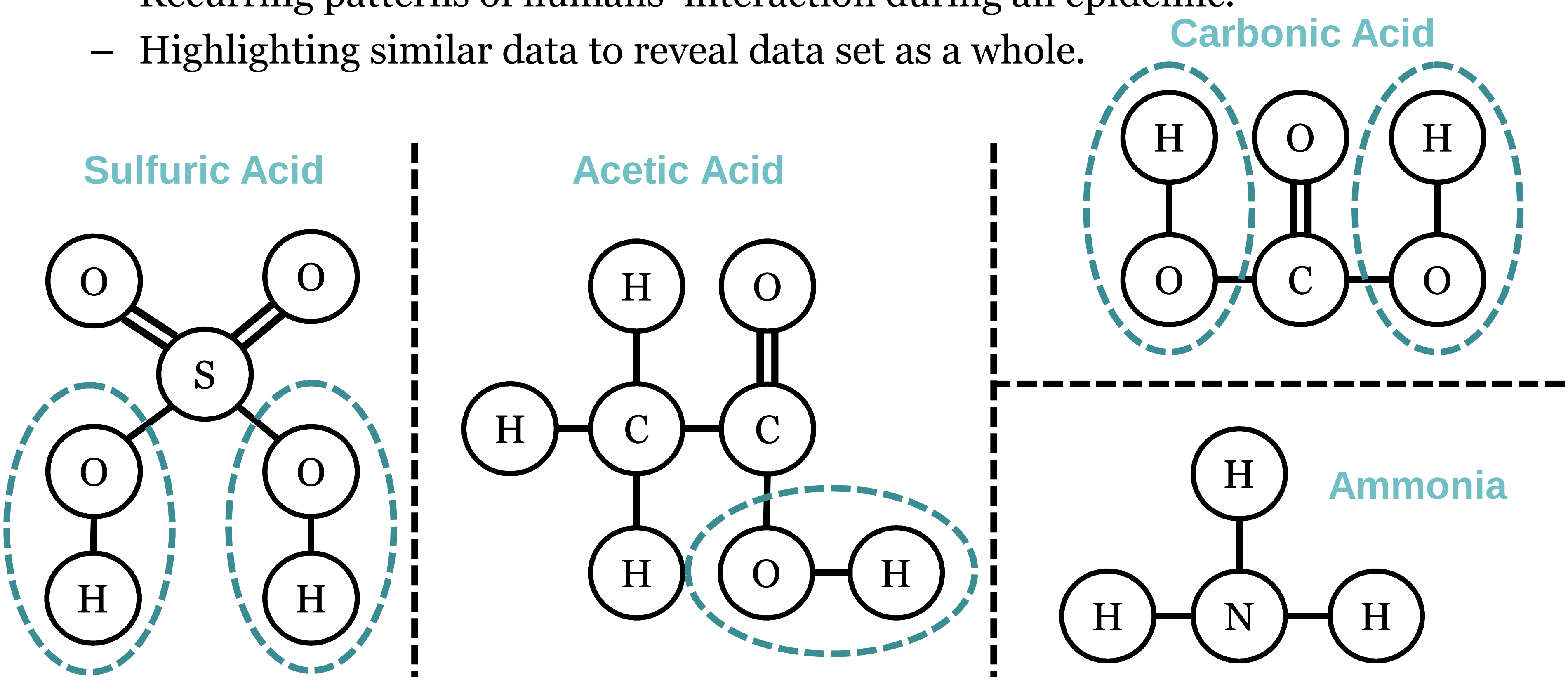
Frequent Subgraph Mining

Frequent Subgraph Mining (FSM) Outline

- **FSM Preliminaries**
- **FSM Algorithms**
  - gSpan – complete FSM on labeled graphs
  - SUBDUE – approximate FSM on labeled graphs
  - SLEUTH – FSM on trees
- **Review**

FSM In a Nutshell

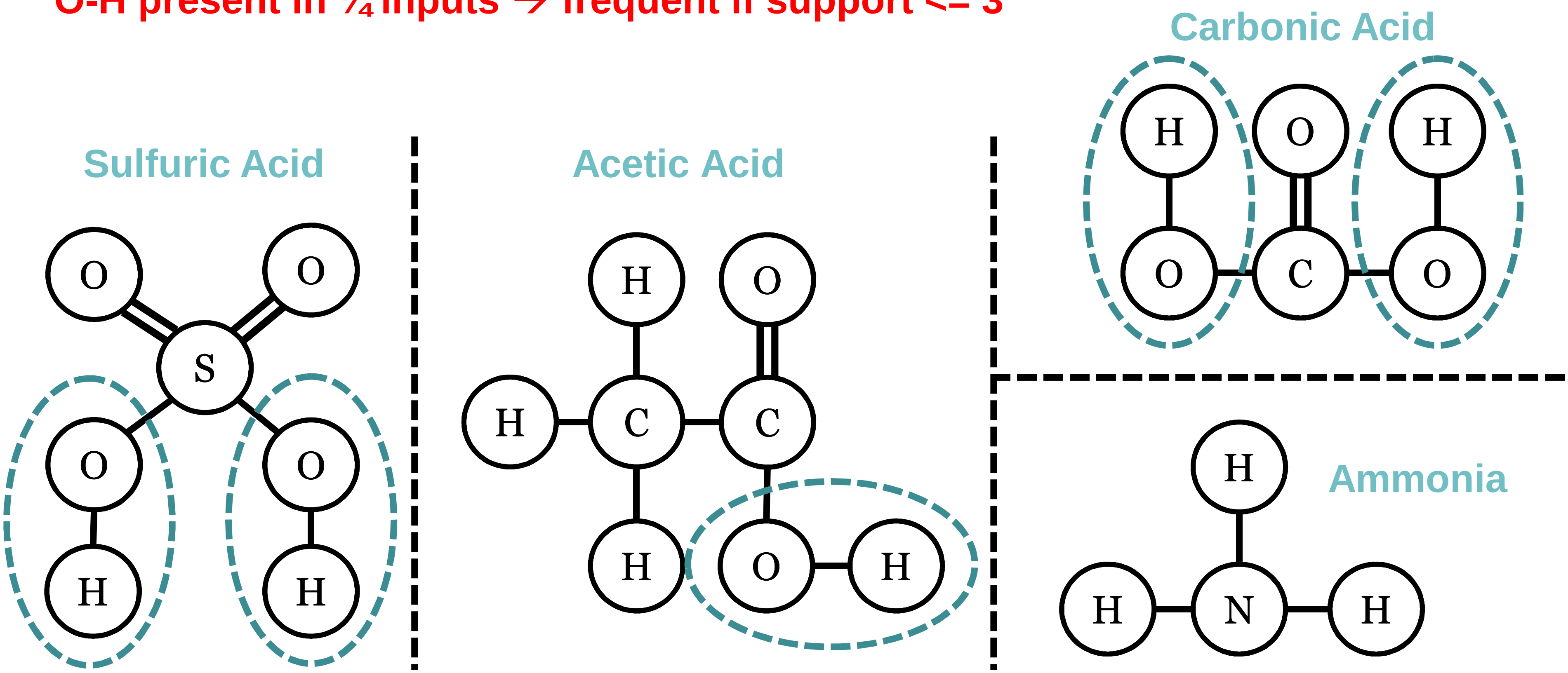
- **Discovery of graph structures that occur a significant number of times across a set of graphs**
- **Ex.: Common occurrences of hydroxide-ion**
- **Other instances:**
  - Finding common biological pathways among species.
  - Recurring patterns of humans interaction during an epidemic.
  - Highlighting similar data to reveal data set as a whole.



FSM Preliminaries

- **Support** is some integer or frequency
- **Frequent** graphs occur more than *support* number of times.

O-H present in ¾ inputs → frequent if support <= 3



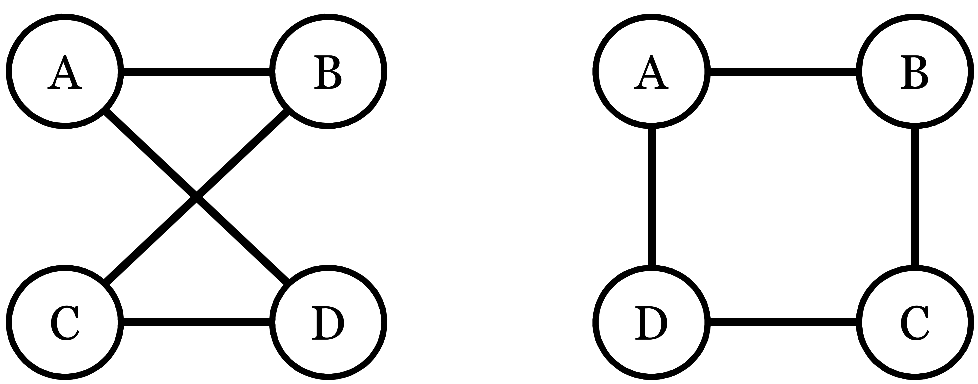
What Makes FSM So Hard?

• **Isomorphic graphs** have same structural properties even though they may look different.

• **Subgraph isomorphism problem**: Does a graph contain a subgraph isomorphic to another graph?

• FSM algorithms encounter this problem while buildings graphs.

• This problem is known to be **NP-complete**!



Isomorphic under A,B,C,D labeling

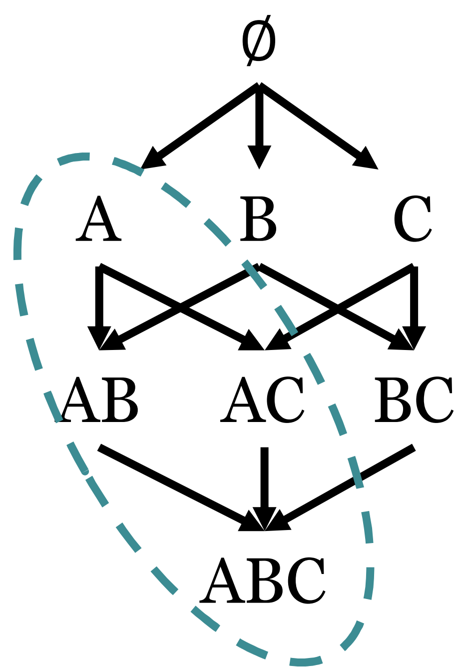
Pattern Growth Approach

- **Underlying strategy of both traditional frequent pattern mining and frequent subgraph mining**
- **General Process:**
  - *candidate generation*: which patterns will be considered? For FSM,
  - *candidate pruning*: if a candidate is not a viable frequent pattern, can we exploit the pattern to prevent unnecessary work?
    - subgraphs and subsets exponentiate as size increases!
  - *support counting*: how many of a given pattern exist?
- **These algorithms work in a breadth-first or depth-first way.**
  - Joins smaller frequent sets into larger ones.
  - Checks the frequency of larger sets.

Pattern Growth Approach – Apriori

- ***Apriori principle:*** if an itemset is frequent, then all of its subsets are also frequent.
  - Ex. if itemset {A, B, C, D} is frequent, then {A, B} is frequent.
  - Simple proof: With respect to frequency, all sets trivially contain their subsets, thus frequency of subset >= frequency of set.
  - Same property applies to (sub)graphs!
- **Apriori algorithm exploits this to prune huge sections of the search space!**

If A is infrequent, no supersets with A can be frequent!



FSM Algorithms Discussed

- **gSpan**
  - *complete* frequent subgraph mining
  - improves performance over straightforward apriori extensions to graphs through *DFS Code* representation and aggressive candidate pruning
- **SUBDUE**
  - *approximate* frequent subgraph mining
  - uses graph compression as metric for determining a "frequently occuring" subgraph
- **SLEUTH**
  - *complete* frequent subgraph mining
  - built specifically for trees

FSM – R package

- **R package for FSM is called** subgraphMining
- **To import:** install.packages(“subgraphMining”)
- **Package contains: gSpan, SUBDUE, SLUETH.**
- **Also contains the following data sets:**
  - cslogs
  - metabolicInteractions.
- **To load the data, use the following code:**

```
# The cslogs data set
data(cslogs)
# The matabolicInteractions data
data(metabolicInteractions)
```

FSM Outline

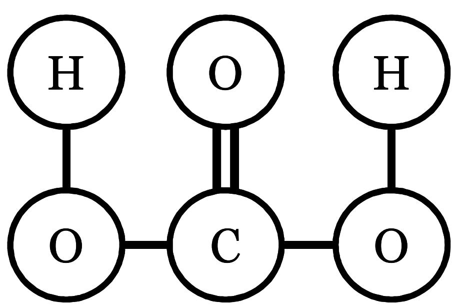
- **FSM Preliminaries**
- **FSM Algorithms**
  - gSpan
  - SUBDUE
  - SLEUTH
- **Review**

gSpan: Graph-Based Substructure Pattern Mining

- **Written by Xifeng Yan & Jiawei Han in 2002.**
- **Form of pattern-growth mining algorithm.**
  - Adds edges to candidate subgraph
  - Also known as, edge extension
- **Avoid cost intensive problems like**
  - Redundant candidate generation
  - Isomorphism testing
- **Uses two main concepts to find frequent subgraphs**
  - DFS lexicographic order
  - minimum DFS code

gSpan Inputs

- **Set of graphs, support**
- **Graph of form** $G = (V, E, L_V, L_E)$
  - V, E – vertex and edge sets
  - L_V – vertex labels
  - L_E – edge labels
  - label sets need not be one-to-one



$$L_V = \{ H, O, C \}$$

$$L_E = \{ \text{single-bond}, \text{double-bond} \}$$

gSpan Components

Strategy:

- build frequent subgraphs *bottom-up*, using DFS code as regularized representation
- eliminate redundancies via minimal DFS codes based on code lexicographic ordering

Depth-first Search (DFS) Code

structured
graph representation
for building, comparing

DFS Lexicographic Order → *minimal* DFS code

canonical comparison of graphs

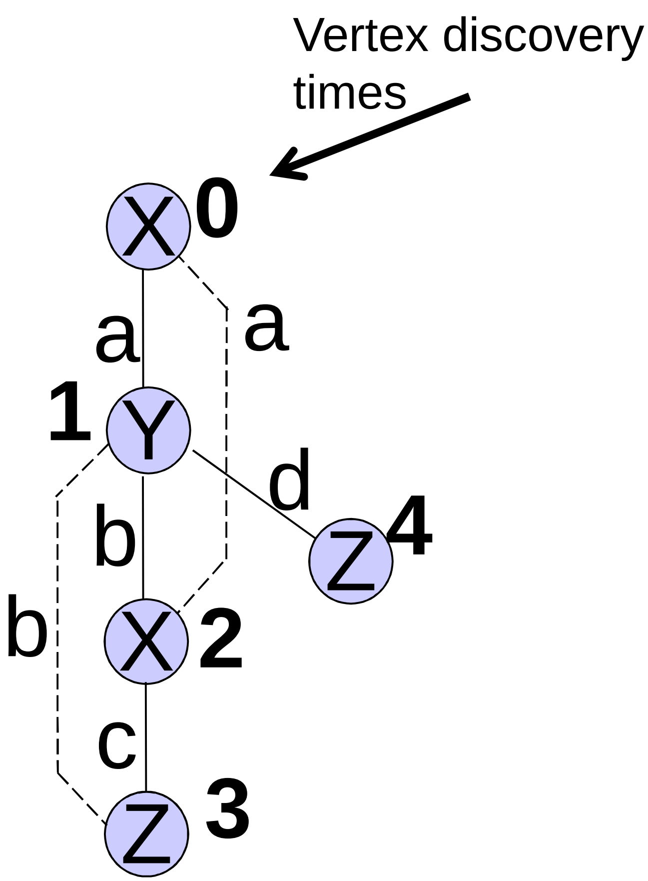
selection, *pruning* of subgraphs

Depth First Search Primer

Todo…?

gSpan: DFS codes

DFS Code: sequence of edges traversed during DFS



Edge #	Code
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)

Format: $(i, j, L_i, L_{(i,j)}, L_j)$

i, j – **vertices by time of discovery**

L_i, L_j **- vertex labels of** v_i, v_j

$L_{(i,j)}$ – **edge label between** v_i, v_j

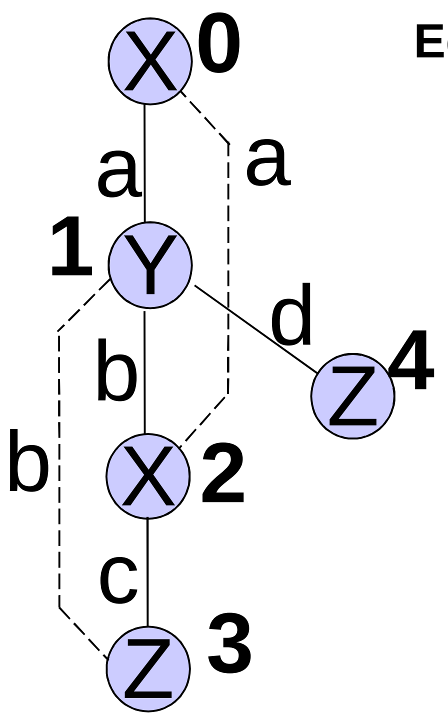
$i < j$ **: forward edge**

$i > j$ **: back edge**

DFS Code: Edge Ordering

- **Edges in code ordered in very specific manner, corresponding to DFS process**
- $e_1 = (i_1, j_1), e_2 = (i_2, j_2)$
- $e_1 \prec e_2 \rightarrow e_1$ **appears before** e_2 **in code**
- **Ordering rules:**
  1. if $i_1 = i_2$ and $j_1 < j_2 \rightarrow e_1 \prec e_2$
     - from same source vertex, e_1 traversed before e_2 in DFS
  2. if $i_1 < j_1$ and $j_1 = i_2 \rightarrow e_1 \prec e_2$
     - e_1 is a forward edge and e_2 traversed as result of e_1 traversal
  3. if $e_1 \prec e_2$ and $e_2 \prec e_3, \rightarrow e_1 \prec e_3$
     - ordering is transitive

DFS Code: Edge Ordering Example



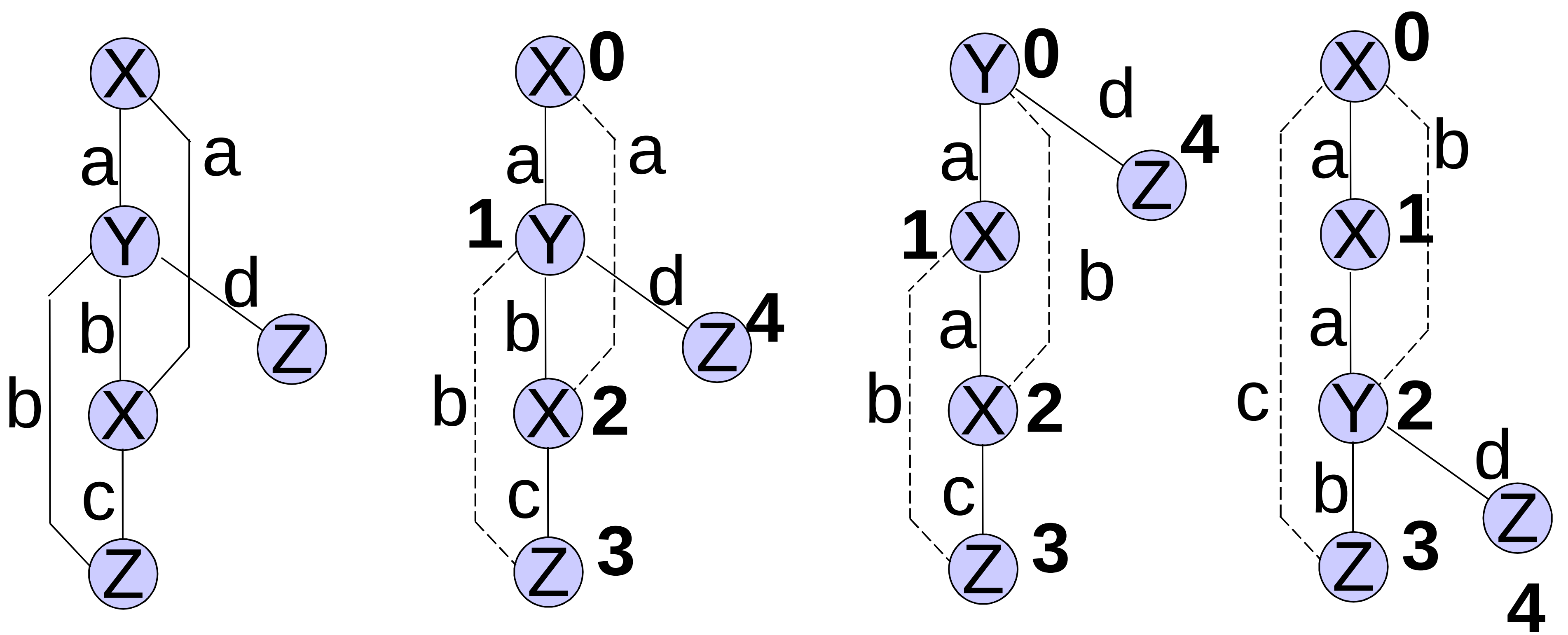
Edge #	Code
0	(0,1,X,a,Y)
1	(1,2,Y,b,X)
2	(2,0,X,a,X)
3	(2,3,X,c,Z)
4	(3,1,Z,b,Y)
5	(1,4,Y,d,Z)

- **Rule applications by edge #**
- $0 \prec 1$ **(Rule 2)**
- $1 \prec 2$ **(Rule 2)**
- $0 \prec 2$ **(Rule 3)**
- $2 \prec 3$ **(Rule 1)**
- **Exercise: what others?**

Edge ordering can be recorded easily *during* the DFS!

Graphs have multiple DFS Codes!

Exercise: Write the 2 rightmost graphs using DFS code



solution to redundant DFS codes: lexical ordering, minimal code!

DFS Lexicographic Ordering vs. DFS Code

- **DFS code: Ordering of edge sequence of a particular DFS**
  - E.g. DFS's that start at different vertices may have different DFS codes
- **Lexicographic ordering: ordering between different DFS codes**

DFS Lexicographic Ordering

- **Given lexicographic ordering of label set L, $\prec_L$**
- **Given graphs G_α, G_β (equivalent label sets).**
- **Given DFS codes**
  - $\alpha = \text{code}(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$
  - $\beta = \text{code}(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$
  - (assume $n \geq m$)
- $\boldsymbol{\alpha \leq \beta}$ **iff either of the following are true:**
  - $\exists t, 0 \leq t \leq \min(n, m)$ such that
    - $a_k = b_k$ for $k < t$ and
    - $a_t \prec_e b_t$
  - $a_k = b_k \; for \; 0 \leq k \leq m$

DFS Lex. Ordering: Edge Comparison

- **Given DFS codes**
  - $\alpha = \text{code}(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$
  - $\beta = \text{code}(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$
  - (assume $n \geq m$)
- **Given t such that** $a_k = b_k$ for $k < t$
- **Given** $\boldsymbol{a_t = (i_a, j_a, L_{i_a}, L_{i_a,j_a}, L_{j_a})}$,
  $\boldsymbol{b_t = (i_b, j_b, L_{i_b}, L_{i_b,j_b}, L_{j_b})}$,
- $\boldsymbol{a_t \prec_e b_t}$ **if one of the following cases**

Case 1:
Both forward edges, AND…

Case 2:
Both back edges, AND…

Case 3: a_t back, b_t forward → $a_t \prec_e b_t$

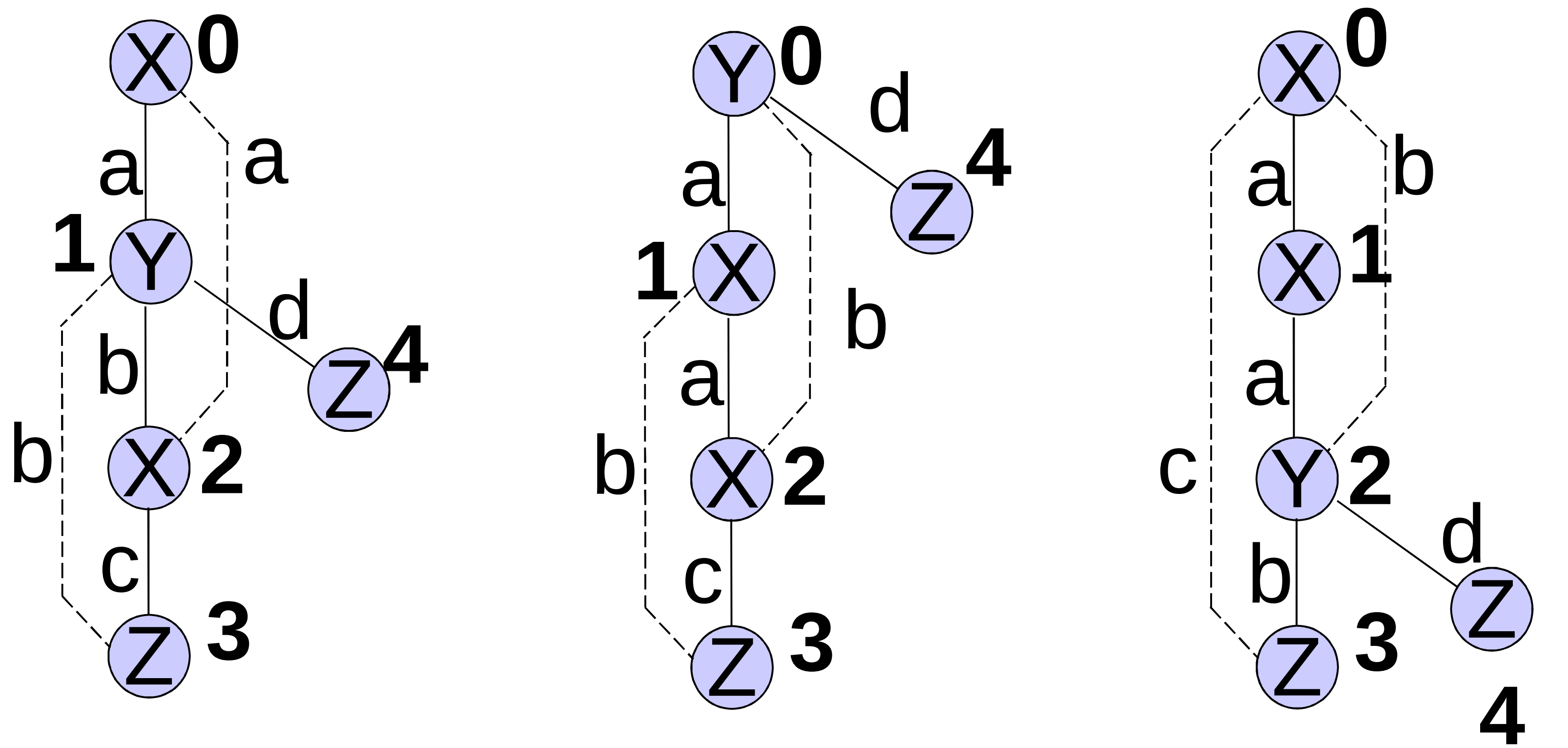
Edge Comparison: Case 1 (both forward)

- **Both forward edges, AND one of the following:**
  - $i_b < i_a$ (edge starts from a ***later*** visited vertex)
    - Why is this (think about DFS process)?
  - $i_a = i_b$ AND labels of a lexicographically less than labels of b, in order of tuple.
    - Ex: Labels are strings, $a_t = (\_, \_, \text{m}, \text{e}, \text{x}), b_t = (\_, \_, \text{m}, \text{u}, \text{x})$
      - m = m, e < u → $a_t \prec_e b_t$
- **Note: if both forward edges, then $j_a = j_b$**
  - Reasoning: all previous edges equal, target vertex discovery times are the same

Edge Comparison: Case 2 (both back)

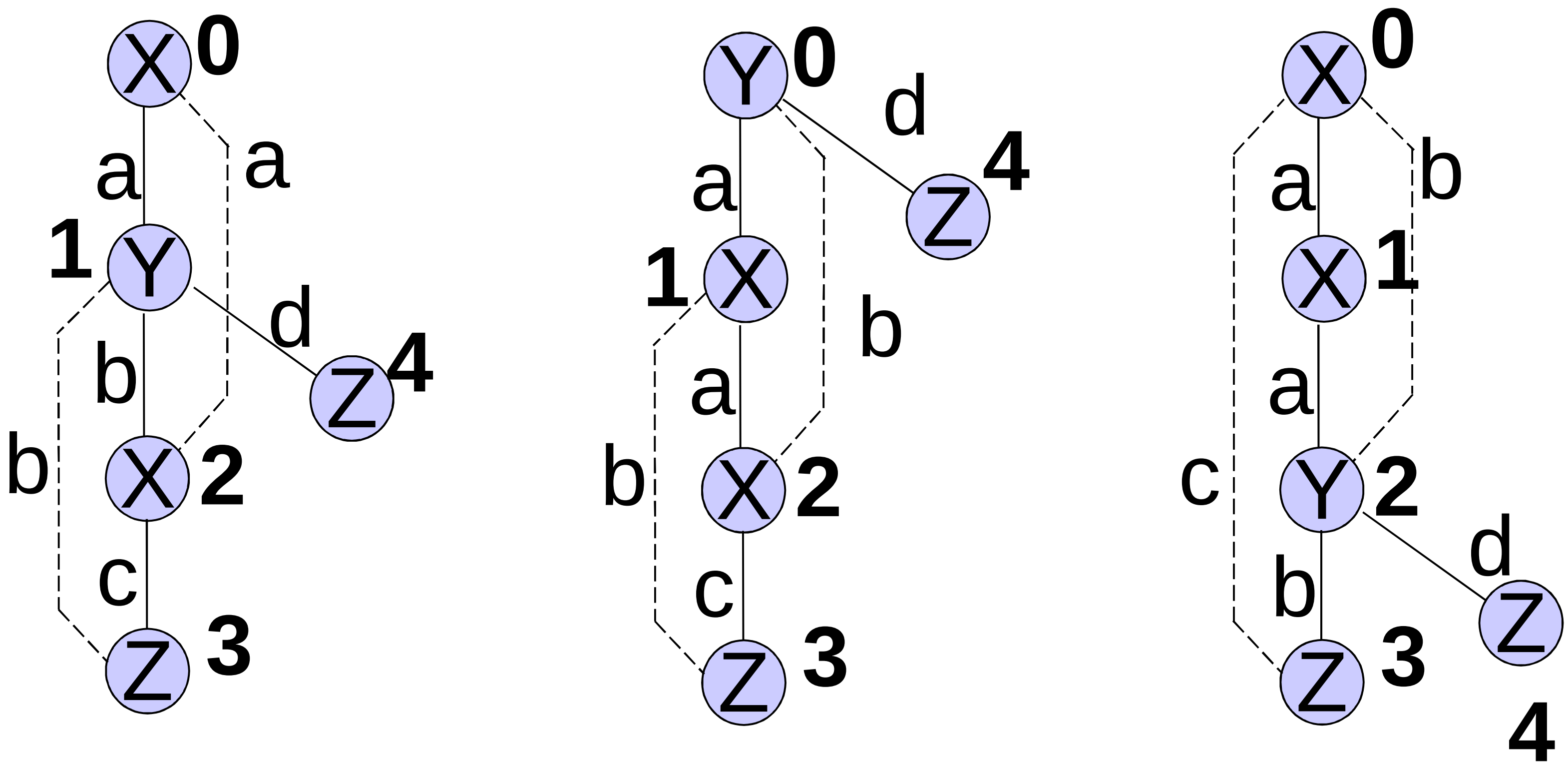
- **Both back edges, AND one of the following:**
  - $j_a < j_b$ (edge refers to earlier vertex)
  - $j_a = j_b$ AND edge label of a lexicographically less than b
    - Note: given that all previous edges equal, vertex labels must also be equal
- **Note: if both back edges, then $\boldsymbol{i_a = i_b}$**
  - Reasoning: all previous edges equal, source vertex discovery times are the same.

Edge #	Code (A)	Code (B)	Code (C)
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)	(3,1,Z,b,X)	(3,0,Z,c,X)
5	(1,4,Y,d,Z)	(0,4,Y,d,Z)	(2,4,Y,d,Z)



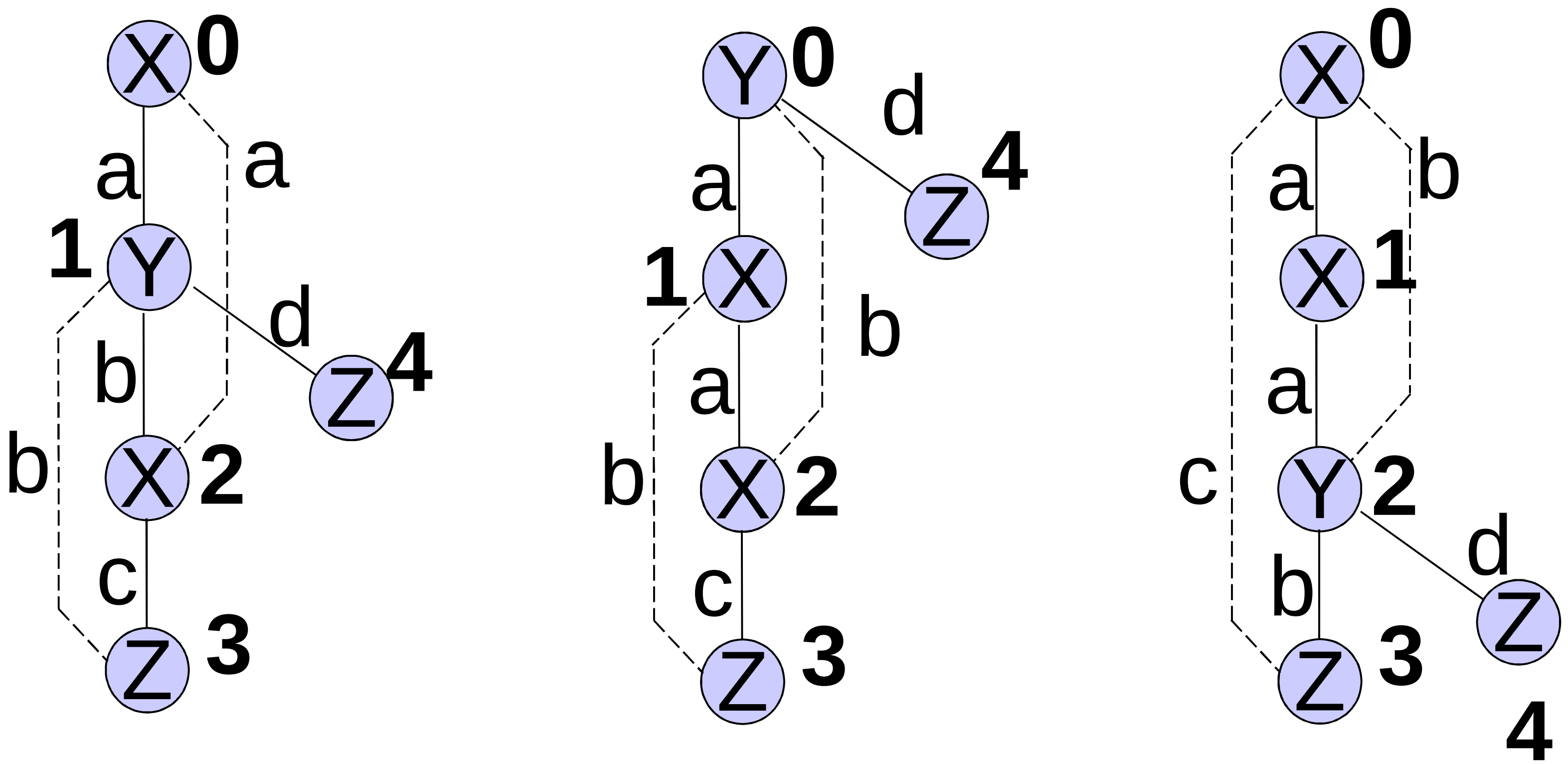
$\prec_L = \{X < Y < Z : a < b < c\}$ C < A < B

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



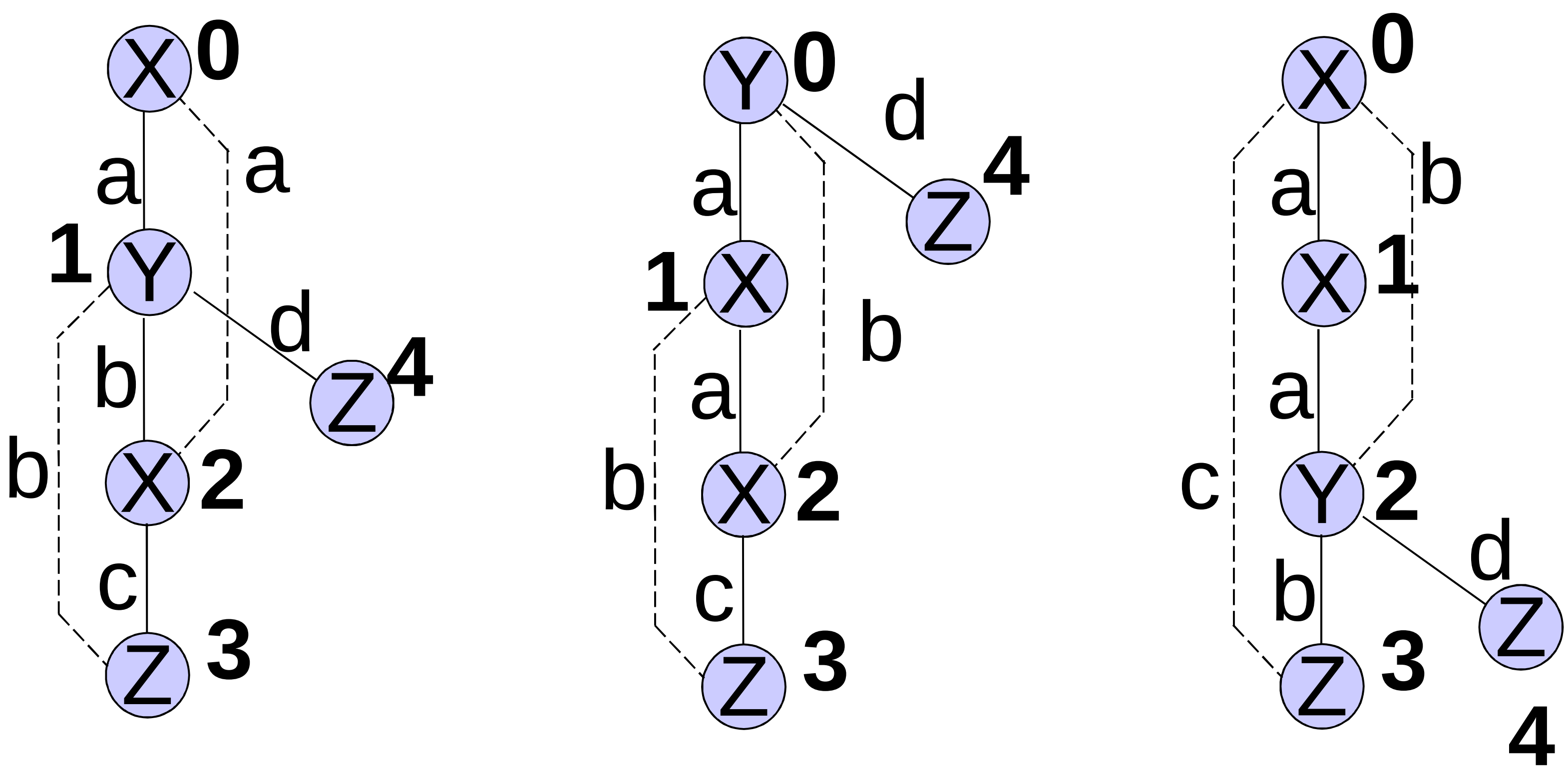
$\prec_L = \{X = Y = Z : b < c < a\}$ A < C < B

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



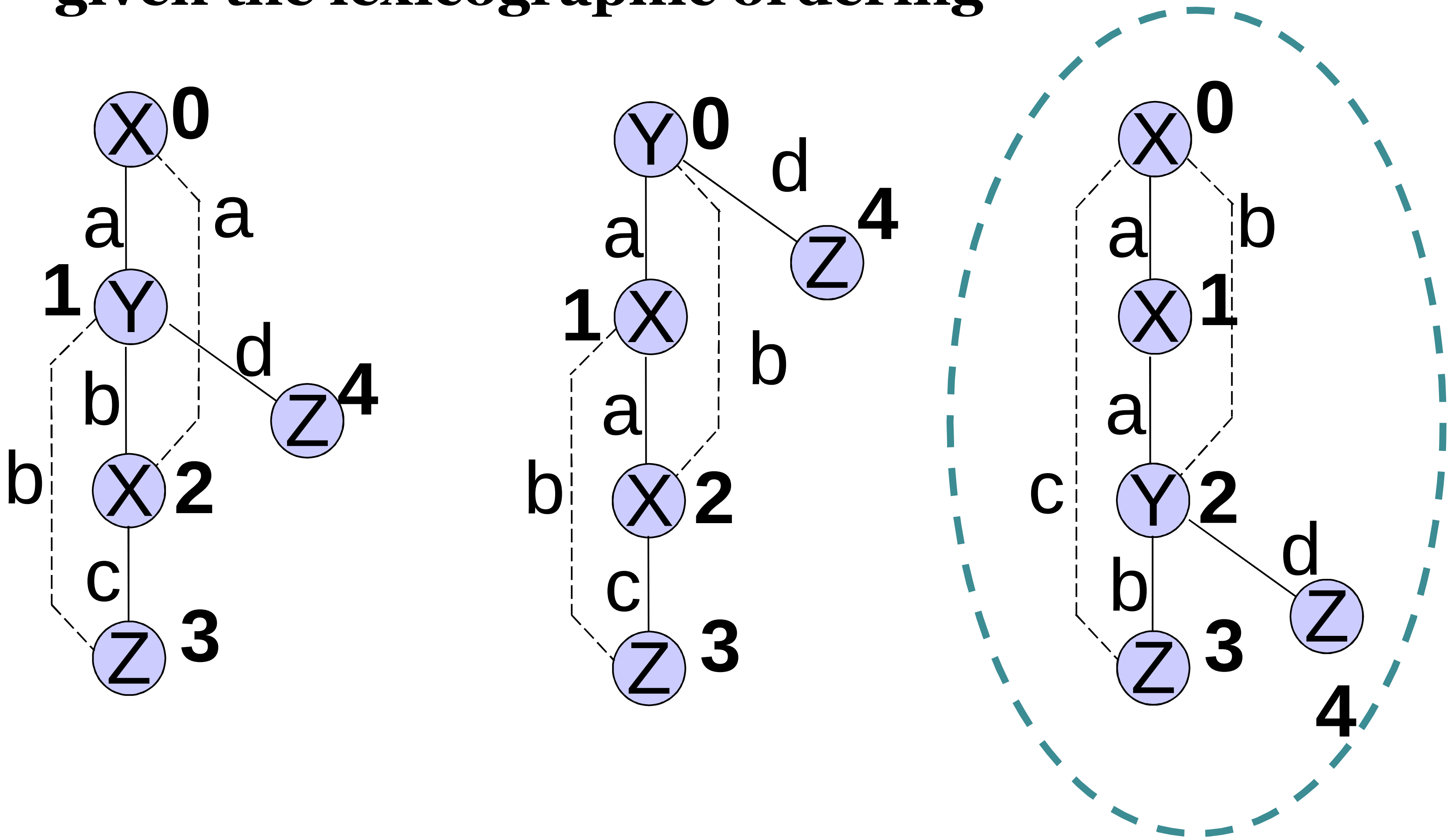
$\prec_L = \{X = Y = Z : b = c = a\}$ C < A < B

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



Minimal DFS code

- **Merely the "minimum" of all possible DFS codes, given the lexicographic ordering**



Minimal for $\prec_L = \{X = Y = Z : b = c = a\}$

DFS Code Building

- Given code $\alpha = (a_0, a_1, \dots , a_m)$ and $\beta = (a_0, a_1, \dots , a_m, b)$
- β is α's **child**
- α is β's **parent**

(0,1,X,a,Y)
(1,2,Y,b,X)

→

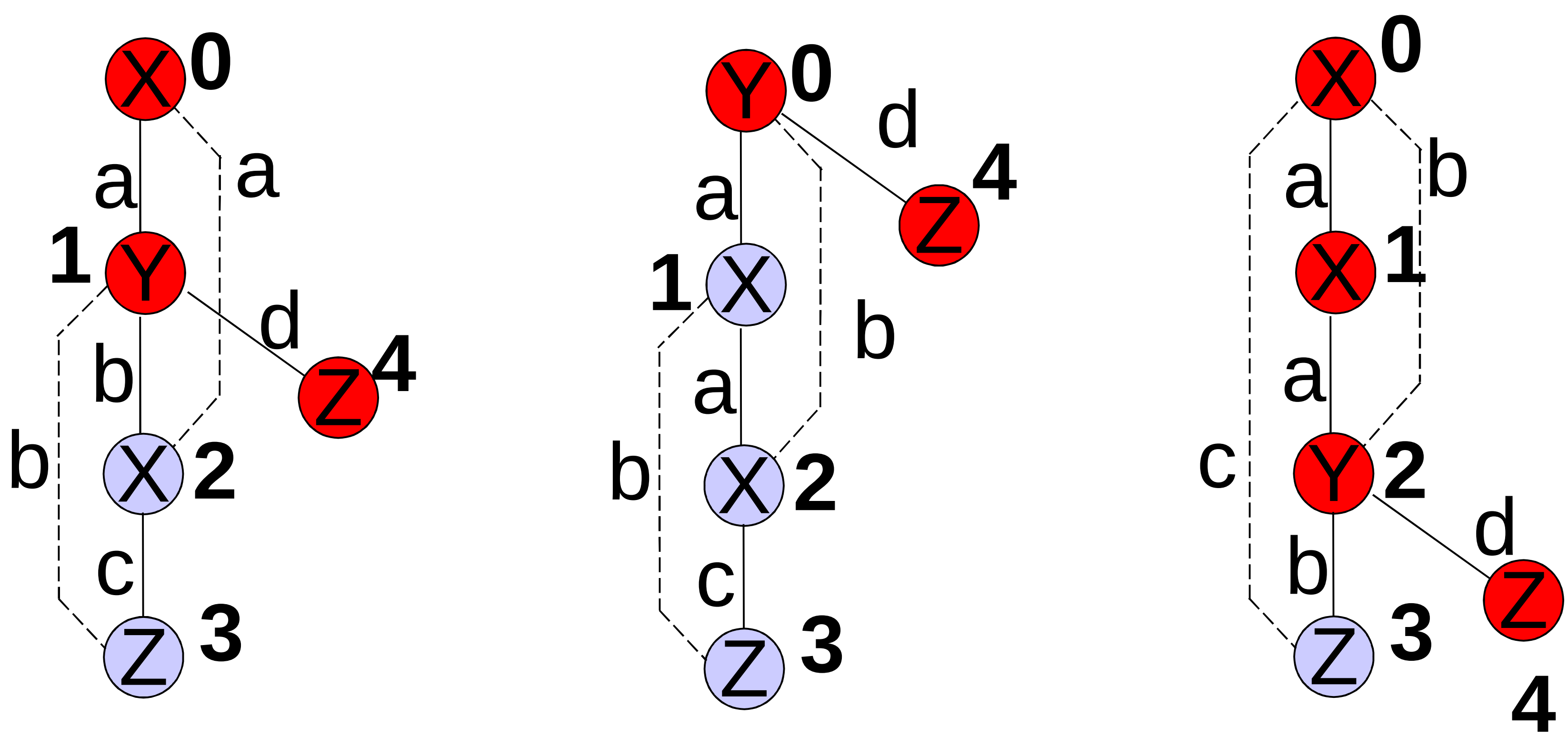
(0,1,X,a,Y)
(1,2,Y,b,X)
(2,0,X,a,X)

→

(0,1,X,a,Y)
(1,2,Y,b,X)
(2,0,X,a,X)
(2,3,X,c,Z)

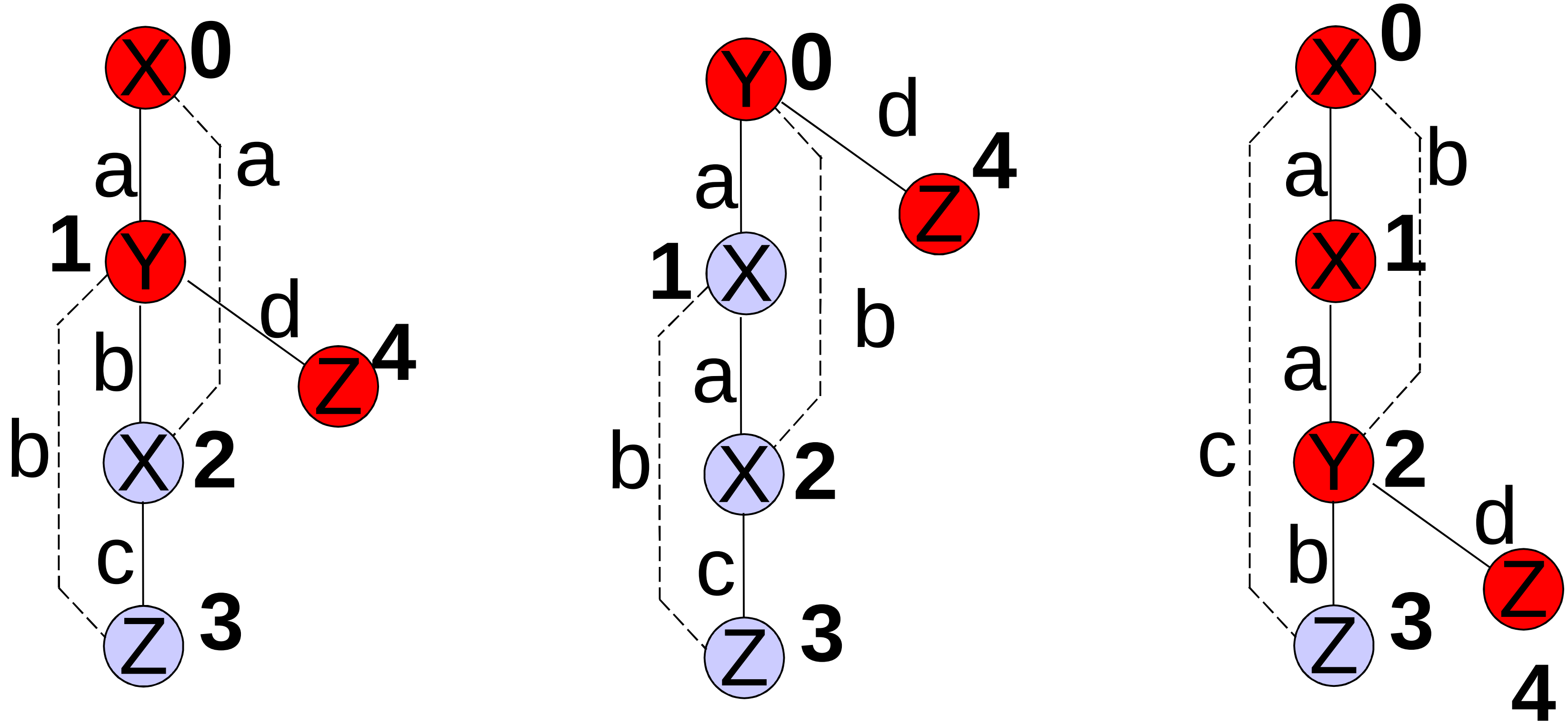
DFS Code Building Basis: Rightmost Path

- **Label vertices by visit order:** $(v_0, v_1, \ldots, v_n)$
  - v_0: first visited, v_n: last visited
  - v_n called the “rightmost” vertex (think of DFS visiting vertices left-to-right in adjacency list)
- **Rightmost path: shortest path between v_0 and v_n using forward edges (examples shown in red)**



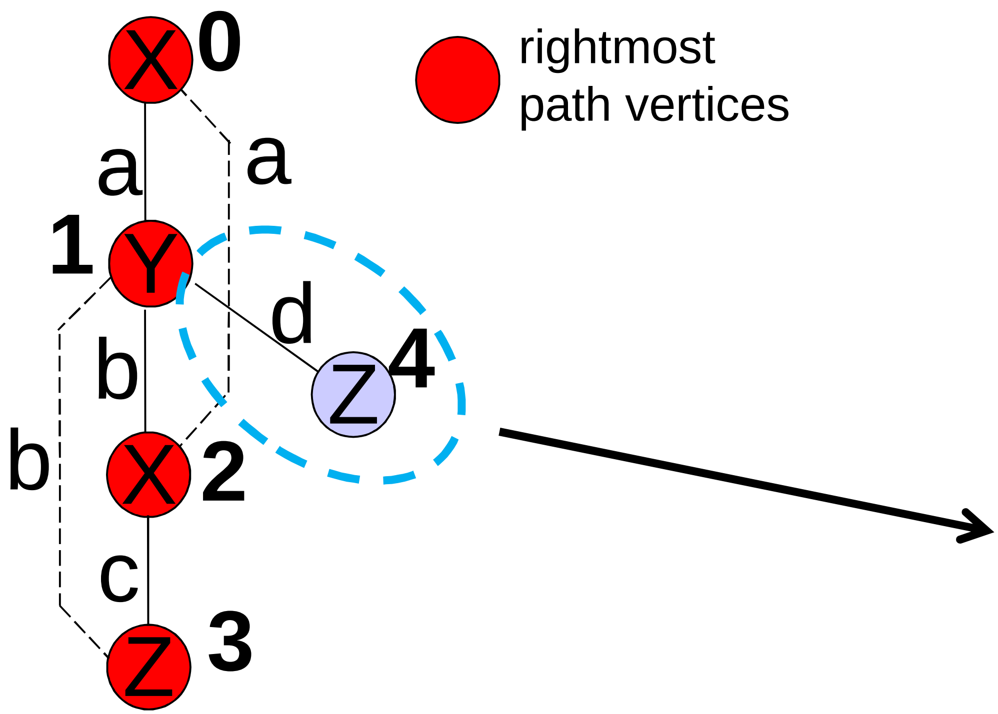
DFS Code Building Basis: Rightmost Path

- **Label vertices by visit order:** $(v_0, v_1, \ldots, v_n)$
  - v_0: first visited, v_n: last visited
  - v_n called the "rightmost" vertex (think of DFS visiting vertices left-to-right in adjacency list)
- **Rightmost path: shortest path between v_0 and v_n using forward edges (examples shown in red)**



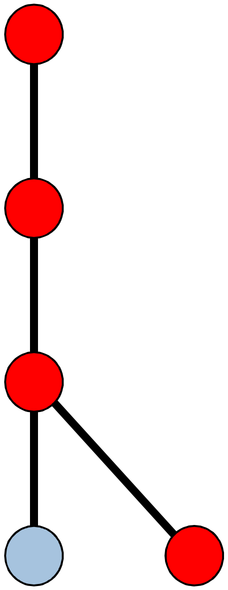
DFS Code Building Basis: Rightmost Path

- **Key:** Forward edge extensions to a DFS code must occur from a vertex on the rightmost path!
- **Key 2:** Back edge extensions must occur from the rightmost vertex!
- **Proof points:**
  - if vertex not on rightmost path, then it has been fully processed by DFS.
  - previous last DFS edge tuple $\prec$ new tuple, if
    - new edge is forward, extended from a vertex on rightmost path, OR
    - new edge is backward, extended from rightmost vertex



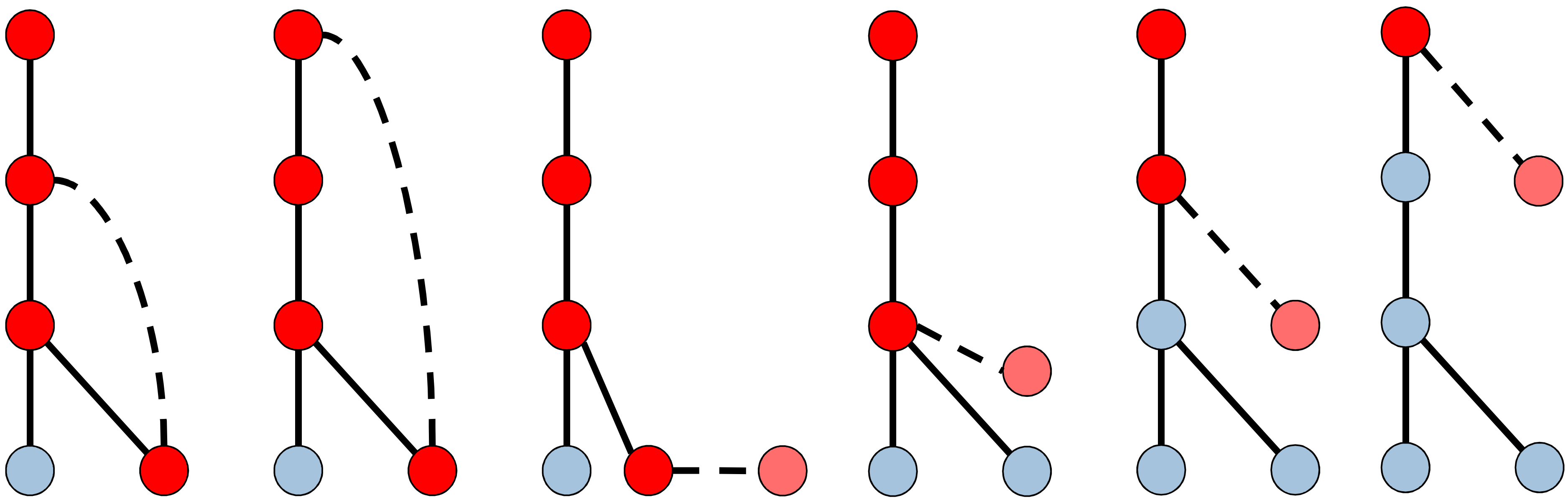
0	**(0,1,X,a,Y)**
1	**(1,2,Y,b,X)**
2	**(2,0,X,a,X)**
3	**(2,3,X,c,Z)**
4	**(3,1,Z,b,Y)**
5	**(1,4,Y,d,Z)**

DFS Code Building Example



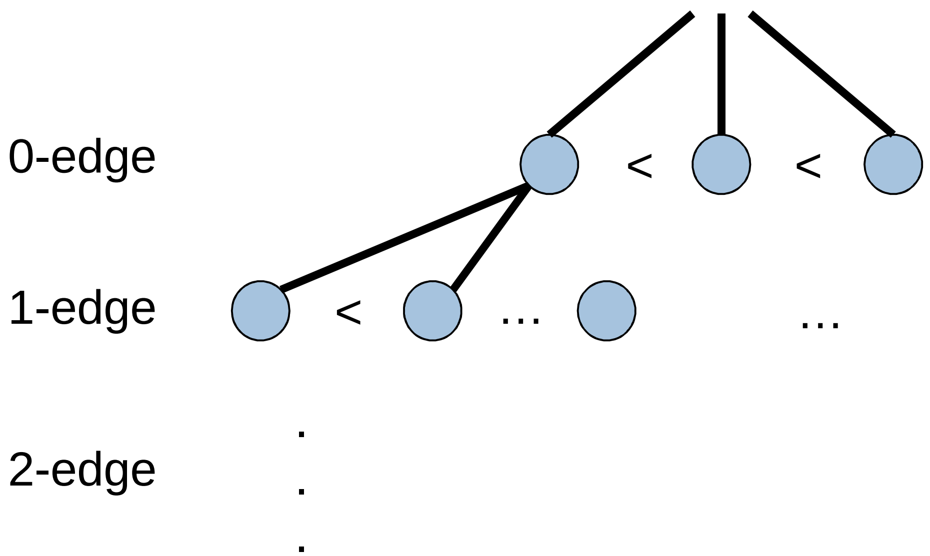
Rightmost path vertex

When building DFS codes, must expand all back edges first!



DFS Code Tree

- **Given vertex label set and edge label set, DFS Code Tree is tree of all possible DFS codes**
  - nodes of tree are DFS codes, except…
    - first level of tree is a vertex for each vertex label
  - each level of the tree adds an edge to the DFS code
  - each parent/child pair follows DFS Code building rules
  - siblings follow DFS lexicographic order



exercise: given 3 vertex labels and 3 edge labels:

- number of nodes in first level?
- branching factor of parents in first level?
- second level?
- third level?
- …

gSpan Algorithm

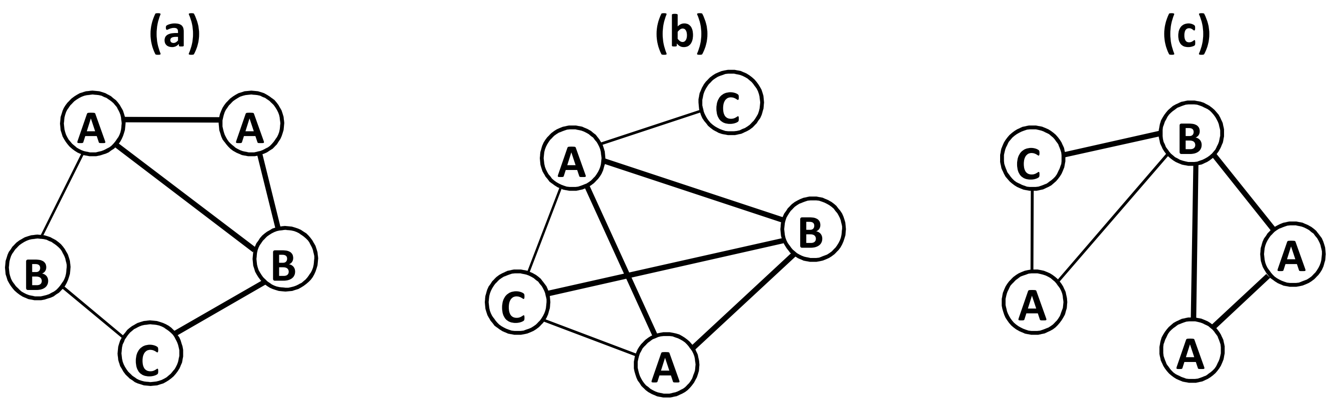
- **Traverse DFS code tree for given label sets**
  - prune using support, minimality of codes.
- **Input: Graph database** D, min_support
- **Output: frequent subgraph set** S
- **General process:**
  - $S \leftarrow$ all frequent one-edge subgraphs in D (using DFS code)
  - Sort S in lexicographic order
  - $N \leftarrow S$ (S gets modified)
  - foreach $n \in N$ do:
    - gSpan_extend ($D, n,$ min_support, S)
  - remove n from all graphs in D (only consider subgraphs not already enumerated)
- **Strategy: grow minimal DFS codes that occur frequently in** D

gSpan Algorithm

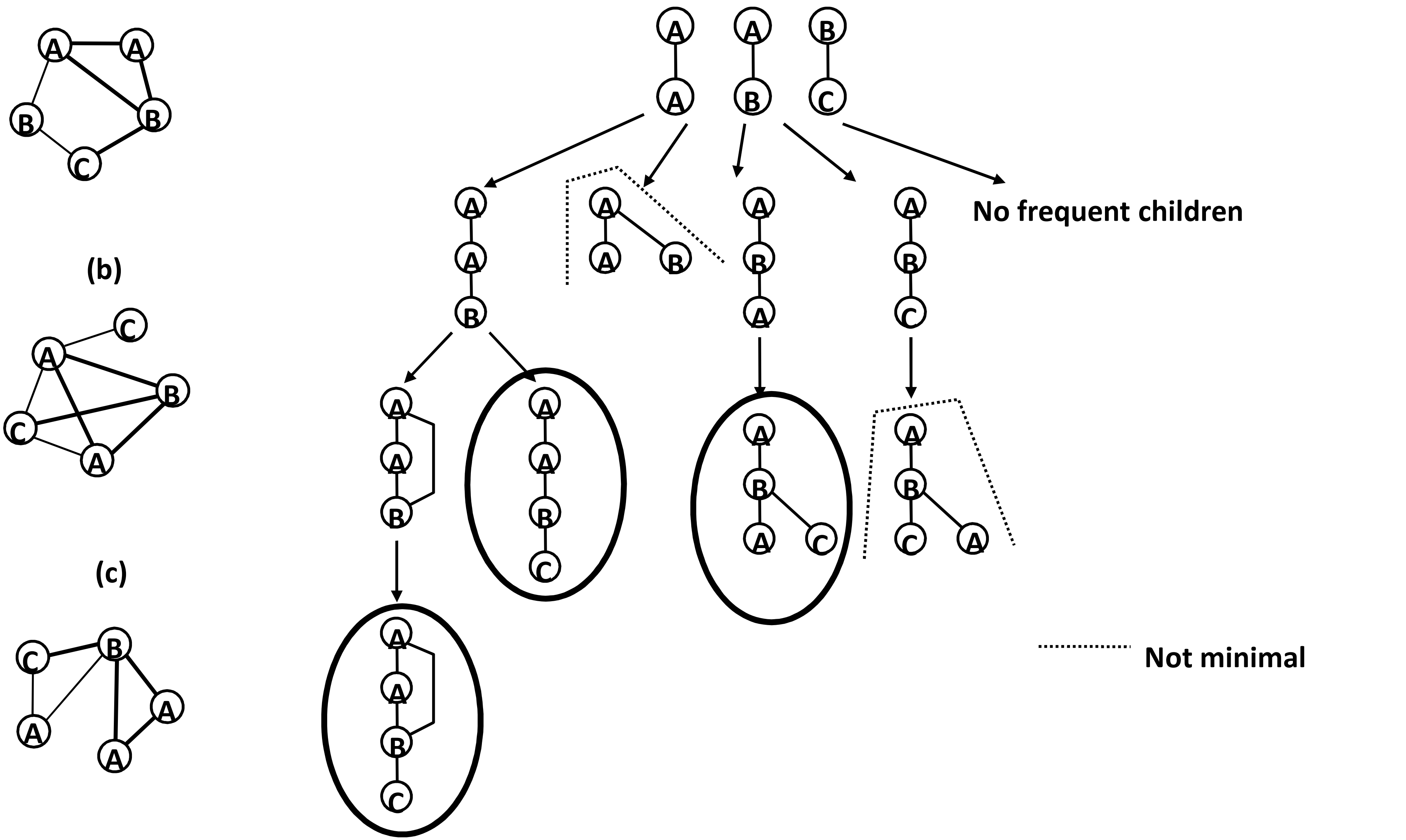
- **gSpan_extend : perform DFS growing and pruning**
- **Input: Graph database** D, min_support, **DFS code** n
- **Input/Output: frequent subgraph set** S
- **Pseudocode:**
  - if n not minimal then end
  - otherwise
    - add n to S
    - foreach single-edge rightmost expansion of n (e)
      - if $support(e)$ >= $\min\_support$
      - recurse using $D, e,$ min_support, S

gSpan Algorithm Example

Inputs: (min_support = 3)



(a) min support = 3



gSpan in R

- **To run gSpan in R, you need the** subgraphMining **package installed. (Written in Java)**
- **Load the** iGraph **R package because it uses iGraph objects.**

```
#Import the subgraphMining package
> library(subgraphMining)
# Create a database of graphs.
# The database should be an R array of
# iGraph objects put into list form.
# freq is an integer percent. The
# frequency should be given as a string.
# Here is an example database of
# two ring graphs
graph1 = graph.ring(5);
graph2 = graph.ring(6);
database = array(dim = 2);
database[1] = list(graph1);
database[2] = list(graph2);
```

```
#And now we call gSpan using a support
# of 80%
> results = gspan(database,
support = “%80”)
# Examine the output, which is
# an array of iGraph objects in
# list form.
> results
[[1]]
Vertices: 5
Edges: 10
Directed: TRUE
Edges

[0] ‘1’ -> ‘5’
[1] ‘5’ -> ‘1’
[2] ‘2’ -> ‘1’
[3] ‘1’ -> ‘2’
...
```

FSM Outline

- **FSM Preliminaries**
- **FSM Algorithms**
  - gSpan
  - SUBDUE
  - SLEUTH
- **Review**

What is SUBDUE?

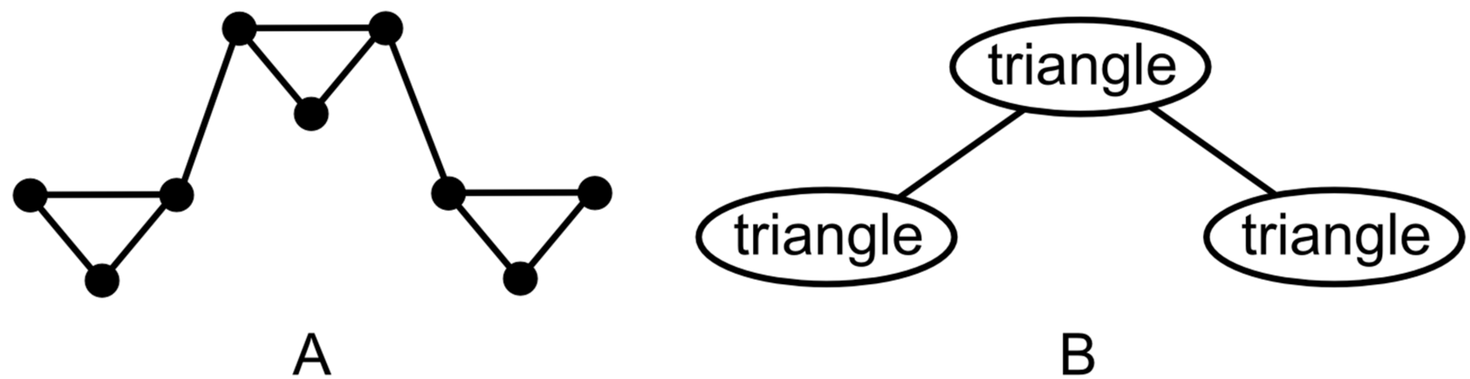
- **L.B. Holder described it in 1988.**
- **Uses *beam search* to discover frequent subgraphs.**
- **Reports *compressed* structures.**
- **Is an approximate version of FSM.**
- **Is not based on support**

Beam Search

- **Beam Search** is a best-first version of breadth-first search.
- At each level of search, only the best k children are expanded.
- k is called **Beam Width.**
- "Best" is a problem-dependent determination

Graph Compression

SUBDUE compresses graphs by replacing subgraphs with pointers.



<u>Before compression</u> → Figure A contains 3 triangles and has 11 edges.

<u>After compression</u> → Figure B, has 3 triangle pointers and has 2 edges.

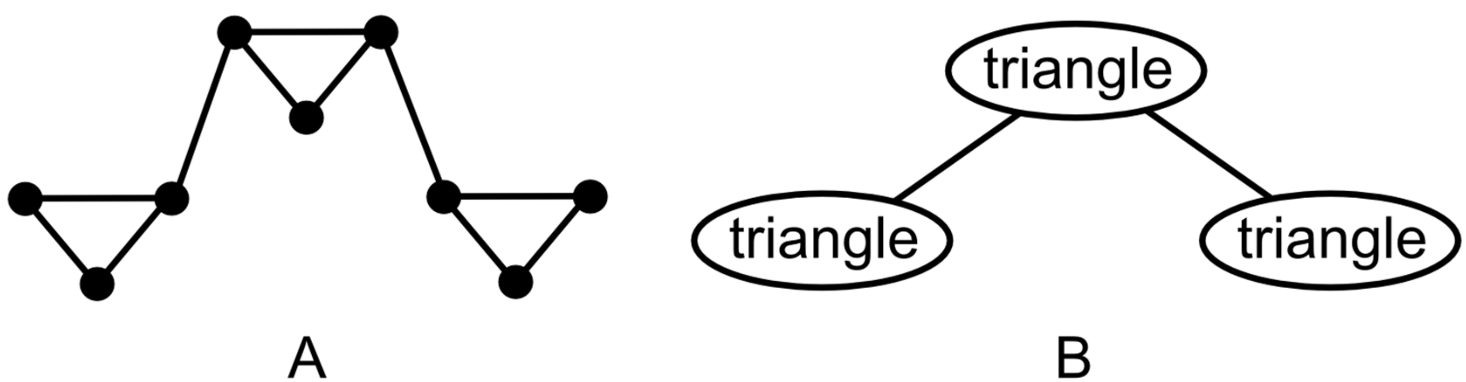
Compressed Description Length

- The **Description Length** of a graph G is the integer number of bits required to represent graph G in some binary format, which is denoted by DL(G).
- The **Compressed Description Length** of a graph G with some subgraph S is the integer number of bits required to represent G after it has been compressed using S, which is denoted DL(G|S).

Description Length Example

Vertex: 8 bits
Edge: 8 bits
Pointer: 4 bits

DL(A) = 9*8 + 11*8 + 0*4 = 160 bits.
DL(A|triangle) = 3*8 + 2*8 + 3*4 = 52 bits
DL(triangle) = 3*8 + 3*8 + 0*4 = 48 bits



SUBDUE Algorithm Overview

- **SUBDUE maintains a global set which holds the subgraphs that provide the overall best compression.**
- **The algorithm begins with all 1-vertex subgraphs**
- **During each iteration, SUBDUE checks to see if any of children (extended subgraphs of) are better candidates.**
- **After the children are considered, they become the new parents and the process starts over.**

SUBDUE Algorithm Pseudocode

- **Input: Graph database** *D*, **beam search width** *w*, **subgraph size** limit, **output size limit** max_best
- **Output: set of frequent subgraphs *S***
- **Pseudocode:**
  - parents ← all single-vertex subgraphs in D
  - search_depth ← 0
  - S ← ∅
  - while search_depth < limit and parents ≠ ∅
    - foreach parent
      - **generate up to beam_width best children**
      - insert children into S
      - remove all but max_best best elements of S
    - parents ← beam_width best children
    - search_depth ← search_depth + 1
- **Best: for subgraph G, minimize DL(D|G)+DL(G)**

set generated by adding all possible labeled edges

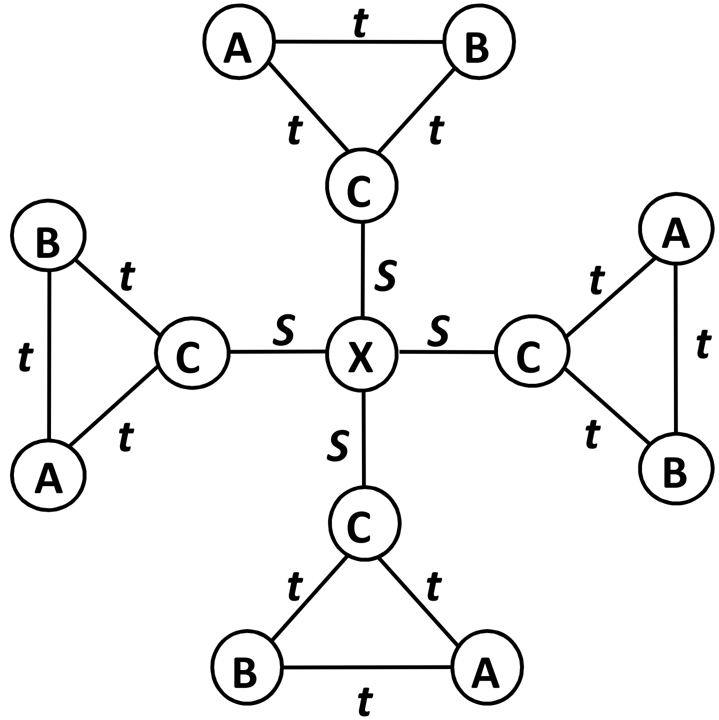
compression performed
using subgraph isomorph

SUBDUE Example

SUBDUE Encoding Bit Sizes

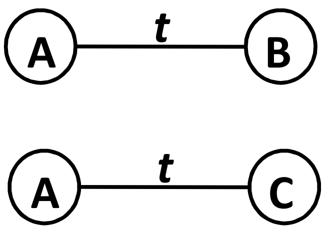
Vertex: 8 bits
Edge: 8 bits
Pointer: 4 bits

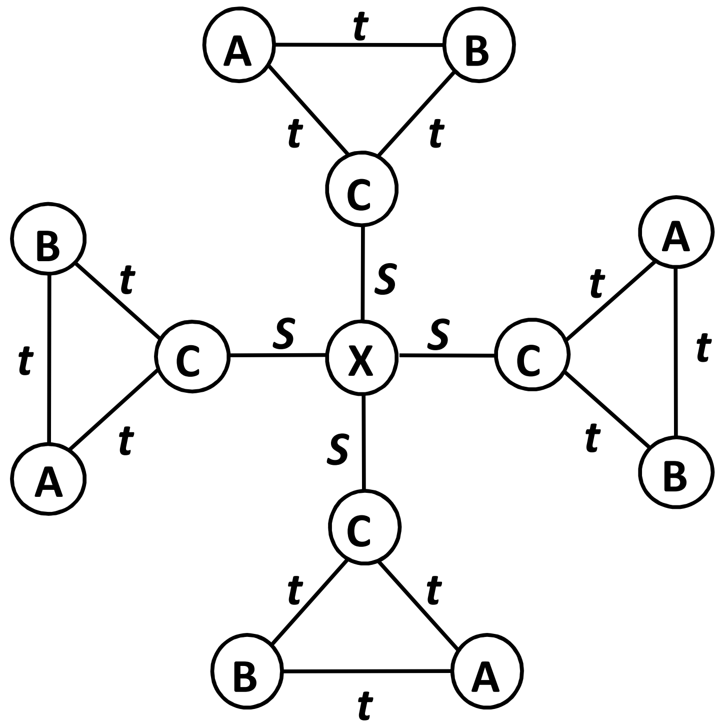
DL(pinwheel) = 13*8 + 16*8 +0*4 = 232 bits.



SUBDUE Example

First generation children of parent **A**:





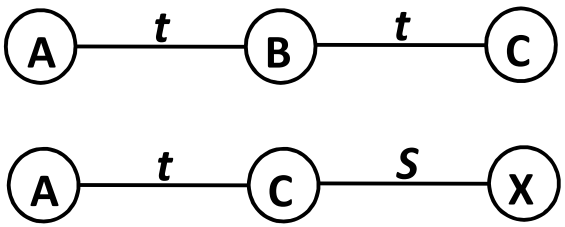
Description length computation (both the same):

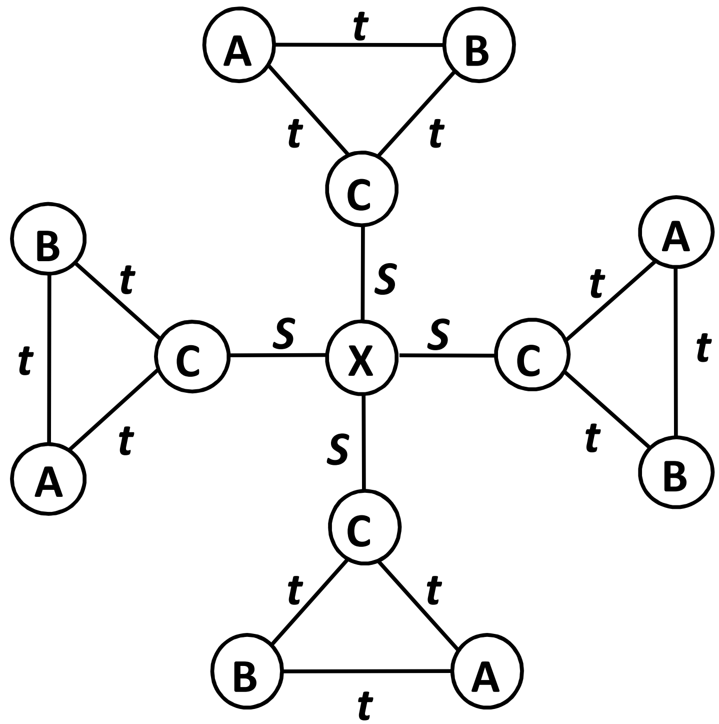
- 4 instances of subgraph
- Vertices after replacement: 13 → 9
- Edges after replacement: 16 → 12
- DL(pinwheel | A-B): 13*8 + 12*8 + 4*4 = 216 bits
- DL(A-B): 2*8 + 1*8 + 0*4 = 24
  - Improvement: 232 – 216 - 24 = -8 bits

Not yet worth it!

SUBDUE Example

Second generation children of parent **A**:





Description length computation using A-B-C

- 4 instances of subgraph
- Vertices after replacement: 13 → 5
- Edges after replacement: 16 → 8
- DL(pinwheel | A-B-C): 5*8 + 8*8 + 4*4 = 120 bits
- DL(A-B-C): 3*8 + 3*8 + 0*4 = 48 bits
  - Improvement: 232 – 120 – 48 = 64 bits

SUBDUE in R

- SubgraphMining **R package contains the functions to run SUBDUE.**
- **Written in C, but has Linux-specific source code.**
- **Compiled binaries are provided, and may use** make **and** make install **commands if it doesn't run on your system.**
- **Uses iGraph objects.**

```
#Import the subgraphMining package
> library(subgraphMining)
# Build your iGraph object. For this example
# we built the graph from Figure ~1.7
# using iGraph and called it graph1.
# Call SUBDUE.
# graph is the iGraph object to mine.
> results = subdue(graph);
# Examine the results
> results
```

FSM Outline

- **FSM Preliminaries**
- **FSM Algorithms**
  - gSpan
  - SUBDUE
  - SLEUTH
- **Review**

SLUETH Outline

- Introduction, preliminaries
- Data Representation
- Subtree generation and comparison
- SLUETH Algorithm

What is SLEUTH?

- **Written by Mohammed Zaki in 2005.**
- **Developed to target a special type of graph: trees**
  - HTML has a tree-like structure
- **Consider the following HTML tree (on the right)**
  - <TITLE> is a descendant of <HTML> and isn't a direct child. (no edge connection)
  - SLEUTH is used in instances like these to mine frequent subtrees.

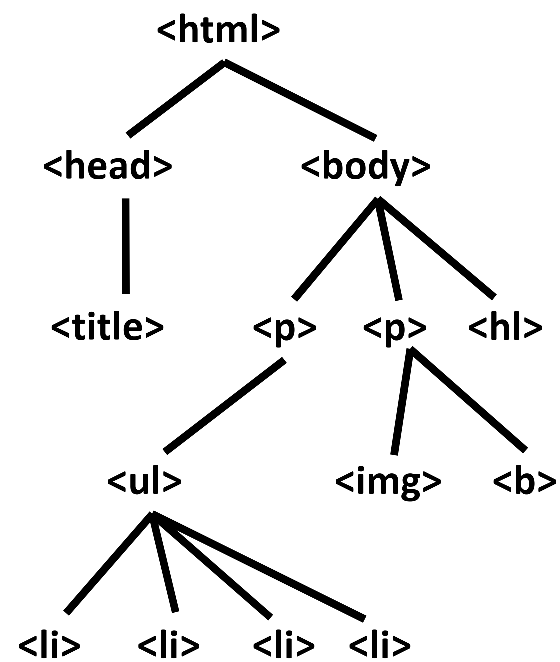
```
<HTML>
    <HEAD>
        <TITLE> My page about puppies! </TITLE>
    </HEAD>
    <BODY>
        <H1> Puppies are amazing. </H1>
        <P> This is a photo of my puppy. Her name is <B> Blix </B>.
            <IMG src="blix.jpg" />
        </P>
        <H1> These are the things I like about puppies: </H1>
        <P>
            <UL>
                <LI>Playful</LI>
                <LI>Cute</LI>
                <LI>Warm</LI>
                <LI>Fuzzy</LI>
            </UL>
        </P>
    </BODY>
</HTML>
```

SLEUTH Preliminaries

- A **tree** is a connected, directed graph T without any cycles.
- A **subtree** T_s is a subgraph of T which is also a tree.
- A tree is a **rooted** tree if a node is distinguished as the root.
- Two nodes are **siblings** if they share a parent and **cousins** if they share a common ancestor.
- A tree is **ordered** if each siblings have an assigned relative order.
- An **unordered** tree is if there is no relative ordering.

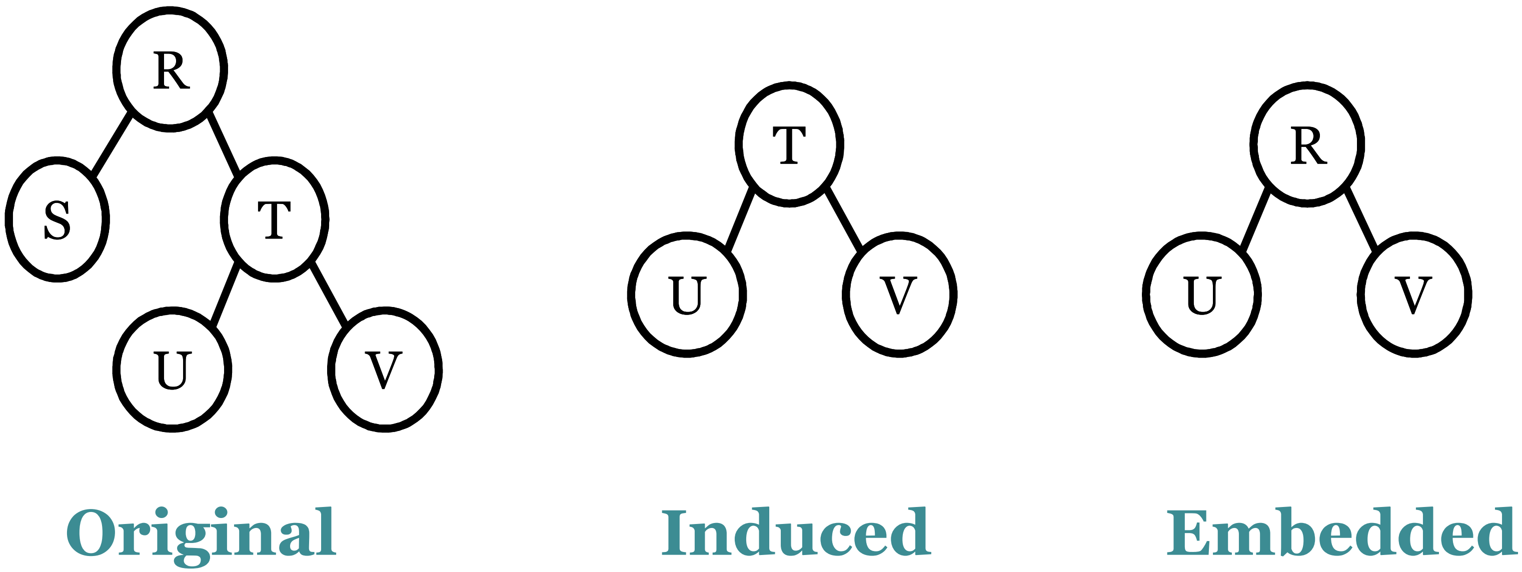
SLEUTH Preliminaries: HTML Example

```
<HTML>
   <HEAD>
      <TITLE> My page about puppies! </TITLE>
   </HEAD>
   <BODY>
      <H1> Puppies are amazing. </H1>
      <P> This is a photo of my puppy. Her name is <B> Blix </B>.
         <IMG src="blix.jpg" />
      </P>
      <H1> These are the things I like about puppies: </H1>
      <P>
         <UL>
            <LI>Playful</LI>
            <LI>Cute</LI>
            <LI>Warm</LI>
            <LI>Fuzzy</LI>
         </UL>
      </P>
   </BODY>
</HTML>
```



- <HTML> is the **parent** of node *<HEAD> and <HEAD>* is a **child** of *<HTML>*.
- *<HTML>* is the **ancestor** of node *<TITLE>*.
- <TITLE> is a **descendant** of <HTML>.

SLEUTH: Induced vs. Embedded



- **Induced** trees can only contains edges from the original tree
- **Embedded** trees can have edges between ancestors and descendants
  - The set of embedded trees is a superset of the set of induced trees
- SLEUTH mines embedded trees, not just induced ones

SLEUTH Motivation

- **Naïve approach → generates possible subtrees found within each pattern (keeping tally of occurrences).**
- **Consider collection of trees *D* with *k* vertices and *d* vertex labels**
- **The potential subtrees that are generated:**

$$candidates(D) = k^{k-2} \times k^{n}$$

- **To illustrate, consider the numbers of 4 labels (d = 4) and a maximum tree size of k = 1,2, … 7 (shown below)**

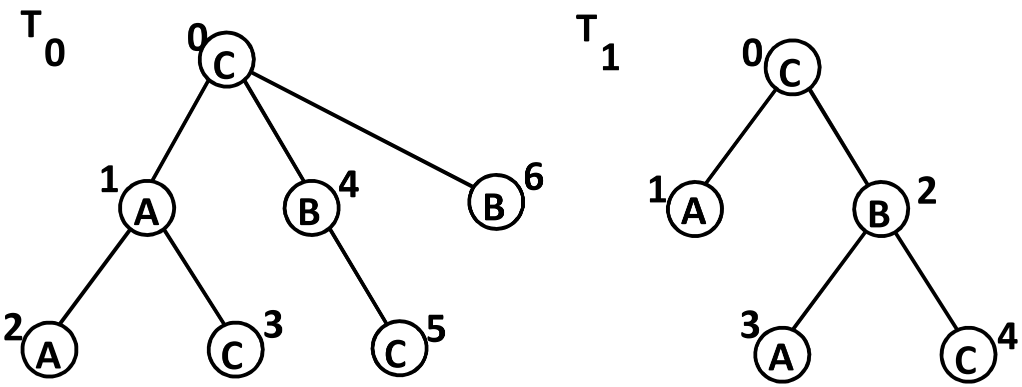
k	1	2	3	4	5	6
configurations	1	1	3	16	125	1296
labellings (d^k)	4	16	64	256	1024	4096
candidates	4	16	192	4096	128,000	5,308,416

SLUETH Outline

- Introduction, preliminaries
- Data Representation
- Subtree generation and comparison
- SLUETH Algorithm

Data Representation

- **Preorder traversal** is a visitation of nodes starting at the root by using depth-first search from left subtree to the right subtree.
- **SLEUTH represents horizontal and vertical formats.**
  - Horizontal → follows preorder traversal
  - Vertical → Lists (tree id, scope)
- **For unordered trees, preorder-based representation forces ordering among siblings**



Vertical Format (tree id, scope):

A	B	C
0, [1, 3]	0, [4, 5]	0, [0, 6]
0, [2, 2]	0, [6, 6]	0, [3, 3]
1, [1, 1]	1, [2, 4]	0, [5, 5]
1, [3, 3]		1, [0, 4]
		1, [4, 4]

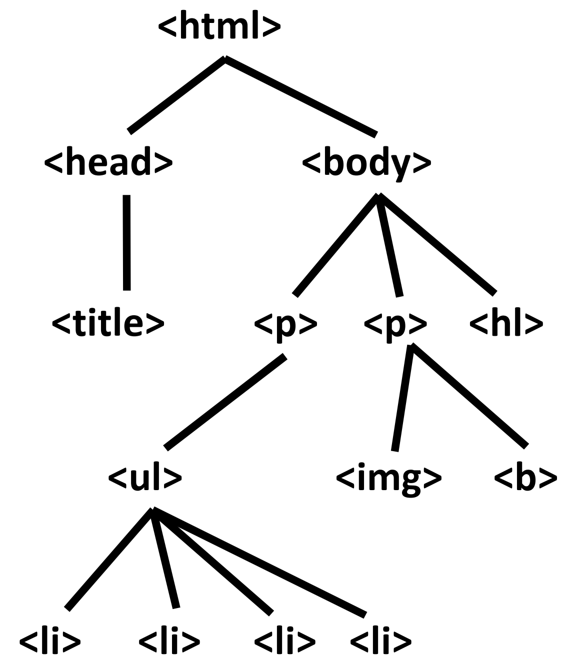
Horizontal Format (tree id, string encoding):

(T_0, C A A $ C $ $ B C $ $ B $)

(T_1, C A $ B A $ C $ $)

Data Representation

- **$ symbol is the backtracking from child to parent.**
- **The HTML document about puppies (on the right) can be encoded as '013$$24$56$7$$589$9$9$9$$$$.'**
- **Vertical format contains one scope-list for each label.**
- **Scope** is a pair of preorder position [*l*,*u*] where *l* is the vertex and u is the right-most descendant.



0	1	2	3	4	5	6	7	8	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
HTML	HEAD	BODY	TITLE	H1	P	IMG	B	UL	LI

SLUETH Outline

- Introduction, preliminaries
- Data Representation
- Subtree generation and comparison
- SLUETH Algorithm

Candidate Subtree Generation

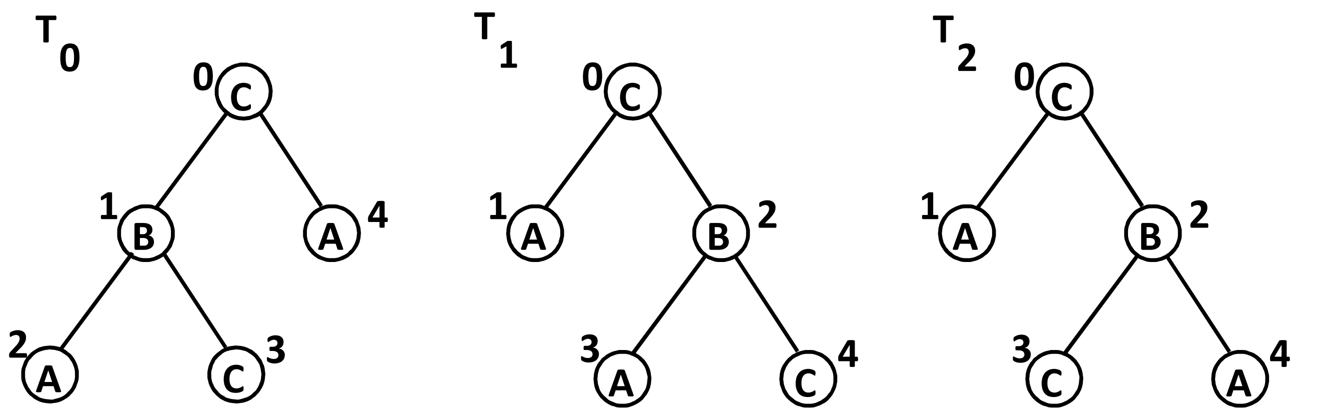
- SLEUTH limits candidate subtree generation by extending only frequent subtrees.
- **Prefix based extension** limits additions of new vertices to the tree to the rightmost path of the tree
- Candidate trees are extensions of the prefix tree.

Candidate may belong to automorphism group
(see next slide)

Candidate Subtree Generation

- **For unordered trees, prefix-based extension creates redundancy problem.**
- **Canonical form** lets you to recognize when you are dealing with the same graph.

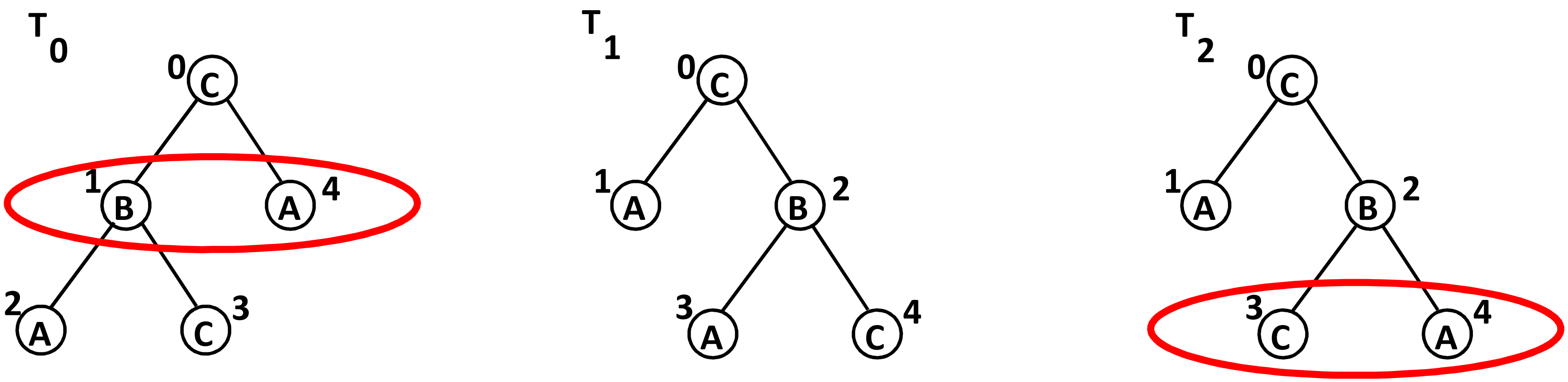
T0: CBA$C$$A$
T1: CABAC$$
T2: CABCA$$



These graphs are automorphic

Prefix Tree Canonical Form

- **Given label set $L = \{l_1, l_2, \dots, l_d\}$**
- **Given ordering $\prec$ where $l_1 \prec l_2 \prec \cdots \prec l_d$**
- **Tree T with vertex labeling ℓ is in canonical form if:**
  - for every vertex $v \in T$,
    - for all children of v, $c_1, c_2, \dots, c_k$, listed in *preorder*,
      - $\ell(c_i) \prec \ell(c_{i+1})$ for $i \in [1, k)$



NOT Canonical

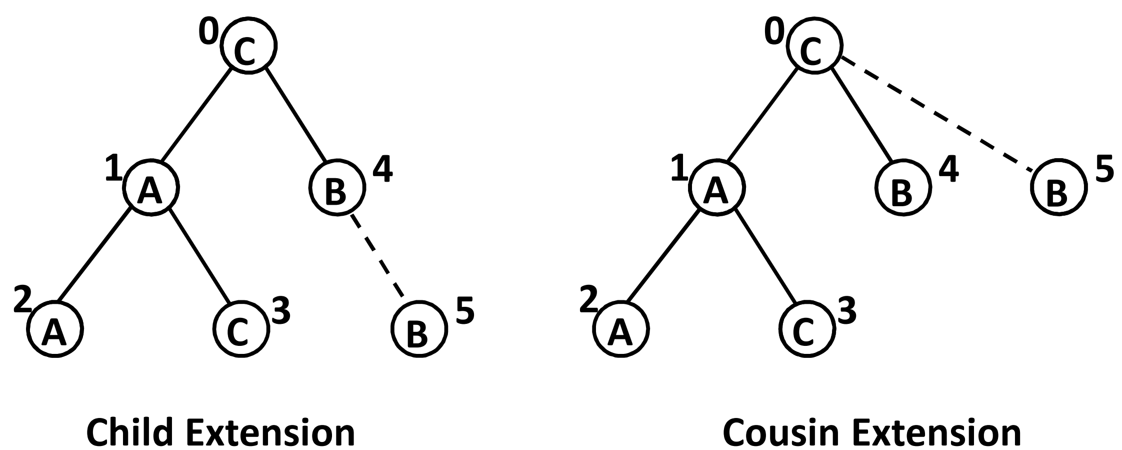
Canonical

NOT Canonical

Candidate Subtree Generation

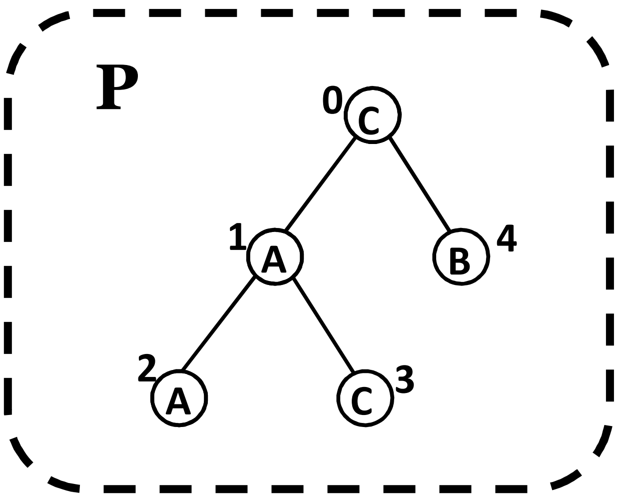
- **SLEUTH generates frequent subtrees using equivalence class-based extension**
  – Child extension → new vertex appended to right-most leaf in prefix subtree.
  – Cousin extension → new vertex appended to any vertex in descendents of right-most leaf of prefix subtree.
  – In either → new vertex become right-most leaf in new subtree.
  – All possible new trees are of the same *prefix equivalence class* (next slide)

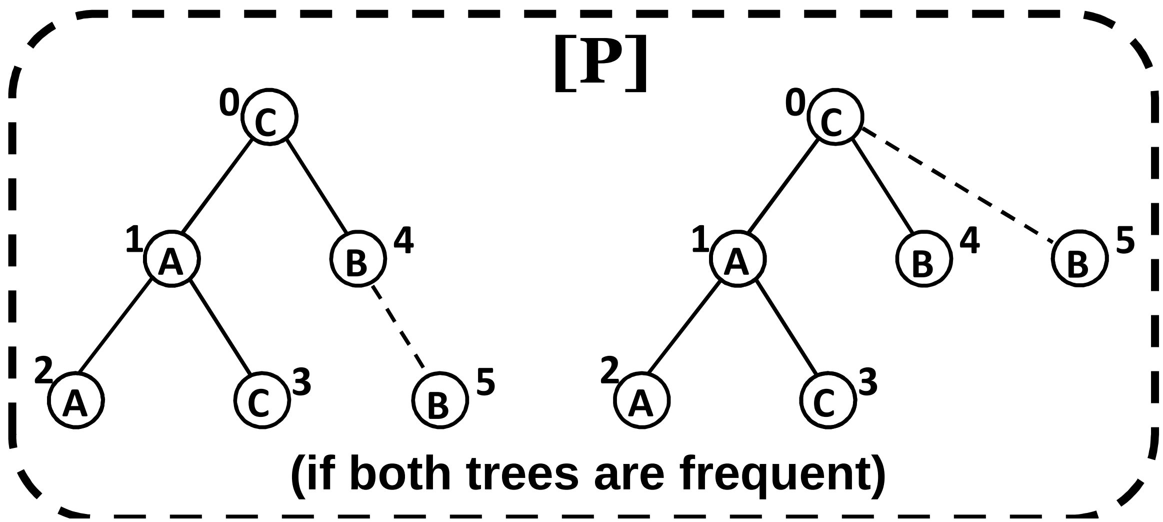
- **This tree is extended by vertex B to either vertex 0 (cousin) or vertex 4 (child).**



Prefix Equivalence Class

- **Set of all child/cousin extensions to a prefix tree**
  - For SLEUTH, the equivalence class also enforces that resulting subtrees be frequent.
- **Given: prefix tree *P***
- **Given label, vertex pair (x, i), let P_x^i denote the subtree created by attaching vertex *i* with label *x*.**
- **Frequent prefix tree equivalence class**
  - $[P] = \{(x, i) |\ P_x^i \text{ is frequent}\}$

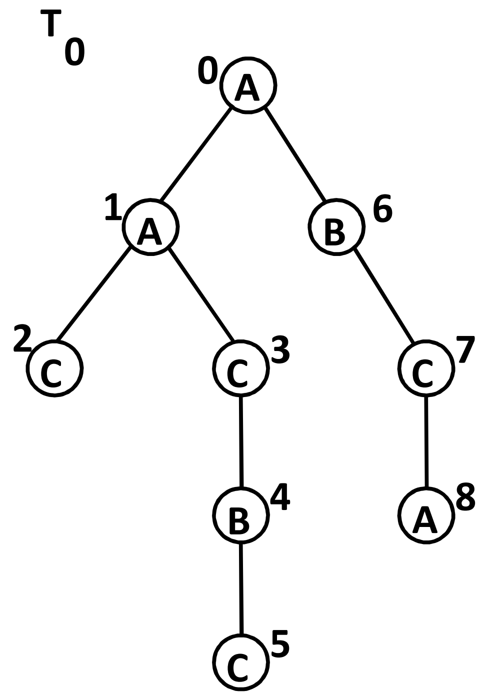




Support Computation – Match labels

- **SLEUTH uses scope lists, match-labels, and scope join lists to match generated subtrees to the input.**

- **Match-labels:**
  - preorder positions in containing tree of vertices in embedded tree

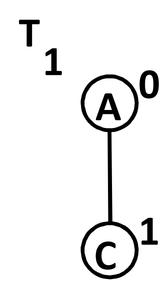


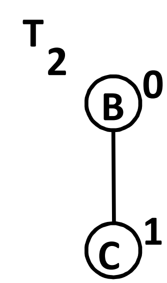
unordered embedded subtree match labels

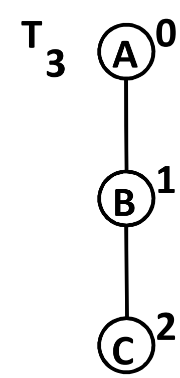
T_1 in T_0: {02, 03, 05, 07, 12, 13, 15}

T_2 in T_0: {45, 67}

T_3 in T_0: {045, 067, 145}

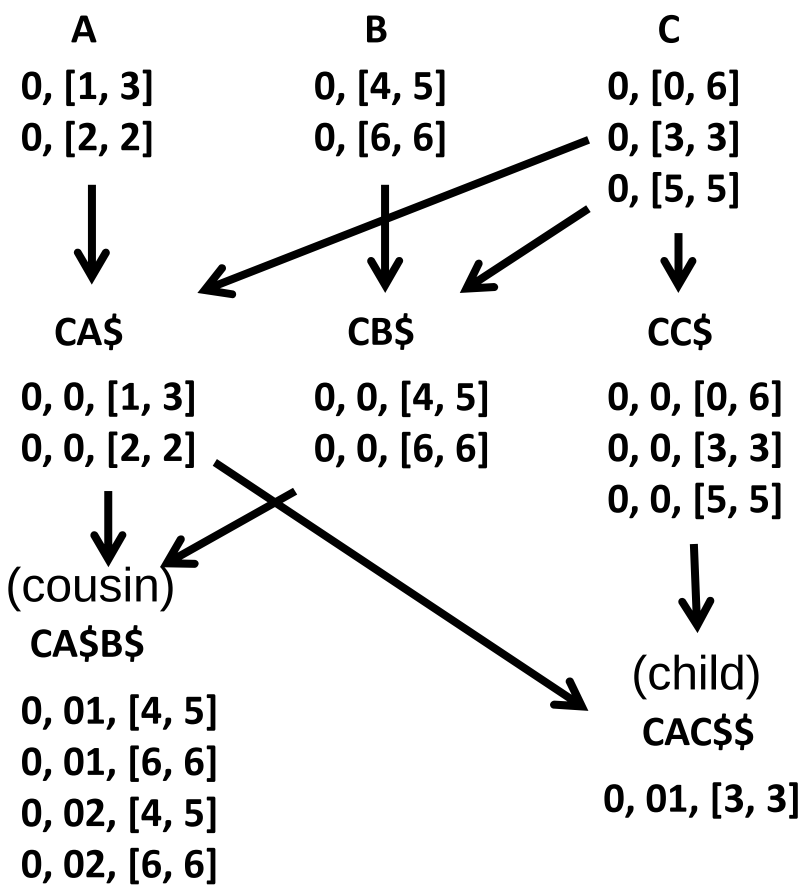


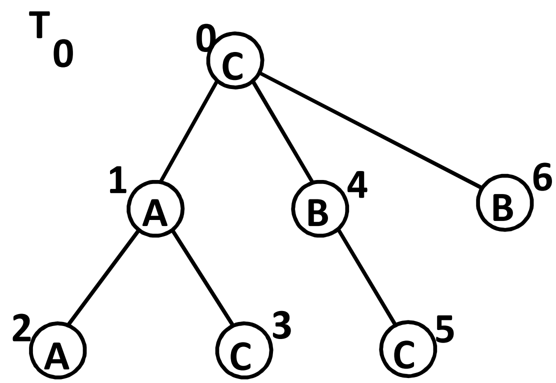




Support Computation – Scope-list Joins

- **Scope-list joins:**
  - scope list of subtrees (in horizontal format)
  - adds third field, the match label for the k-subtree





building scope-list joins:
use scope list to determine whether vertex is cousin or descendant

SLUETH Outline

- Introduction, preliminaries
- Data Representation
- Subtree generation and comparison
- SLUETH Algorithm

SLEUTH Algorithm - Initialize

- **Input: Tree database D, support boundary** threshold
- **Pseudocode:**
  - $F_1 \leftarrow$ frequent 1-subtrees (with scope lists)
  - $F_2 \leftarrow$ set of prefix equivalence classes of elements in F_1 (with scope lists)
  - for each $[P] \in F_2$
    - Enumerate-Frequent-Subtrees($[P], S$)
- **Top-level: compute all singleton subtrees, generate frequent extensions of the subtrees, then begin recursive procedure.**

SLEUTH Algorithm - Enumeration

- **Input: frequent prefix equivalence class** $[P]$
- **Pseudocode**
  - foreach added label,vertex pair (x, i) in $[P]$
    - if P_x^i is not canonical, skip to next pair
    - initialize $[P_x^i]$ to the prefix tree P_x^i and no extensions
    - foreach element $(y, j) \in [P]$ not equal to (x, i)
      - if (y, j) is a child or cousin extension of P_x^i and resulting tree is frequent:
        - » add (y, j) and/or $(y, k-1)$* to $[P_x^i]$, along with scope-lists
    - if $[P_x^i]$ contains no extensions, output P_x^i
    - else, recurse on $[P_x^i]$
- *** -** k: $[\mathbf{P_x^i}]$ **size. If** $\boldsymbol{i}$ **is a descendent of** $\boldsymbol{j}$**, then the extended vertex would now attach to** $\boldsymbol{k-1}$ **rather than** $\boldsymbol{i}$ **(see cousin vs. child scope-list join)**

SLEUTH in R

```
1 #Load the subgraphMining package into R
2 > library(subgraphMining)

3 # Call the SLEUTH algorithm
4 # database is an array of lists
5 # representing trees. See the README
6 # in the sleuth folder for how to
7 # encode these.
8 # support is a float.

9 > database = array(dim=2);
10 > database[1] = list(c(0,1,-1,2,0,-1,1,2,-1,-1,-1))
11 > database[2] =
          list(c(0,0,-1,2,1,2,-1,-1,0,-1,-1,1,-1))

12 > results = sleuth(database, support=.80);

13 # Examine the output, which will be
14 # encoded as trees like the input.
15 [1] "vtreeminer.exe –i input.txt
   –s 0.8 –o > output.g"
```

```
16 DBASE_NUM_TRANS : 2
17 DBASE_MAXITEM : 3
18 MINSUPPORT : 2 (0.8)
19 0 - 2
20 1 - 2
21 2 - 2
22 0 0 - 2
23 0 0 -1 1 - 2
24 0 0 -1 1 -1 1 - 2
25 0 0 -1 1 -1 2 – 2

26 ...

27 [1,3,3,0.001,0] [2,9,7,0,0] [3,38,11,0.001,0]
[4,60,11,0,0]
28 [5,53,5,0,0] [6,16,1,0,0] [7,2,0,0,0]
[SUM:181,38,0.002] 0.002
29 TIME = 0.002
30 BrachIt = 103
```

FSM Outline

- **FSM Preliminaries**
- **FSM Algorithms**
  - gSpan
  - SUBDUE
  - SLEUTH
- **Review**

Strengths and Weakness

- **Apriori-based Approach (Traditional):**
  - **Strength**: Simple to implement
  - **Weakness**: Inefficient
- **gSpan (and other Pattern Growth algorithms):**
  - **Strength**: More efficient than Apriori
  - **Weakness**: Still too slow on large data sets
- **SUBDUE**
  - **Strength**: Runs very quickly
  - **Weakness**: Uses a heuristic, so it may miss some frequent subgraphs
- **SLEUTH:**
  - **Strength**: Mines embedded trees, not just induced, much quicker than more general FSM
  - **Weakness**: Only works on trees… not all graphs