Graph Mining

SubGraph

A graph g is a subgraph of another graph g′ if there exists a **subgraph isomorphism** g --> g′.
Same Vertices with related same edges.

The goal and challenges in frequent subgraph mining

Goal: Reduce the number of subgraph isomorphism detections (Because it's NP-C)

The general 2-step framework for frequent subgraph mining

Step1: Generate frequent substructure candidates
Step2: Check frequency of each candidate (Subgraph isomorphism test: NP-C)

Frequent Substructure mining 2 methods:

Graph first , then subGraph

Method 1: Apriori-based Approaches (Edge-based generation)

- Graph isomorphism detection: Line (2) —> Line (4)
- Subgraph isomorphism detection: Line (5) —> Line (6)

Method 2: Pattern-Growth Approaches (gSpan) **Add following codes between (5) & (6)**

- Graph isomorphism detection: If s◊_re Isequals(minDFS(s◊_re))
- Subgraph isomorphism detection: add s◊_re to C && count its frequency

Apriori-Based Approaches Only BFS (low-wise)	**Pattern-Growth Approaches BFS/DFS**	**gSpan**
Inputs: D, a graph data set; min_sup, the minimum support threshold.	Inputs: g, a frequent graph; D, a graph data set; min_sup, minimum support threshold.	Inputs: s, a DFS code; D, a graph data set; min sup, the minimum support threshold.
Method: $S_1 \leftarrow$ frequent single-elements in the data set; Call AprioriGraph(D, min_sup, S_1); **procedure** AprioriGraph(D, min_sup, S_k) (1) $S_{k+1} \leftarrow \varnothing$; (2) **for each** frequent $g_i \in S_k$ **do** (3) **for each** frequent $g_j \in S_k$ **do** (4) **for each** size $(k+1)$ graph g formed by the merge of g_i and g (5) **if** g is frequent in D and $g \notin S_{k+1}$ **then** (6) insert g into S_{k+1}; (7) **if** $s_{k+1} \neq \varnothing$ **then** (8) *AprioriGraph*(D, min_sup, S_{k+1}); (9) **return;**	**Method:** $S \leftarrow \varnothing$; Call PatternGrowthGraph(g, D, min_sup, S); **procedure** PatternGrowthGraph(g, D, min_sup, S) (1) **if** $g \in S$ **then return;** (2) else insert g into S; (3) scan D once, find all the edges e such that g can be (4) **for each** frequent $g \diamond_x e$ **do** (5) *PatternGrowthGraph*($g \diamond_x e$, D, min_sup, S); (6) **return;** g_{xe}: new formed graph f:forward b:backward Stopped once no frequent graph can be generated	**Method:** $S \leftarrow \varnothing$; Call gSpan(s, D, min_sup, S); **procedure** PatternGrowthGraph(s, D, min_sup, S) (1) **if** $s \neq dfs(s)$, **then** (2) **return;** (3) insert s into S; (4) set C to $\varnothing$; (5) scan D once, find all the edges e such that s can be *right-most* extended to $s \diamond_r e$; insert $s \diamond_r e$ into C and count its frequency; (6) sort C in DFS lexicographic order; (7) **for each** frequent $s \diamond_r e$ in C **do** (8) *gSpan*($s \diamond_r e$, D, min_sup, S); (9) **return;**
Outputs: S_k, the frequent substructure set.		

Apriori-Based Approaches

Edge-based generation

A new candidate is generated by adding an extra edge.

A & B: Frequent subgraphs C_i: Candidate graphs   k: number of edge

size-(k-1) graph A + size(k-1) graph B = multiple possible size-k graphs C_i

Apriori Principles (association rule)

Any subset of a **frequent** itemset must be **frequent**.
Any subgraph of a **frequent** graph must be **frequent**.

Any superset of an **infrequent** itemset must be **infrequent**.
Any supergraph of an **infrequent** graph must be **infrequent**.

Why need to require a large number of graph isomorphism detection?

Need to do:

mergeable frequent subgraph detections & candidate duplication detections.

Pattern-Growth Approaches

Edge-based generation

A new candidate is generated by extending an extra edge. k: number of edge

size-(k+1) candidate graphs are generated by extending frequent size-k subgraphs.

DFS for pattern-growth Approach

- A graph G subscripted with a DFS tree T is written as GT ,T is called a **DFS subscripting** of G.
- Given a DFS tree T:
  - starting vertex in T , **v0** , the root.
  - last visited vertex, **vn**, the *right-most* vertex.
- DFS code: Transform each *subscripted* graph to an *edge sequence*, so that we can build an **order** among these sequences.
  - **Edge order**: which maps edges in a subscripted graph into a sequence. **DFS Code.**
    - A **DFS code** can **uniquely** identify an augmented DFS tree and hence, a graph. Treat DFS codes the same as their corresponding graphs.
  - **Sequence order**: which builds an order among edge sequences (i.e., graphs). **Lexicographic Order.**
  - **Goal:** is to select the subscripting that generates the minimum sequence as its base subscripting. **Minimum DFS code.**

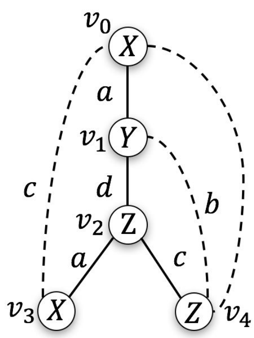
DFS **Edge Order** method

Step 1:

- Order all the forward edges by the DFS traversal ordering.

Step 2:

- Place every backward edge (v_j, v_i) with $j > i$ right after the forward edge (ending at v_j).
- For multiple backward edges (v_j, v_{i1}) and (v_j, v_{i2}) starting from v_j with i1 < i2, place (v_j, v_{i1}) before (v_j, v_{i2}). Eg. 0 < 1, (v4, v0) is front of (v4, v1)

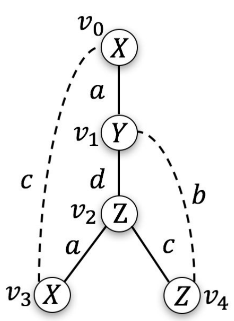


DFS tree

Step1	(v0, v1), (v1, v2), (v2, v3), (v2, v4)
Step2	(v0, v1), (v1, v2), (v2, v3), (v3, v0), (v2, v4),(v4, v0), (v4, v1)

DFS Code method

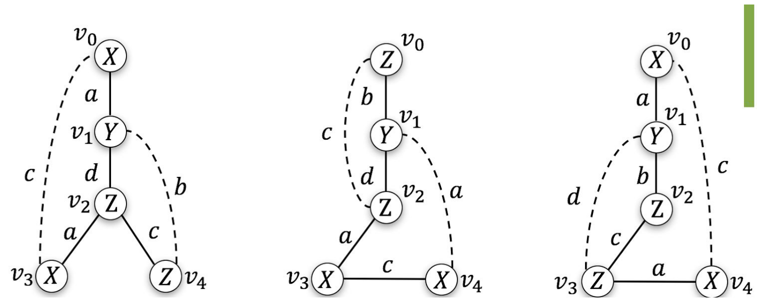
- Given a DFS code (e0, …, ei), the extended DFS code generated by a **right-most** extension has the form of (e0, …, ei, e(i+1)).
  - the **right-most extensions** can preserve the prefix of DFS codes.
- DFS Code Format: (i, j, Label-Vertex_i, Label-Edge, Label-Vertex_j)



Edge	DFS Code
e_0	$(0, 1, X, a, Y)$
e_1	$(1, 2, Y, d, Z)$
e_2	$(2, 3, Z, a, X)$
e_3	$(3, 0, X, c, X)$
e_4	$(2, 4, Z, c, Z)$
e_5	$(4, 1, Z, b, Y)$

DFS **Lexicographic Order** method

- Need to finish the DFS Edge Code first, then do the Lexicographic Order
- is the dictionary order of the **DFS codes** by treating each of them as an English word.

❖ 

Edge	DFS Code γ_1	DFS Code γ_2	DFS Code γ_3
e_0	$(0,1,X,a,Y)$	$(0,1,Z,b,Y)$	$(0,1,X,a,Y)$
e_1	$(1,2,Y,d,Z)$	$(1,2,Y,d,Z)$	$(1,2,Y,b,Z)$
e_2	$(2,3,Z,a,X)$	$(2,0,Z,c,Z)$	$(2,3,Z,c,Z)$
e_3	$(3,0,X,c,X)$	$(2,3,Z,a,X)$	$(3,1,Z,d,Y)$
e_4	$(2,4,Z,c,Z)$	$(3,4,X,c,X)$	$(3,4,Z,a,X)$
e_5	$(4,1,Z,b,Y)$	$(4,1,X,a,Y)$	$(4,0,X,c,X)$

❖ γ_3 < γ_1 < γ_2

❖ γ_1<γ_2 because (0, 1, X, a, Y)<(0, 1, Z, b, Y).

❖ γ_3<γ_1because (0, 1, X, a, Y)=(0, 1, X, a,Y) and (1, 2, Y, b, Z)<(1,2,Y,d,Z). *If equal, compare next edges until scanned all.*

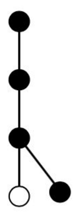
Minimum DFS **Code** method

❖ Denote minDFS(G)

❖ Given two graphs G and G′, G is isomorphic to G′ if and only if minDFS(G)=minDFS(G′).

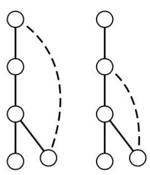
❖ Computing minDFS(G) is at least as hard as the graph isomorphism detection

❖ **Minimum DFS codes** are only generated by the **right-most extensions** on the minimum DFS code prefixes.



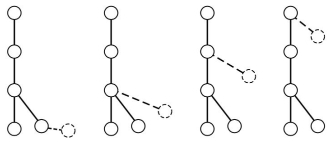
What is right-most extension?

Backward extension: a *new edge* e can be **added** between the ***right-most vertex*** and another **vertex** on the *right-most* path.



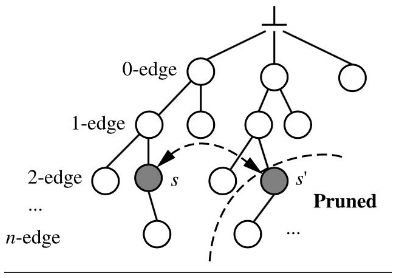
Forward extension: it can introduce *a new vertex* and **connect** to a **vertex(***can be a* ***right-most vertex*****)** on the *right-most* path.

Because both kinds of extensions take place on the right-most path, we call them right-most extension, denoted by G ◊r e (for brevity, T is omitted here).



Is it necessary to perform **right-most** extension on **non-minimum** DFS codes ?

No.If codes s and s′ encode the same graph, the search space under s′ can be safely prune.



Lexicographic search tree.

How to reduce the generation of duplicate graph for pattern-growth Approach?

Use **gSpan algorithm**. Each frequent graph should be extended as conservatively(保守的) as possible.

Apriori-Based Approaches Advantage Vs. Disadvantage

Advantage	**Disadvantage**
AprioriGraph greatly reduces the number of subgraph isomorphism detections.	Requires a large number of graph isomorphism detection

Apriori-Based Approaches Vs. Pattern Growth

Apriori	**Pattern Growth**
• It utilizes the **Apriori principle** to generate candidates. • Require a large number of graph **isomorphism** detections.	• Perform **right-most extensions** on the minimum DFS codes to reduce duplicate generations. • By the minimum DFS codes, it can reduces the number of graph **isomorphism** detections. (K.O Apriori)

Pattern Growth Vs. gSpan

Pattern Growth	**gSpan**

• Generation and detection of a duplicated graph increase workload. • Non-efficient.	• No need to search previously discovered frequent graphs for duplicate detections. (Because of **minimum DFS code**) • does not extend any duplicate graph, yet still guarantees the discovery of the complete set of frequent graphs. • DFS in less memory