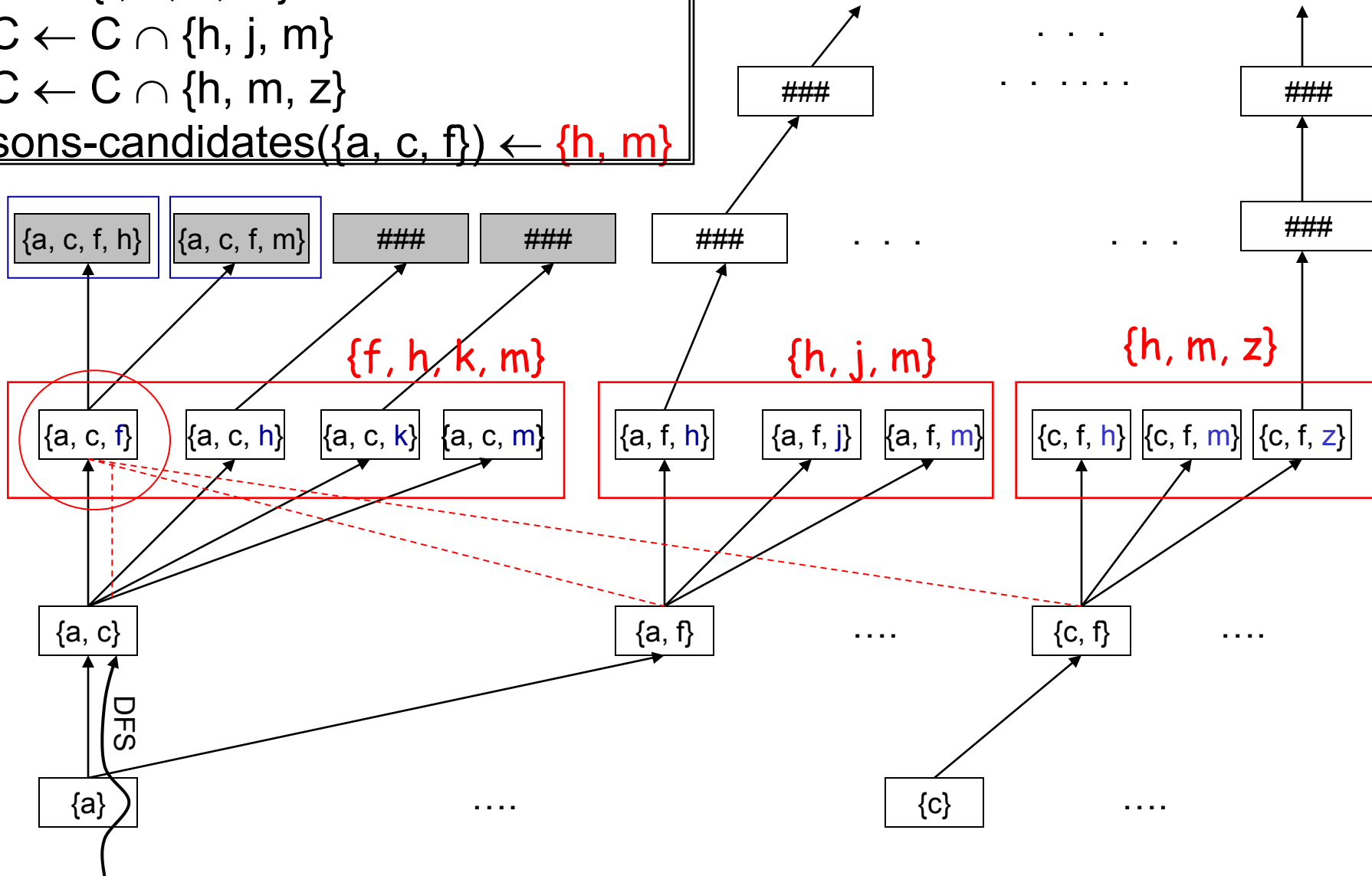
Which restrict to FAM pruning?



Fast Candidate Generation & FAM Pruning

(intersect the respective lists)

$C \leftarrow \{f, h, k, m\}$
 $C \leftarrow C \cap \{h, j, m\}$
 $C \leftarrow C \cap \{h, m, z\}$
 $\text{sons-candidates}(\{a, c, f\}) \leftarrow \{h, m\}$





Fast Candidate Generation & FAM Pruning

- DSPM algorithm adapted this idea to generate and prune subgraphs candidates.

This technique of candidate generation and FAM Pruning is highly efficient.

- Outcomes

- More space can be explored each iteration.
- More efficient support counting.



Performance of DSPM

- Was about twice better than gSpan on a synthetic database



Different Approaches for GM

- Apriori Approach
 - AGM
 - FSG
 - Path Based
- DFS Approach
 - gSpan
 - FFSM
- Diagonal Approach
 - DSPM
- Greedy Approach
 - Subdue



D. J. Cook and L. B. Holder
Graph-Based Data Mining
Tech. report, Department of
CS Engineering, 1998



Subdue Algorithm

- A greedy algorithm for finding some of the most prevalent subgraphs.
- This method is not complete, as it may not obtain all frequent subgraphs, although it pays in fast execution.



Subdue Algorithm (Cont.)

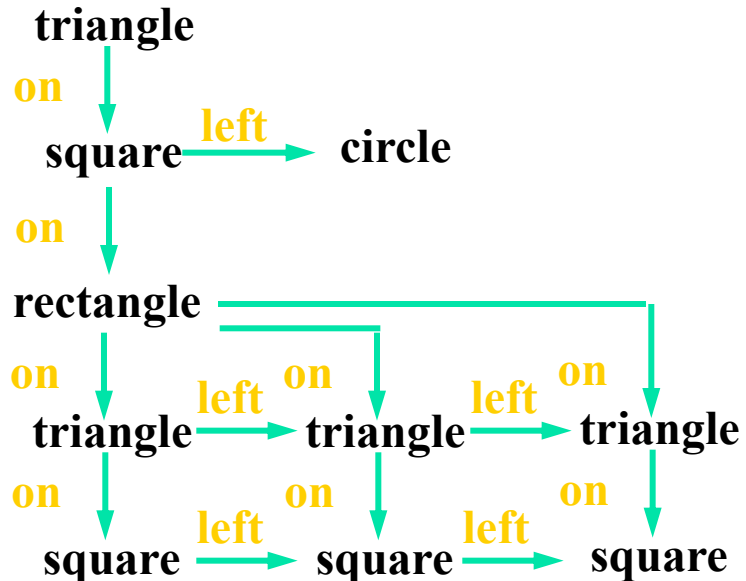
- It discovers substructures that compress the original data and represent structural concepts in the data.
- Based on *Beam Search* - Like breadth-first search in that it progresses level by level. Unlike breadth-first search, however, beam search moves downward only through the best W nodes at each level. The other nodes are ignored.



Subdue Algorithm steps

Step 1: Create substructure for each unique vertex label

DB:



Substructures:

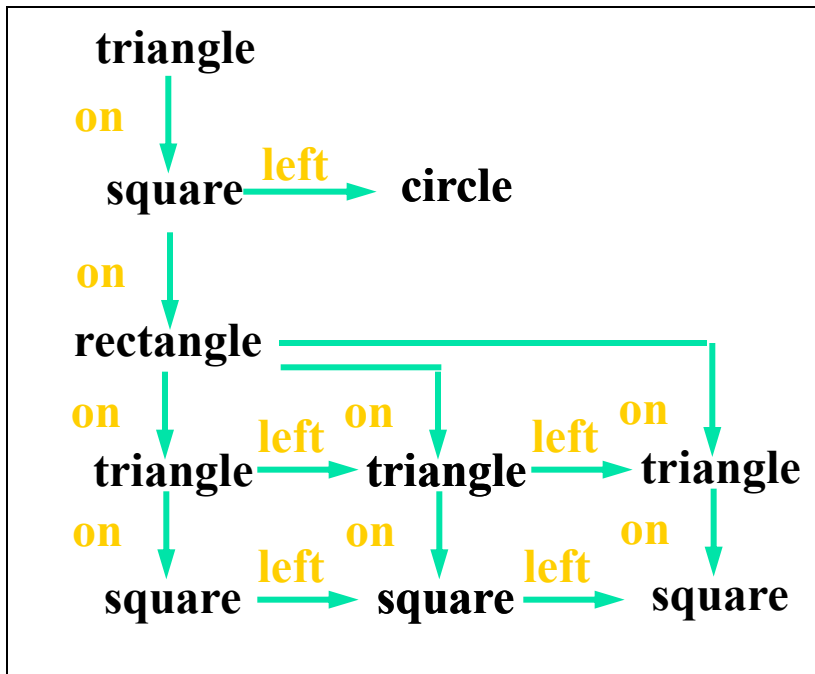
triangle (4)
square (4)
circle (1)
rectangle (1)



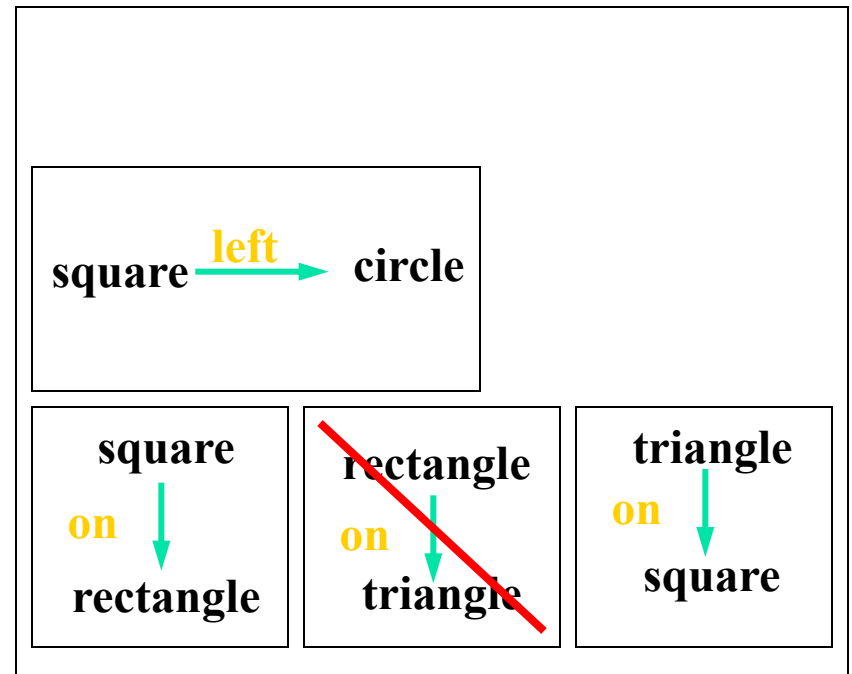
Subdue Algorithm steps (Cont.)

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

DB:



Substructures:





Subdue Algorithm steps (Cont.)

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



Agenda

- Introduction
- Problem Definition
- FSG
- gSpan
- Scalable mining of large Disk-based DBs
(Wang et. Al. – KDD 2004)



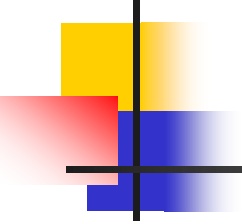
Motivation

- Graph Mining has very broad applications
 - Mining structural patterns from chemical compounds
 - Plan databases
 - XML Documents (on semantic web)
 - Citation/social networks
- But these are really large datasets:
 - XML Documents
 - Semantic web is www size, plus metadata
 - Hundreds or even thousands of different labels for data
 - Chemical Structures
 - Millions of different structures
 - Easily hundreds of labels in these graphs

Motivation - Previous Approaches

- Many approaches to this exist already
 - Most assume that databases are not very large
 - Assume that the entire database fits into main memory
 - Computation-centric
 - Perform poorly on larger datasets that are I/O bound
 - gSpan as an example (Yan, et al.)
 - Performance is reported for data sets up to only 320 KB
 - Test machine has 448MB main memory
 - Running time scales **exponentially** with large numbers of graph labels (raising from 10 to 45 labels, increases runtime by a factor of 84)
- Not effective on large datasets!

Frequent Operations

- 
- Major data access operations in mining frequent graph patterns (specifically in gSpan's):
 1. Given an edge, find its support in the graph database
 2. Given an edge, find the actual graphs where the edge appears in the database
 3. Given an edge, find the adjacent edges (to expand the current graph pattern)
 - gSpan typically needs **random access** to elements of the graph database and to its projections
 - Don't want to have to go to disk for each of these operations

ADI (Adjacency Index) Structures

■ Linked List of Graph id's

- Graph id's for a particular edge stored contiguously
- Efficient to retrieve all of them from memory at once
 - Facilitates operation 2: retrieve graphs of which an edge is a member
- The length of this list is stored in the edge table
 - Facilitates operation 1: support query for edge

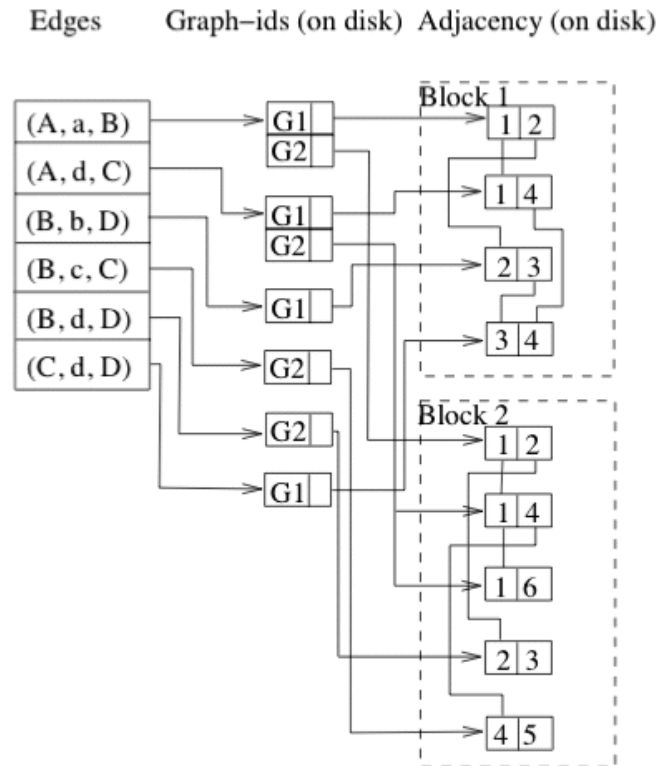


Figure 3: An ADI structure.

Space requirements

$$O(\sum_{i=1}^n |E(G_i)|)$$

- Total size of ADI is bounded by number of edges in all graphs:
 - Generally smaller than this
 - Graphs are often sparse on edges
 - Users typically only interested in frequently occurring edges.
- Not all of the ADI need be in memory
 - Can store bottom 1-3 levels on disk, if needed.

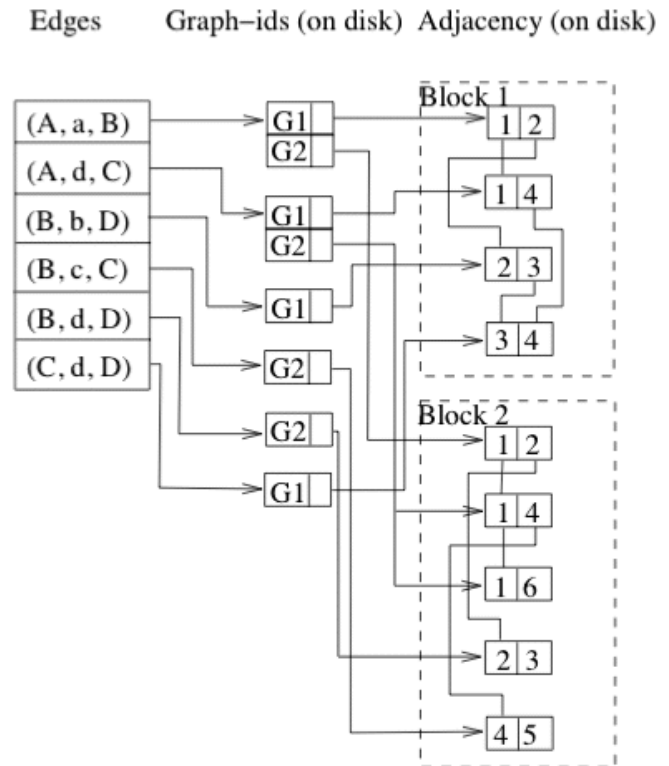


Figure 3: An *ADI* structure.

Constructing the ADI

- Requires only 2 passes through the database
 - Identify frequent edges
 - Creates edge table
 - Read and process graphs one by one
 - Builds graph lists
 - fills in adjacency info
- 2 major costs are
 - Adjacency lists = cost of copying original DB + bookkeeping
 - Updating graph id lists needs random access to edge table and linked lists
 - Needs good caching of lists to be efficient
- Can be expensive, but only needs to be done once.

Constructing the ADI

Input: a graph database GDB and min_sup

Output: the ADI structure

Method:

```

scan  $GDB$  once, find the frequent edges;
initialize the edge table for frequent edges;
for each graph do
    remove infrequent edges;
    compute the minimum DFS code [11];
    use the DFS-tree to encode the vertices;
    store the edges in the graph onto disk and form
        the adjacency information;
    for each edge do
        insert the graph-id to the graph-id list
        associated with the edge;
        link the graph-id to the related adjacency
        information;
    end for
end for

```

Figure 4: Algorithm of ADI construction.

Edges Graph-ids (on disk) Adjacency (on disk)

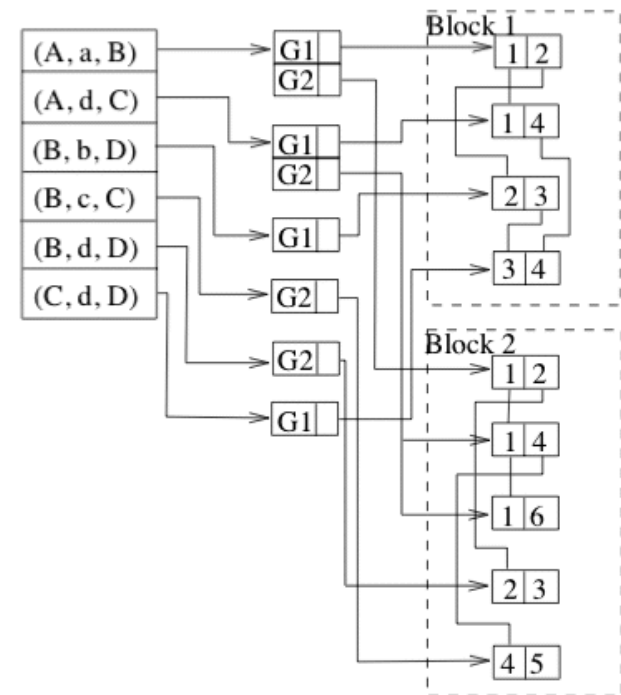


Figure 3: An ADI structure.



Algorithm ADI-Mine

A pattern growing algorithm – improvement of gSpan:

- First constructs the ADI structure if it doesn't already exist
- Obtain frequent edges from edges table in the ADI
- Use these edges' frequent adjacent edges to grow larger frequent graph patterns

Algorithm ADI-Mine

Input: a graph database GDB and min_sup

Output: the complete set of frequent graph patterns

Method:

```
construct the  $ADI$  structure for the graph database if
it is not available;
for each frequent edge  $e$  in the edge table do
    output  $e$  as a graph pattern;
    from the  $ADI$  structure, find set  $F_e$ , the set of
    frequent adjacent edges for  $e$ ;
    call  $subgraph-mine(e, F_e)$ ;
end for
```

Procedure $subgraph-mine$

Parameters: a frequent graph pattern G , and

the set of frequent adjacent edges F_e

// output the frequent graph patterns whose

// minimum DFS-codes contain that of G as a prefix

Method:

```
for each edge  $e$  in  $F_e$  do
    let  $G'$  be the graph by adding  $e$  into  $G$ ;
    compute the DFS code of  $G'$ ; if the DFS code is
    not minimum, then return;
    output  $G'$  as a frequent graph pattern;
    update the set  $F_e$  of adjacent edges;
    call  $subgraph-mine(G', F_e)$ ;
end for
return;
```

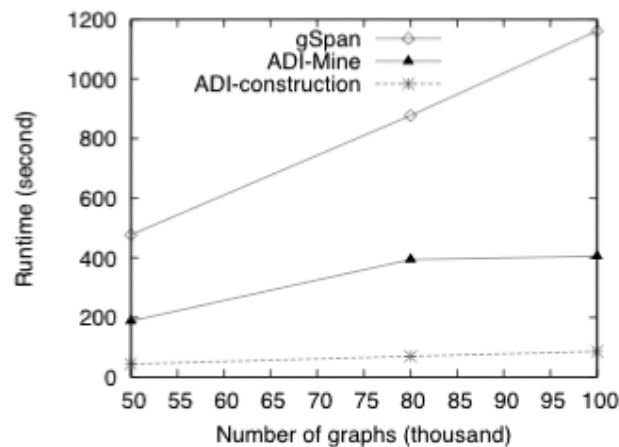
Figure 6: Algorithm $ADI-Mine$.

Differences with gSpan

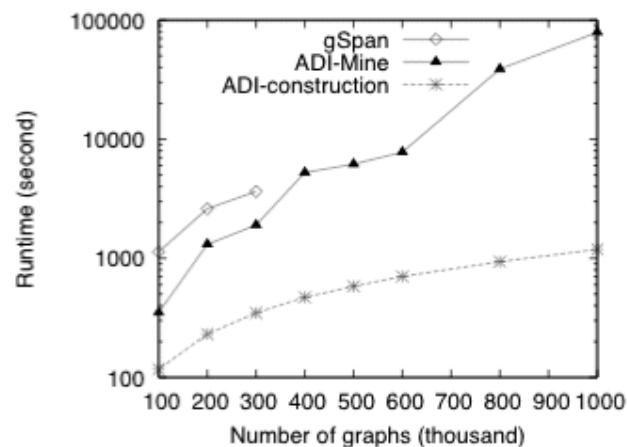
- gSpan loads graphs into memory repeatedly and checks if they contain particular edges
 - Can end up searching more than we need to by loading graphs that may not have the edge we're looking for
 - Really, bigger issue is that this loads the graph into memory, and it's costly to go to disk.
- ADI-Mine can simply go straight through the edge table, by the label of the edge it's searching for
 - Graphs we need are readily available
 - Located in contiguous memory
 - No extra searching and no loading of unnecessary graphs from disk (in large databases)

Scalability on size

Memory and large on-disk DB's, respectively



(b) Scalability on size
D50-100kN30I5T20L200
min_sup = 1%

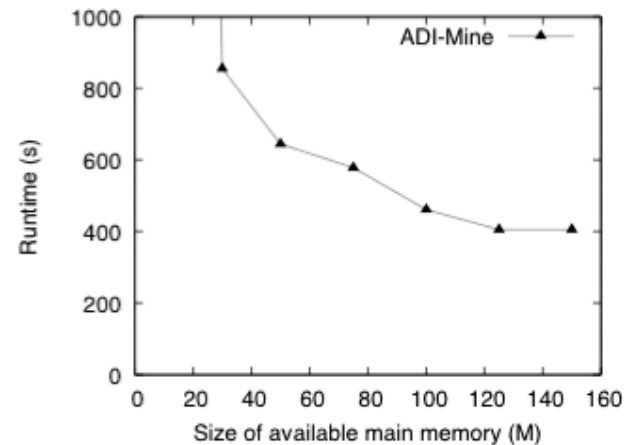


(a) scalability on size
D100k-1mN30I5T20L200
min_sup = 1%

Note at right that gSpan is unable to work on datasets larger than about 300K graphs

Runtime vs. main memory

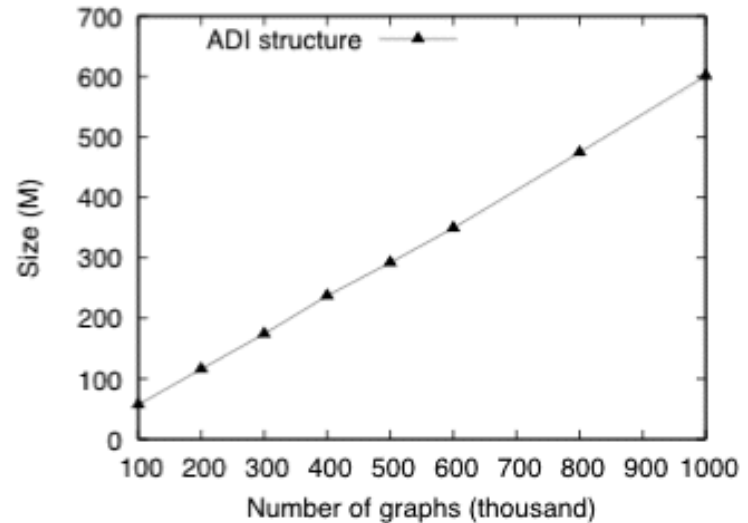
- Runtime vs. main memory for large, disk-based runs
 - We're probably swapping pages (in the B-tree) more frequently at the lower memory sizes, so performance suffers.
 - Performance converges when we can fit the working set in memory



(c) Runtime vs. main memory
D100kN30I5T20L200
min_sup = 1%

ADI size


Size of the ADI structure grows linearly with amount of data ■



(b) Size of *ADI* structure
D100k-1mN30I5T20L200
min_sup = 1%



Outline

- Basic concepts of Data mining and Association rules
 - Apriori algorithm
- Motivation for Graph mining
- Applications of Graph Mining
- Mining Frequent Subgraphs - Transactions
 - BFS/Apriori Approach (FSG and others)
 - DFS Approach (gSpan and others)
 - Diagonal Approach
 - Greedy Approach
- Mining Frequent Subgraphs – Single graph 
 - The support issue
 - The Path-based algorithm
 - Constraint-based mining
- Conclusions