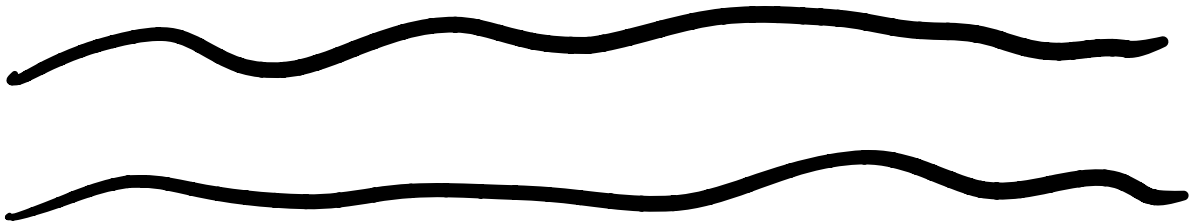
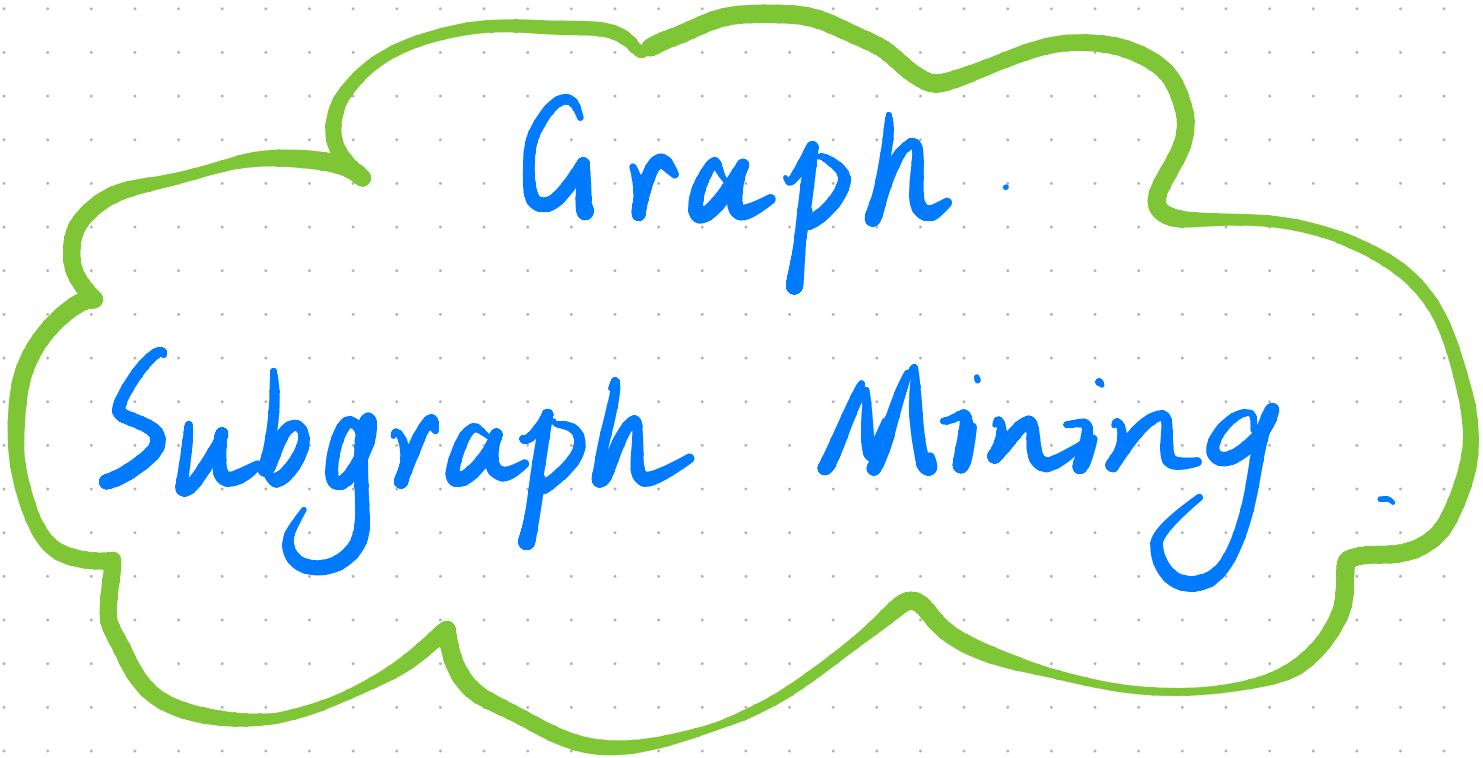
Graph



Content

1. Naive Graph
2. gSpan


Graph
Subgraph Mining

NaiveGraph (g, D, min_sup, S)

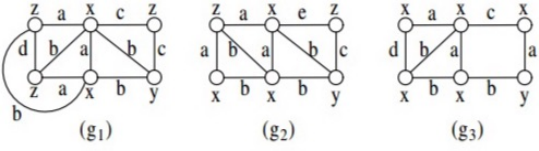


Figure 1: A Sample Graph Dataset D

Input : A graph g, a graph dataset D, and min_sup

Output : The frequent graph set S

1. if g exists in S, then RETURN
2. else insert g to S
3. scan D once, find every edge e, such that g can be extended to $g \diamond_x e$ and it is frequent
4. for each frequent $g \diamond_x e$ do
5. Call NaiveGraph ($g \diamond_x e$, D, min_sup, S)
6. RETURN

inefficiency of extending g to $g \diamond_x e$

for an n-edge graph, it may have n different ways to be formed from (n-1) edges if we do not consider isomorphism.

NaiveGraph

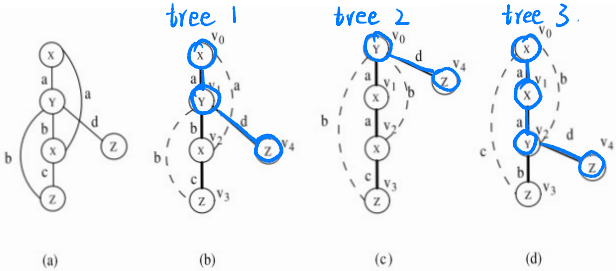
reason →

gSpan

(a more clever way to extend graphs & pruning)

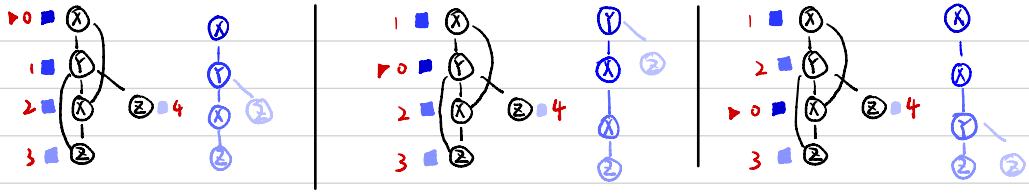
In Algorithm 1, NaiveGraph requires extending g in any possible position, which will result in a huge number of duplicate graphs. We would like to show that there is a more clever way to extend graphs. gSpan restricts the extension as follows: Given g and a DFS tree T in g, e can be extended from the right-most vertex connecting to any other vertices on the right-most path (*backward extension*); or e can be extended from vertices on the right-most path and introduce a new vertex (*forward extension*). We call these two kinds of restricted extension as *right-most extension*, denoted by $g \diamond_r e$ (for simplicity, we omit T here). This restricted extension is different from $g \diamond_x e$ described in NaiveGraph.

right-most extension ($g \diamond_r e$)



choosing different nodes as roots to build different trees.

Figure 2: Depth-First Search Tree and its Forward/Backward Edge Set



$v_0 - v_1 - v_2 - v_3 - v_4$

X - Y - X - Z - Z

Y - X - X - Z - Z

X - X - Y - Z - Z.

rightmost path : straight path from the root to the last one.

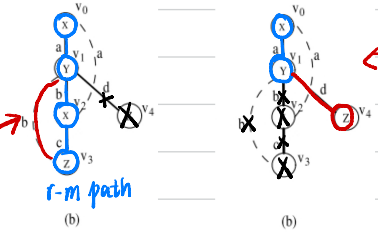
$v_0 - v_1 - v_4$

$v_0 - v_4$

$v_0 - v_1 - v_2 - v_4$

v_0 : root (first node to be extended)

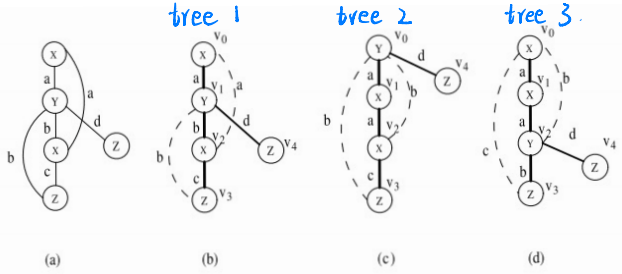
v_1 : second node to be extended



DFS extension

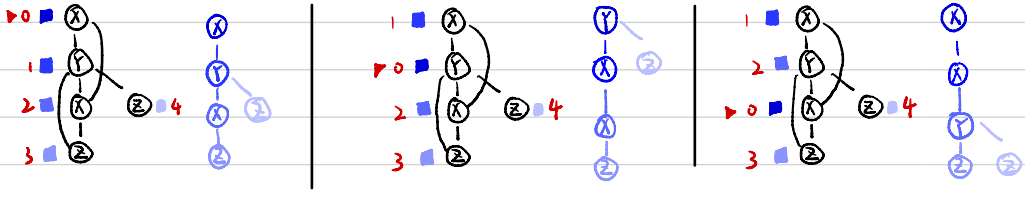
- backward extension: edge from right-most vertex connecting to any other vertices on the right-most path.
- forward extension: edge from vertices on the right-most path and introduce a new vertex

as follows: Given g and a DFS tree T in g, e can be extended from the right-most vertex connecting to any other vertices on the right-most path (*backward extension*); or e can be extended from vertices on the right-most path and introduce a new vertex (*forward extension*). We call these two kinds of restricted extension as *right-most extension*, denoted by $g \diamond_r e$ (for simplicity, we omit T here). This restricted extension is different from $g \diamond_x e$ described in NaiveGraph.



choosing different nodes as roots to build different trees.

Figure 2: Depth-First Search Tree and its Forward/Backward Edge Set

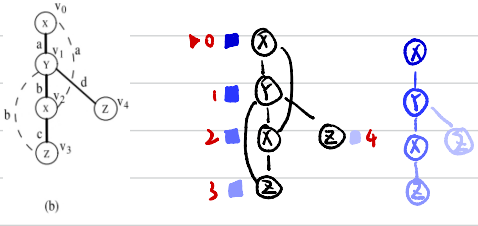


Extension Method

for vertex v.

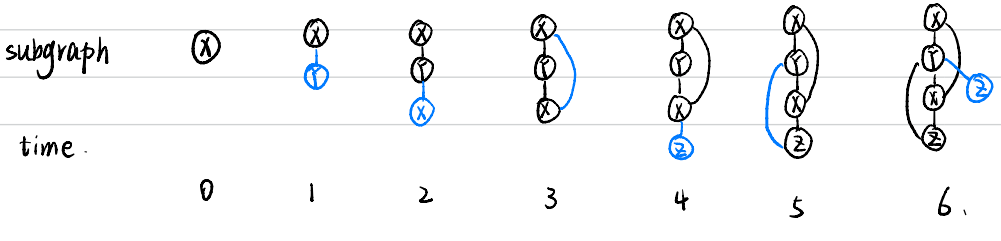
1. all of its backward edges should appear just before its forward edges.
2. if v does not have any forward edge, we put its backward edges just after the forward edge where v is the second vertex.

How a tree extend?



edge no.	(b) α
0	$(0, 1, X, a, Y)$
1	$(1, 2, Y, b, X)$
2	$(2, 0, X, a, X)$
3	$(2, 3, X, c, Z)$
4	$(3, 1, Z, b, Y)$
5	$(1, 4, Y, d, Z)$

Table 1: DFS code for



How to mine from a graph set

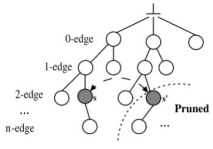


Figure 3: Search Space

Algorithm 2 GraphSet_Projection($\mathbb{GS}$,$\mathbb{S}$).

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: $\mathbb{S}^1 \leftarrow$ all frequent 1-edge graphs in $\mathbb{GS}$;
5: sort $\mathbb{S}^1$ in DFS lexicographic order;
6: $\mathbb{S} \leftarrow \mathbb{S}^1$;
7: **for each** edge $e \in \mathbb{S}^1$ **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($\mathbb{GS}$, $\mathbb{S}$, s);
10: $\mathbb{GS} \leftarrow \mathbb{GS} - e$;
11: **if** $|\mathbb{GS}| < minSup$;
12: **break**;

Step 1 (line 1-6): Remove infrequent vertices and edges from the graph set $\mathbb{GS}$. Relabel the remaining ones in descending frequency. Add all frequent 1-edge graphs into $\mathbb{S}^1$ and sort them in DFS lexicographic order. For example, we have a label set {A,B,C,...} for vertices, {a,b,c,...} for edges. Each 1-edge graph has only one edge such as $(0, 1, A, a, A)$, $(0, 1, A, a, B)$, ..., and the like. According to DFS lexicographic order, $(0, 1, A, a, A) < (0, 1, A, a, B) < \ldots$. Since we

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

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

pruning

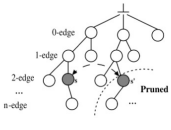


Figure 3: Search Space

According to the definition of Minimum DFS code, the first occurence of DFS code of a graph is its minimum DFS code.
=> if s and s' represente the same graph, (包含哪些边) 判断
s' is pruned

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**
$c.GS \leftarrow c.GS \cup \{g\}$;
4: **if** g covers all children of s **break**;

Explaination →

By enumerating all occurrences (more accurately until all possible children are discovered) of s in each graph, Enumerate(s) (Subprocedure 2) counts the occurrences of all the children of s in the graph. ~~For example, Figure 4(a) shows a frequent subgraph discovered through previous routines,~~

Example →

~~graph.~~ For example, Figure 4(a) shows a frequent subgraph discovered through previous routines, the possible children are shown in Figure 3(b)-3(f). Figure 4(b) is a graph in a graph dataset. In Figure 4(c), a thickened line or cycle marks an occurrence of 4(a) in 4(b). Dotted lines and cycles are illustrated as potential candidates of 4(a)'s children (with one more edge). By Property 1, (W, a, Z) is not a valid child. By counting the occurrences of the children over all the graphs where s takes place, we know which child is a frequent subgraph too. ~~Furthermore, we record IDs of all~~

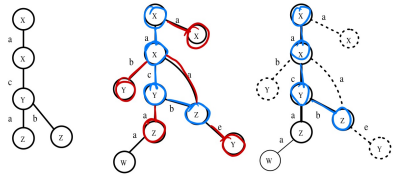


Figure 4: Subgraph Growing

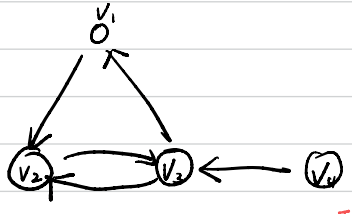
edge no.	(b) α	(c) β	(d) γ
0	$(0, 1, X, a, Y)$	$(0, 1, Y, a, X)$	$(0, 1, X, a, X)$
1	$(1, 2, Y, b, X)$	$(1, 2, X, a, X)$	$(1, 2, X, a, Y)$
2	$(2, 0, X, a, X)$	$(2, 0, X, b, Y)$	$(2, 0, Y, b, X)$
3	$(2, 3, X, c, Z)$	$(2, 3, X, c, Z)$	$(2, 3, Y, b, Z)$
4	$(3, 1, Z, b, Y)$	$(3, 0, Z, b, Y)$	$(3, 0, Z, c, X)$
5	$(1, 4, Y, d, Z)$	$(0, 4, Y, d, Z)$	$(2, 4, Y, d, Z)$

minimum DFS code.

Table 1: DFS code for Figure 2(b), 2(c), and 2(d)

Graph

Adajacency Matrix



A (adjacency matrix)

	1→1	1→2	1→3	1→4	
	0	1	0	0	1→4
	0	0	1	0	2→4
	1	1	0	0	3→4
	0	0	1	0	4→4

A^T

	1→1	2→1	3→1	4→1	
	0	0	1	0	4→1
	1	0	1	0	4→2
	0	1	0	1	4→3
	0	0	0	0	4→4

$A \cdot A^T = [b_{ij}]$ ij

b_{11} : 1→1 1→2 1→3 1→4

outdegree 1→1 1→2 1→3 1→4

$A^T \cdot A = [b_{ij}]$ ij

b_{11} : 1→1 2→1 3→1 4→1

indegree 1→1 2→1 3→1 4→1

$A^2 = [b_{ij}]$ ij

b_{12} : 1→1 1→2 1→3 1→4

1→2 2→2 3→2 4→2

从 $v_1 \to v_2$，长度为2的路有 b_{12} 条。