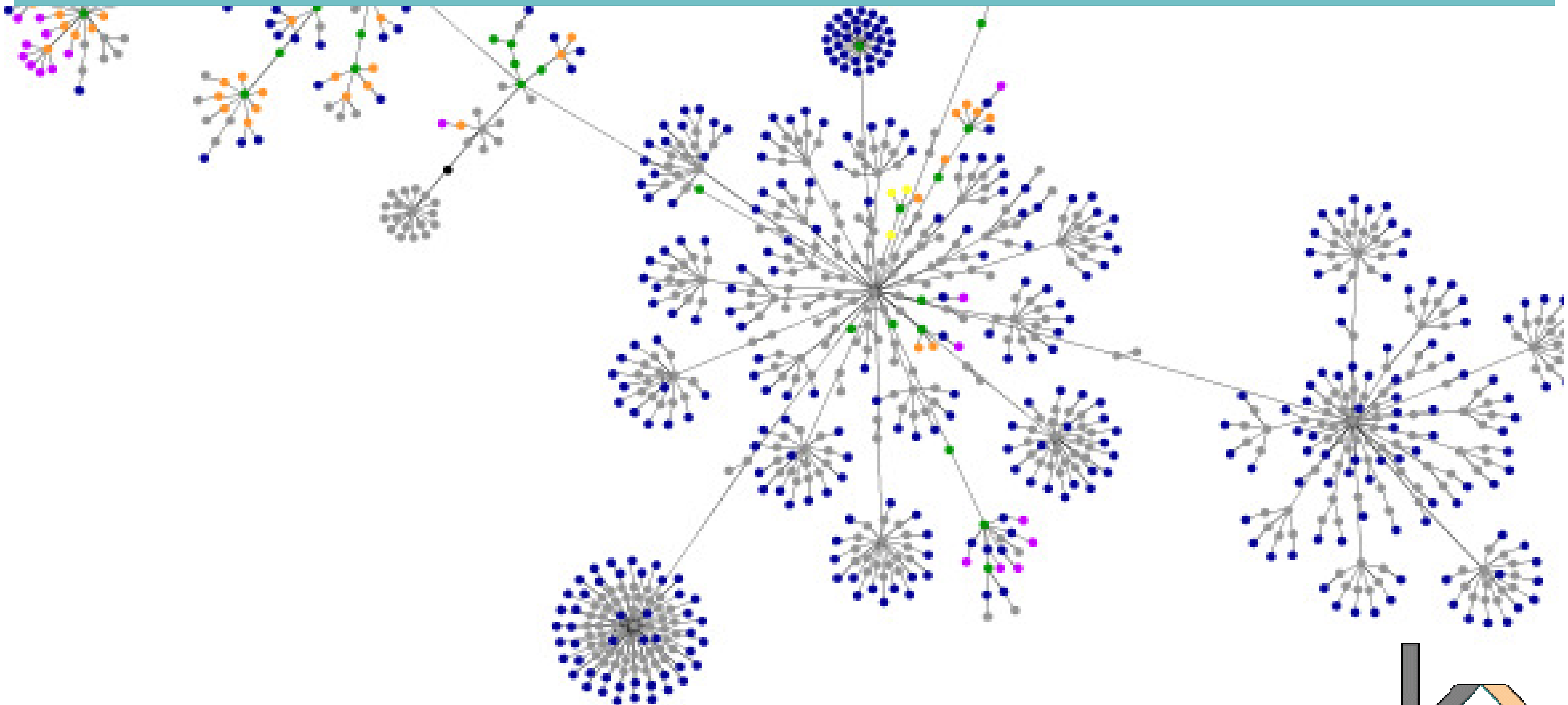
Mining frequent Closed Graph Pattern

Seminar aus maschninellem Lernen
Referent: Yingting Fan

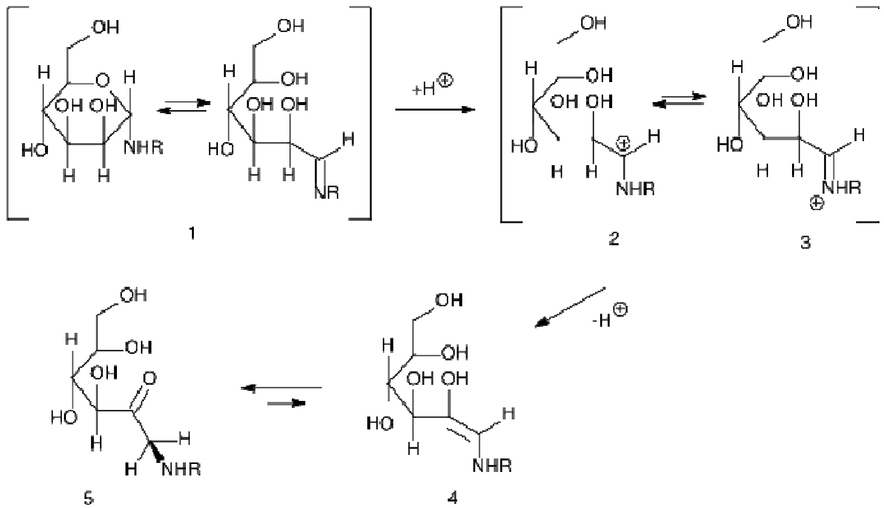


Outline

- Motivation and introduction
- gSpan
- ClosedGraph
- Conclusion
- Literatures
- Appendix

Motivation

- Data structure shows complicated relationship among the graph
- Wide application in bioformatics, Web exploration etc.
- Apriori-based candidate generation-and-test approach
- Pattern-growth philosophy



<u>Pattern Mining Algorithms:</u>

- **gSpan** (Graph-Based Substructure Pattern Mining)
- **Closed Graph** (Mining Closed Frequent Graph Patterns)

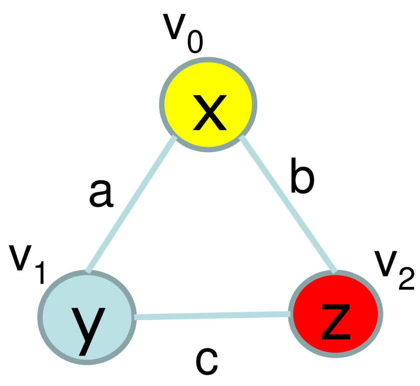
Introduction

- Graph Basics – Definition of a labeled graph
- Isomorphism and subgraph isomorphism
- Frequent Subgraph Mining
- DFS Lexicographic Ordering
  - DFS tree
  - Right-Most Extension
  - Linear order
  - DFS code
  - DFS Lexicographic Order and DFS code Tree

Graph Basic

- Data structure – labeled simple graph as a 4-tuple $G(V,E,L,\ell)$

  V set of vertices
  $E \subseteq V \times V$ set of edges
  L set of labels
  ℓ $V \cup E \rightarrow L$ a function mapping labels to the vertices and the edges.



Example:

$V = \{v_0, v_1, V_2\}$

$E = \{(v_0,v_1),(v_1,v_2),(v_2,v_0)\}$

$\ell_0 = X$, $\ell_{(o,1)} = a$

Isomorphism

- An isomorphism is a bijective funktion:
  f: V(G) → V(H)
  G und H are two data structures and l is the label function

$$
\begin{aligned}
& f: V(G) \rightarrow V(H) \\
& l_G(u) = l_H(f(u)) && \text{for } u \in V(G) \\
& \left.\begin{aligned} & (f(u), f(v)) \in E(H) \\ & l_G(u,v) = l_H(f(u), f(v)) \end{aligned}\right\} && \text{for } (u,v) \in E(G)
\end{aligned}
$$

- A subgraph isomorphism from graph G to graph H is a isomorphism from graph G to subgraph H

Frequent Subgraph Mining (1)

- Given: a graph dataset GS={G_i|i=0,...,n} und Mininum Support (MinSup)

$$\varsigma(g,G) = \begin{cases} 1 & \textit{if } g \textit{ is isomorphic to a subgraph of } G, \\ 0 & \textit{if } g \textit{ is not isomorphic to any subgraph of } G. \end{cases}$$

$$\sigma(g,GS) = \sum_{G_i \in GS} \varsigma(g,G_i)$$

- Support(g) presents the number of graphs in GS in which g is a subgraph
- Looking for:
  find all graphs in GS in which the support not less than MinSup

$$\sigma(g,GS) \geqq \text{MinSup}$$

Frequent Subgraph Mining (2)

- Subgraph Isomorphism test is NP-complete
- NaiveGraph is simple but ineffizient; Why?
  1. Generation (k+1)-edge graph from k-edge graph
  2. Checking the frequency of these candidates

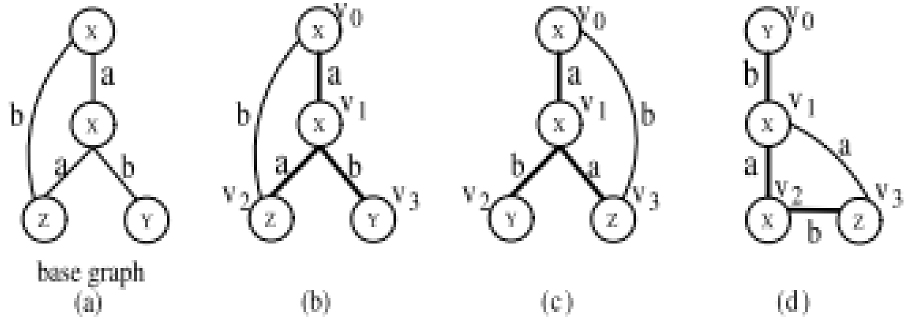
- Construction of canonical DFS codes based on DFS Tree
  1. It reduced the generation of duplicate graphs
  2. No need to search previous discovered frequent graphs
  3. Completeness guarantee without extending any duplicate

- First, every graph owns a canonical DFS code and the codes are equivalent for isomorphic graphs
- Later, Construction of DFS Code Tree
  → every vertice is a labeled graph

DFS Tree and this Subscripting

- DFS tree can be constructed with performing a depth-first search and DFS tree is not unique
- Mark vertices in their discovery time:
  $i<j$ means v_i is discovered before v_j
- Beginning vertex v_0 and Right-most vertex v_n
- Right-most path: straight way from v_0 to v_n

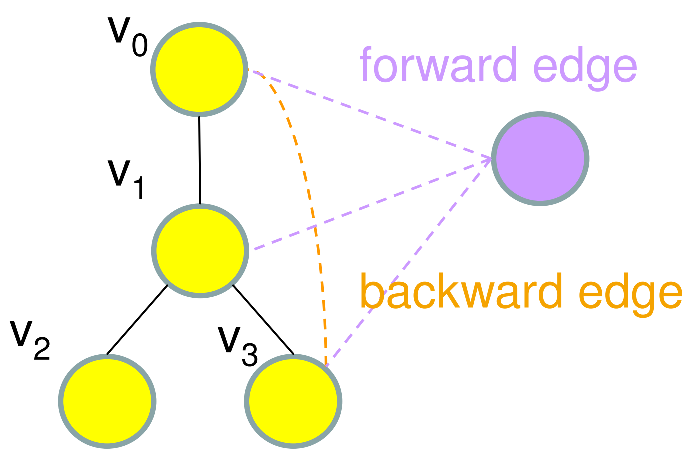
- Two kind of edge set:
  - Forward edge (i,j):
    $(v_i,v_j) \in E(G)$ and $i<j$
  - Backward edge (i,j):
    $(v_i,v_j) \in E(G)$ and $i>j$



In (c):
(0,1) is forward edge of v_0
(3,0) is backward edge of v_3

Right-Most Extension s*e

- Goal: reduce a huge number of duplicate graphs while extending a graph
- Right-most extension generated with two rules
  - Vertex can be extended from the right-most vertex connecting to any other vertices on the right-most path(backward extension)
  - Vertex can be extended from vertices on the right-most path an introduces a new vertex (forward extension)



Linear Order and DFS code (1)

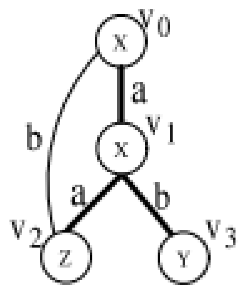
- DFS tree defines the linear order of forward edges and now we add backward edges into the order.
- With the following rule statement we create a linear order
  Assume: $e_1 = (i_1, j_1)$, $e_2 = (i_2, j_2) \rightarrow e_1 \prec e_2$

  (i) $e_1, e_2 \in E_T^f$, and $j_1 < j_2$ or $i_1 > i_2 \wedge j_1 = j_2$.
  (ii) $e_1, e_2 \in E_T^b$, and $i_1 < i_2$ or $i_1 = i_2 \wedge j_1 < j_2$.
  (iii) $e_1 \in E_T^b$, $e_2 \in E_T^f$, and $i_1 < j_2$.
  (iv) $e_1, \in E_T^f$, $e_2 \in E_T^b$, and $j_1 \leqslant i_2$.

- Linear order of edges is DFS codes

The linear order for figure (b) is:

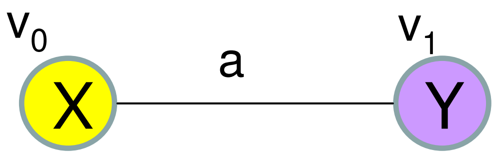
(0,1),(1,2),(2,0),(1,3)



(b)

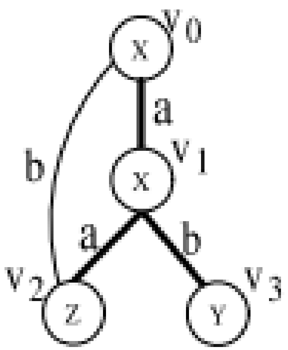
Linear Order and DFS code (2)

- We introduce a new notation to denote DFS codes
  we present an edge by a 5-tuple $(i,j,l_i,l_{(i,j)},l_j)$



(0,1,X,a,Y)

- Example: linear order (0,1),(1,2),(2,0),(1,3)



(b)

edge	γ_0	γ_1	γ_2
e_0	$(0,1,X,a,X)$	$(0,1,X,a,X)$	$(0,1,Y,b,X)$
e_1	$(1,2,X,a,Z)$	$(1,2,X,b,Y)$	$(1,2,X,a,X)$
e_2	$(2,0,Z,b,X)$	$(1,3,X,a,Z)$	$(2,3,X,b,Z)$
e_3	$(1,3,X,b,Y)$	$(3,0,Z,b,X)$	$(3,1,Z,a,X)$

DFS Lexicographic Order
- Minimun DFS code

- We define a minimum DFS code in using the new notation of linear order
- Compare first vertex label l_i, then $l_{(i,j)}$ and at last l_j
- We can extend the order definition in the DFS codes of different graphs (Definition see appendix)
- Let the canonical label to be the smallest DFS code according lexicographic order min(G)
- **Theorem:** two graphs G and H are isomorphic iff min(G)=min(H)
- Mining frequent subgraph is equivalent mining their corresponding minimum DFS code

DFS code tree

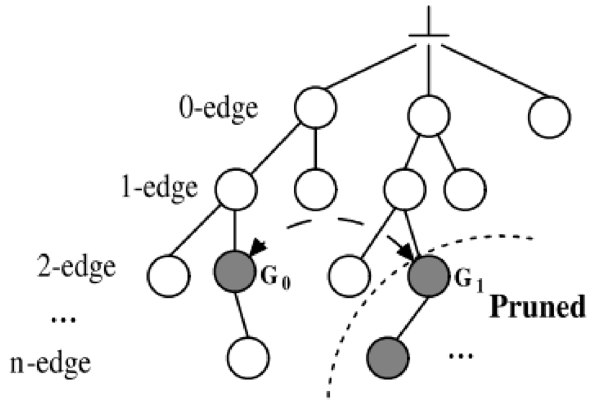
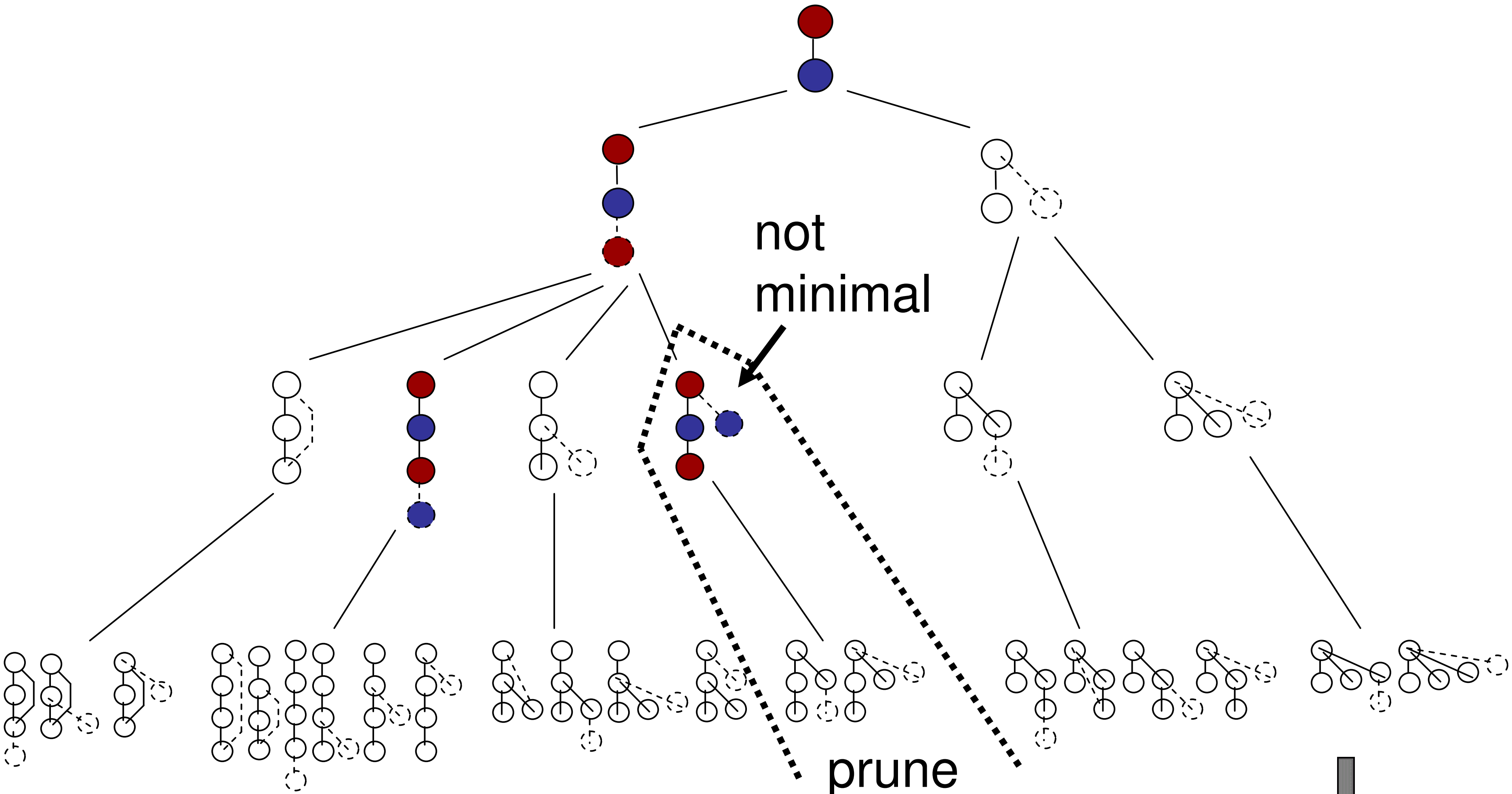


Figure 1: A Search Space: DFS Code Tree

G_0 and G_1 are isomorphic
$G_0 \prec G_1$
We can prune the entire subgraph of G_1

- Every node denotes a possible right-most extension
- **Theorem:** right-most extension guarantees the completeness of mining result
- **Lemma:** performing only the right-most extension on the minimum DFS codes guarantees the completeness of mining result
- If DFS code is not the minimum one, we can prune the entire subtree below this node without destroying the completeness

Example



gSpan algorithm

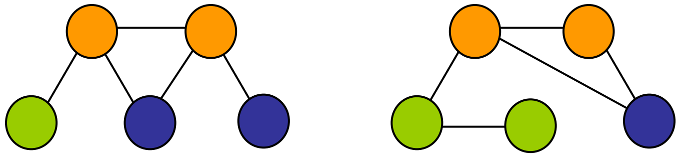
Algorithm gSpanMining(D,MinSup,S)

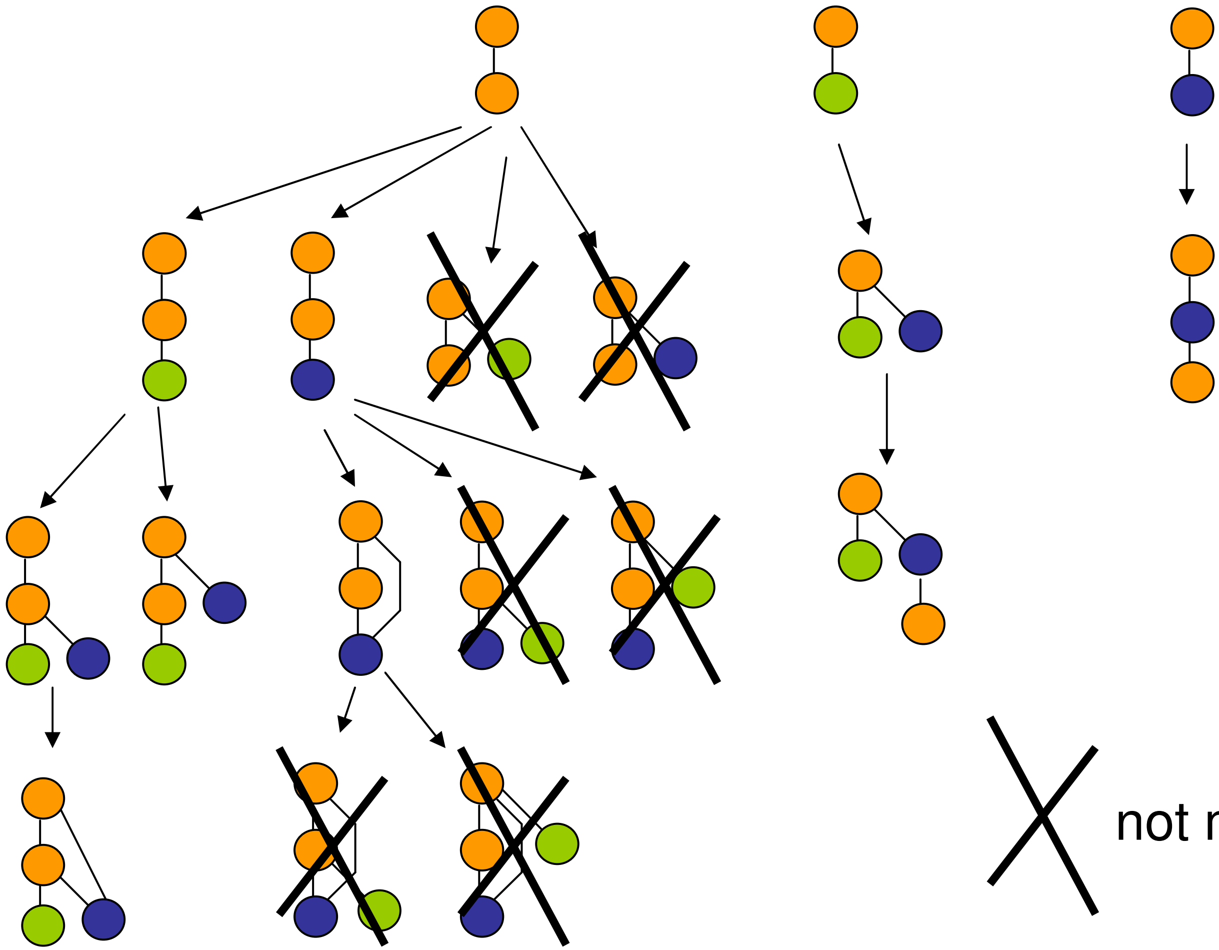
```
1:sort labels of the vertices and edges
   in D by frequency;
2:remove infrequent vertices and
  edges;
3:relabel the remaining vertices and
  edges (descending);
4:S^0←code of all frequent graphs with
  single edge;
5:sort S^0 in DFS lexicographic order;
5:S←S^0;
6:for each code s in S^0 do
7:        gSpan(s,D,MinSup,S);
8:        D:=D-s;
9:        if |D|<MinSup;
10:       break;
```

Algorithm **gSpan(s,D,MinSup,S)**

```
1:if s≠min(s), then
2:        return;
3:insert s into S
4:set C to ∅
5:scan D once, find every edge e
  such that s can be right-most
  extended to frequent s*e;
  insert s*e into C;
6:sort C in DFS lexicographic order;
7:for each s*e in C do
8:        Call gSpan(s*e,D,MinSup,S);
9:return
```

Example





not minimal

gSpan

Pro:

- It reduces the generation of duplicate graphs
- No need to search previous discovered frequent graphs in order to detect duplicates
- No need to extend any duplicate graph but still guarantees the completeness
- Competitve performance compared with other algorithms

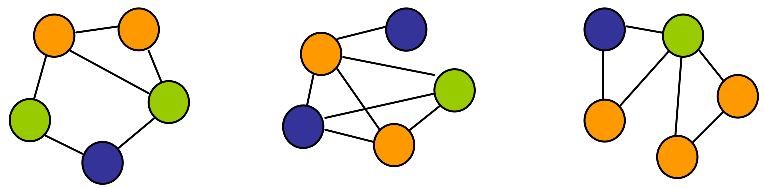
Contra:

- Still inefficient with large size graph
- It is impossible to mine frequent large subgrahs with exponential growth

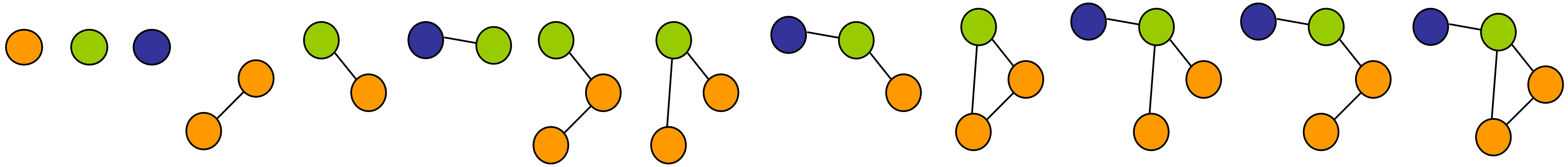
→ New efficient method: *CloseGraph*

Closed Graph

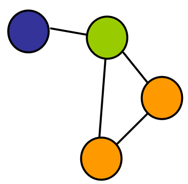
A Graph g is closed in a database if there exists no proper supergraph of g that has the same support as g.



All subgraphs with MinSup 3:



All closed graphs with MinSup 3:

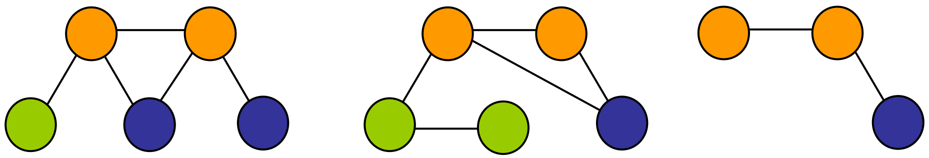


Occurrence and Extended Occurence

- **Occurrence I(g,D)**
  the sum of the number of subgraph isomorphisms of g in every graph of D

- **Extended occurrence $\mathcal{L}$(g,g´,D)**
  the sum of the number of extendable subgraph isomorphism of g as propor subgraph of g´in every graph of D

Equivalent occurence:



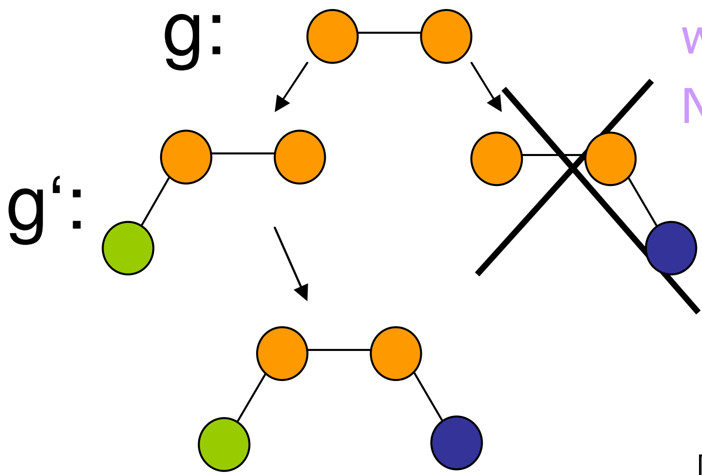
Dataset D

support(g)	2
$\mathcal{I}$(g,D)	1+1 = 2
$\mathcal{L}$(g,g´,D)	1+1 = 2

Equivalent occurence:

I(g,D) = $\mathcal{L}$(g,g´,D)

- graph g occurs just as a subgraph of g´everywhere
- Only g´will be expanded
- Early Termination



always works?

NO

Failure of Early Termination

- Early Termination does not work for every case
- Example:

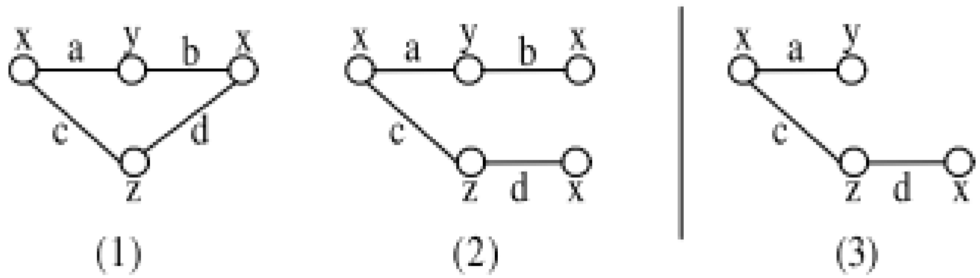


Figure 5: Failure of Early Termination

g = {x,y} and g´= {x,y,x}
g is subgraph of g´and will not be expanded.
we can`t find graph (3) with early termination.

CloseGraph algorithm

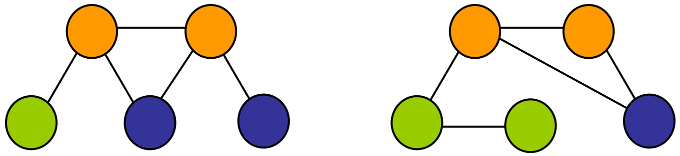
Algorithm CloseMining(D,MinSup,S)

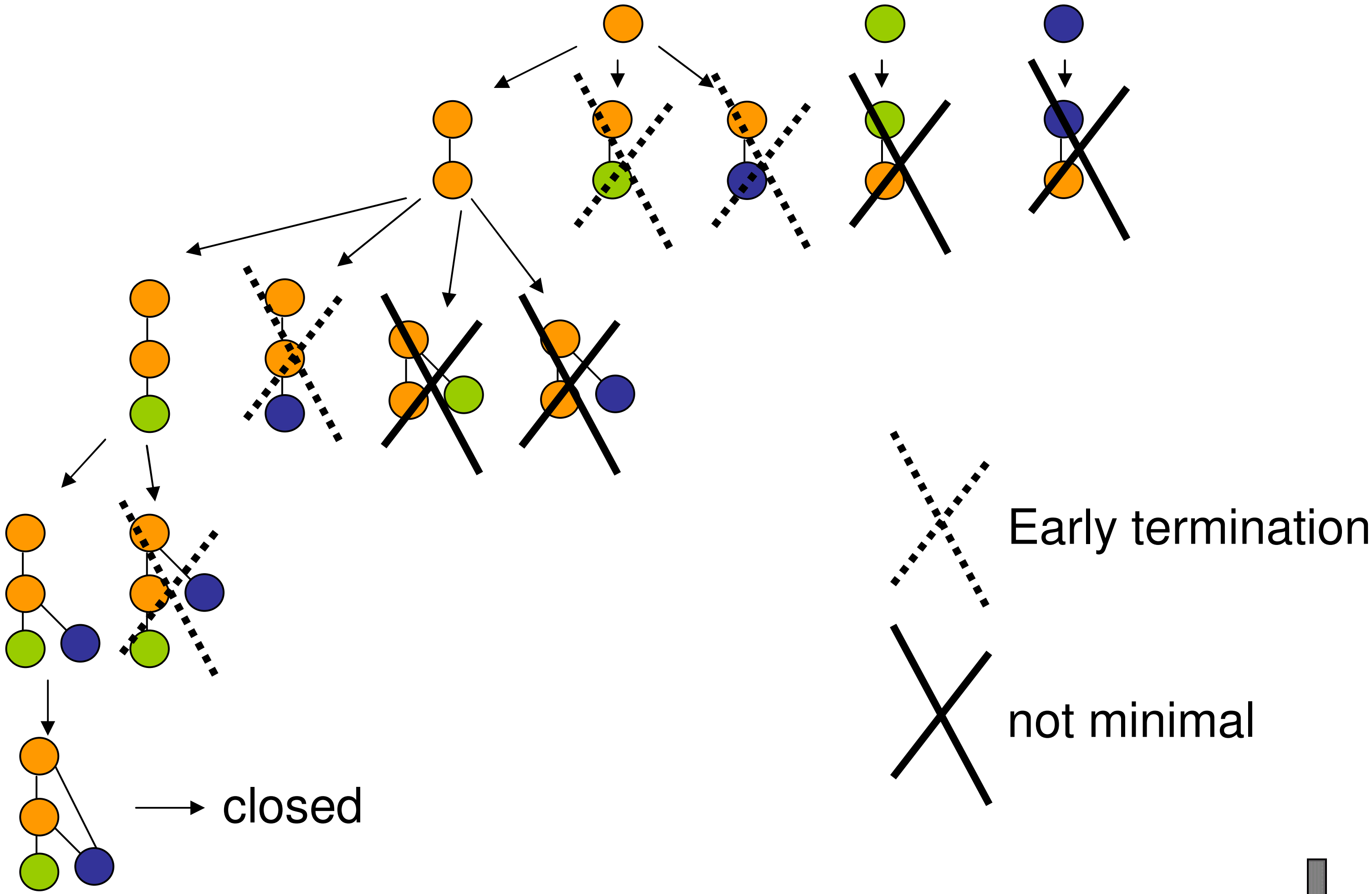
remove infrequent vertices and edges;
$S^0 \leftarrow$ code of frequent graphs with single vertex;
$S \leftarrow S^0$;
for each code s in S^0 do
    CloseGraph(s,D,MinSup,S);

Algorithm **CloseGraph(s,D,MinSup,S)**

if $s \neq min(s)$, then
    return;
if $\exists e'$, $g' = g_p * e'$ and $g' < g_s$ and $I(g_p,D) = \mathcal{L}(g_p,g',D)$ and g_p is not failure case of early termination then
    return;
set C to $\varnothing$
scan D once, find every edge e such that s can be right-most extended to frequent s*e;
insert s*e into C;
delect any possible failure of early termination in s;
if $\nexists s*e \in C$, sup(s) =sup(s*e) then
    insert s into S:
remove s*e from C which cannot be right-most extended from s;
sort C in DFS lexicographic order;
for each s*e in C do
    Call ClosedGraph(s*e,s,D,MinSup,S);
return;

Example





Conclusion

- CloseGraph demonstrated high efficiency over gSpan by large size frequent graph
- A better failure detection algorithm may further improve the performance
- Algorithm is assignable for similar graph problems like directive graph, not associated graph

Literature

[1] Xifeng Yan, Jiawei Han: gSpan: Graph-Based Substructure Pattern Mining. ICDM 2002: 721-724

[2] Xifeng Yan, Jiawei Han: CloseGraph: mining closed frequent graph patterns. KDD 2003: 286-295

[3] other presentations on gSpan and ClosedGraph

- http://www.kbs.uni-hannover.de/Stamm/lehre/praesenzlehre/ki2/gSpan6.ppt
- http://www.informatik.uni-freiburg.de/~ml/teaching/ws04/lm/20041214_CloseGraph_Guetlein.ppt
- http://www.cis.hut.fi/Opinnot/T-61.6020/2008/gspan.pdf

[4] Software package of mining frequent graphs in a graph database http://www.xifengyan.net/software/gSpan.htm

Appendix

- NaiveGraph-Algorithm
- DFS lexicographic order

NaiveGraph-algorithm

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithm 1 NaiveGraph(g, D, min_sup, S)

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**;

DFS Lexicographic Order

DFS Lexicographic Order *is a linear order defined as follows. If* $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ *and* $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n), \alpha, \beta \in Z$*, then* $\alpha \leqslant \beta$ *iff either of the following is true.*

(i) $\exists t, 0 \leqslant t \leqslant \min(m, n), a_k = b_k$ *for* $k < t, a_t < b_t$

(ii) $a_k = b_k$ *for* $0 \leqslant k \leqslant m$*, and* $m \leqslant n$.