Advanced Techniques for Mining Structured Data:

Graph Mining

Frequent Subgraph Mining

Dr C.Loglisci

PhD Course in Computer Science and Mathematics XXXII cycle

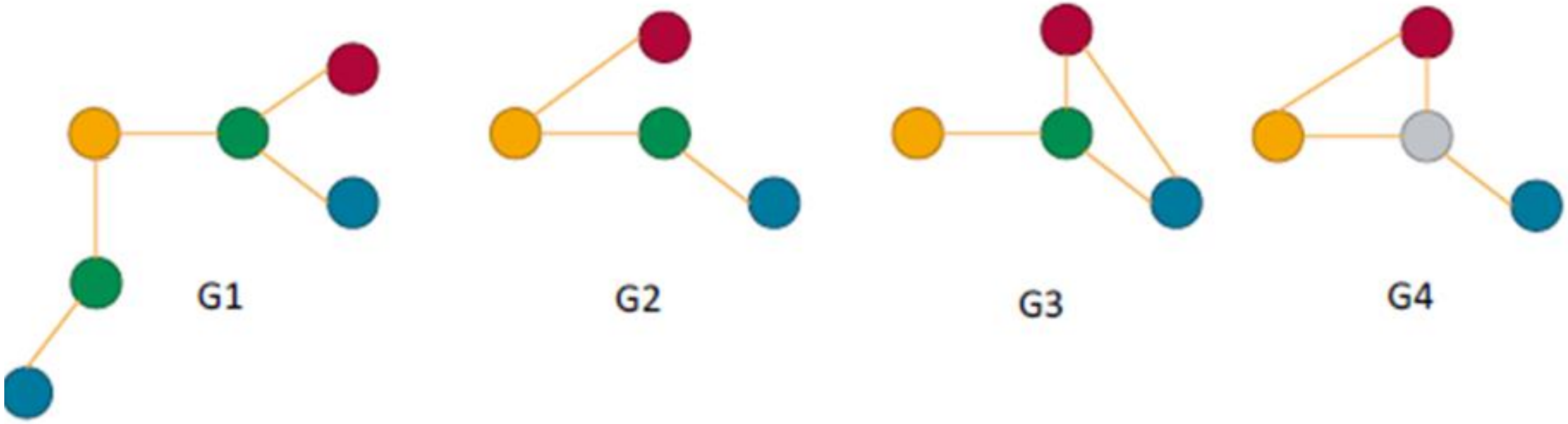
Frequent Subgraphs (Patterns)

- Discovery of graph structures that occur a significant number of times across a set of graphs

- Frequent subgraphs
    - A (sub)graph is frequent if its support (#occurrences) in a given dataset is no less than a minimum support threshold

- Examples are
    - Finding common biological pathways among species.
    - Recurring patterns of humans interaction during an epidemic.
    - Highlighting similar data to reveal data set as a whole

Frequent Subgraphs (Patterns)



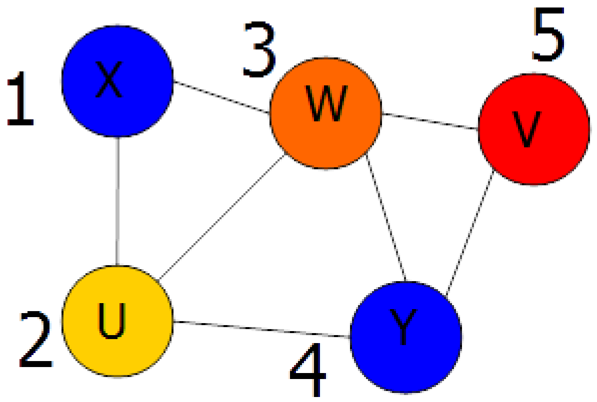
Frequent subgraph
Min support = 3

Frequent Subgraphs(Patterns)

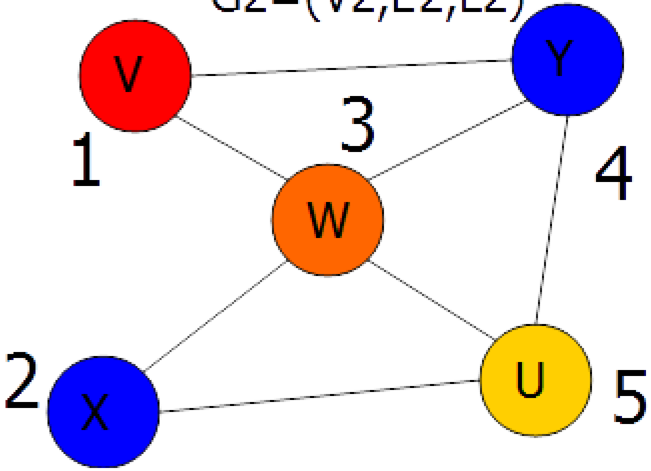
Recall the graph isomorphism problem:

- 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?

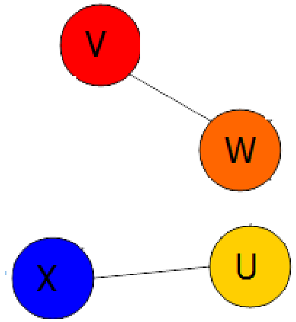
G1=(V1,E1,L1)



G2=(V2,E2,L2)



f(V1.1) = V2.2
f(V1.2) = V2.5
f(V1.3) = V2.3
f(V1.4) = V2.4
f(V1.5) = V2.1



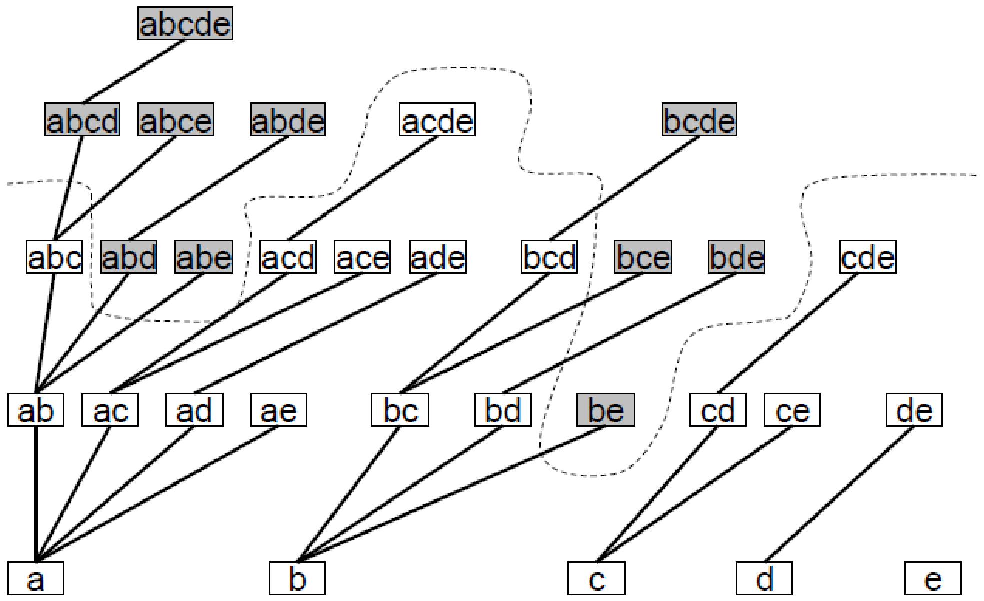
- components which are not connected are not of interest

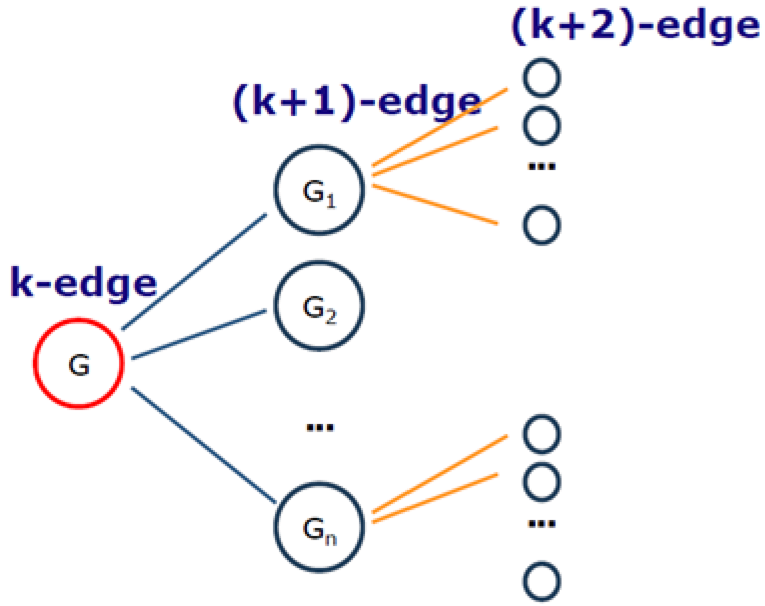
State-of-Art Approaches

- Pattern-growth approaches extends existing frequent sub-graphs by adding one edge
- Apriori-based approaches: joins (two) small-size patterns to create bigger size patterns (through Apriori principle)
- We will see two approaches, which employ anti-monotonic property of frequency

Pattern-growth with gSpan

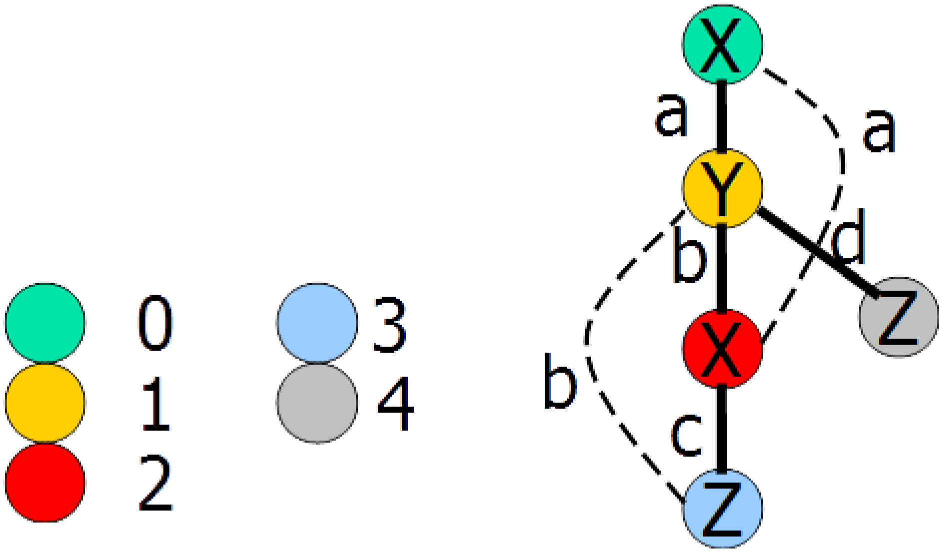
- Basic idea:
  - a canonical, efficient and univoque representation (codes) for graphs
  - lexicographic order for sorting the codes
  - tree search space based on the codes
  - building subgraphs by adding new edges
  - frequent subgraphs & tree pruning





Canonical representation : depth-first search code

- we use a (vi, vj, l(vi), l(vj), l(vi,vj)) to represent an edge
- transformation of a bi-dimensional structure into a sequence, which is easier to handle
- representation of the direction of exploration of the graph, *forward* edge (*vi* < *vj*) Vs *backward* edge (*vi* > *vj*)



e0: (0,1,x,y,a)

e1: (1,2,y,x,b)

e2: (2,0,x,x,a)

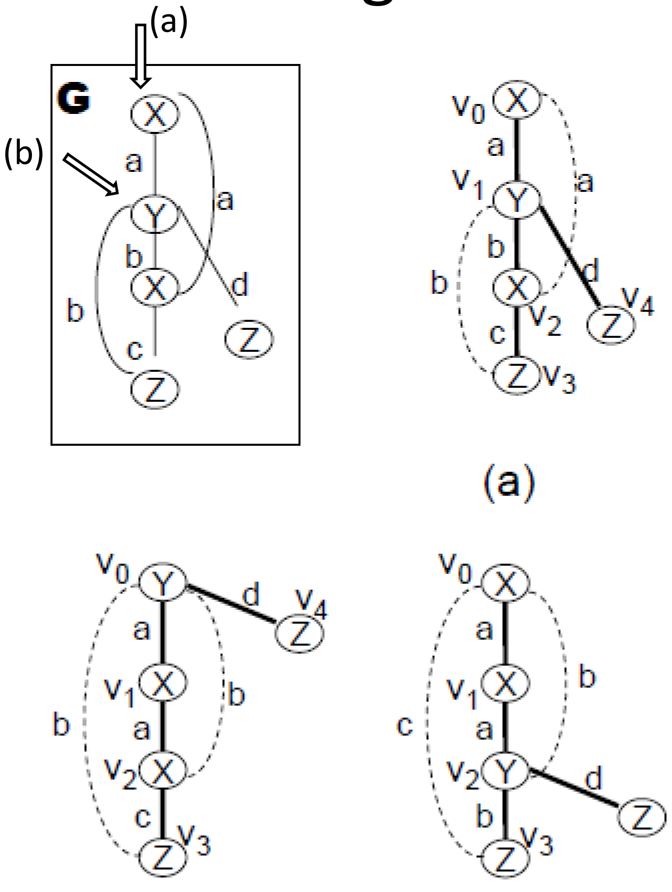
e3: (2,3,x,z,c)

e4: (3,1,x,y,b)

e5: (1,4,x,z,d)

Canonical representation : depth-first search code

- different (DFS) graph codes can be generated



(a)

(b)

(c)

	(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)

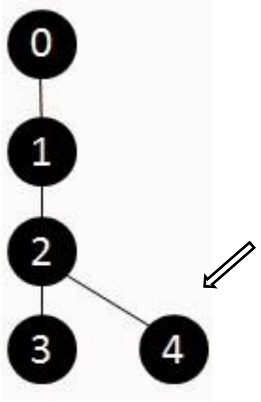
- we need a code to univocally identify graphs, compare them and add edges

Canonical representation : depth-first search code

- graph codes should have ordered edges
    - sorting intra-edges, within a graph
    - use of neighborhood restriction rules
    - these provide indication about how performing edge extension
- only one code is selected to represent the graph
- the minimum code (min(G)) is selected on the basis of lexicographic order on the labels (vertices and edges)
    - sorting intra-graphs
    - two codes for the same graph G, A:$(x0, x1, \ldots, xn)$ and B:$(y0, y1, \ldots, yn)$ have the relation $A \leq B$ iff:
        - there exists t, $0 \leq t \leq \min(m,n)$, $x_k = y_k$ for all k, s.t. $k<t$, and $x_t < y_t$
        - $x_k = y_k$ for all k, s.t. $0 \leq k \leq m$ and $m \leq n$.

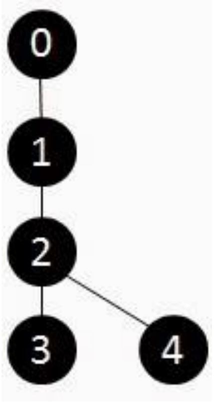
Tree search space

- it may be proved that two graphs A and B are isomorphic iff min(A)=min(B). This is used to count the occurrences
- given A: (x0, x1, …, xn),B: (x0, x1, …, xn,**b**)
  - A parent of B, B child A
  - sibling nodes organized in lexicographic order
  - given (z0, z1, …, zn) a non-minimum code, (z0, z1, …, zn,b), its child, is not minimum. It will be pruned
- graphs are extended by backward edges and forward edges
- to preserve the minimality of the code the following steps should be applied

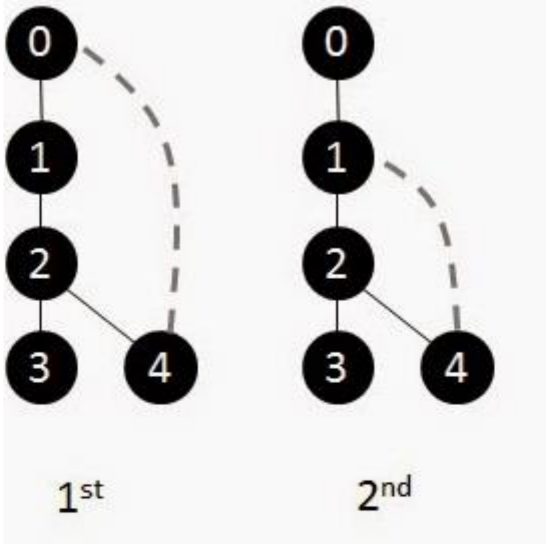


Building subgraphs

- all back edges first

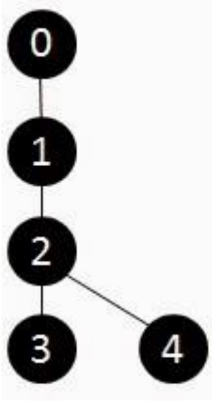


- …by using the (last) *rightmost* vertex in the code…

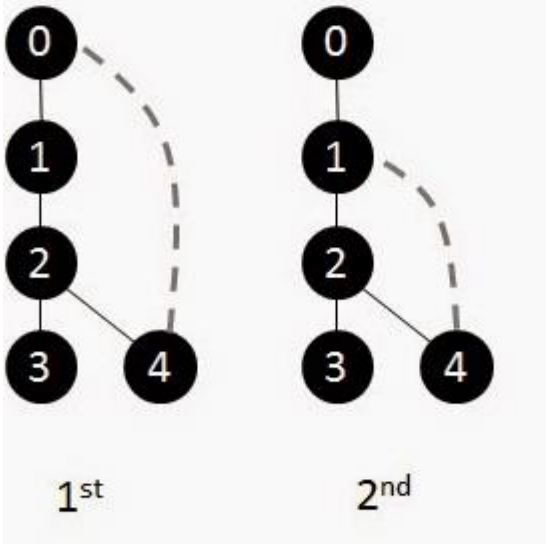


Building subgraphs

- all back edges first

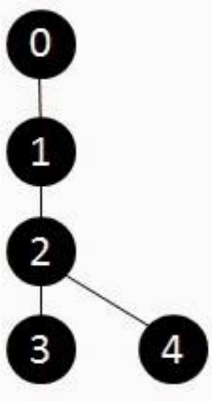


- …and the vertices of the rightmost path, by following the order in which they appear

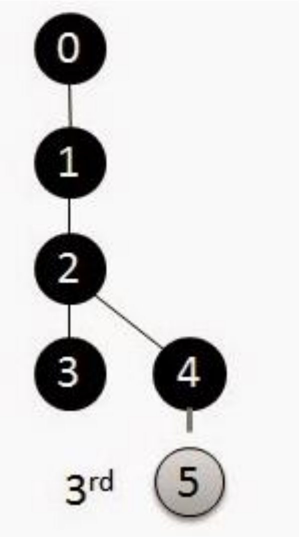


Building subgraphs

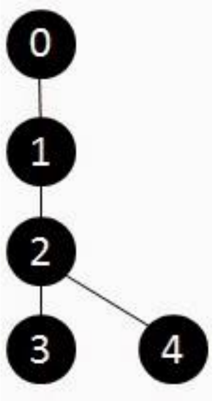
- then, forward edges by using the other nodes, as they appear in the lexicographic order…



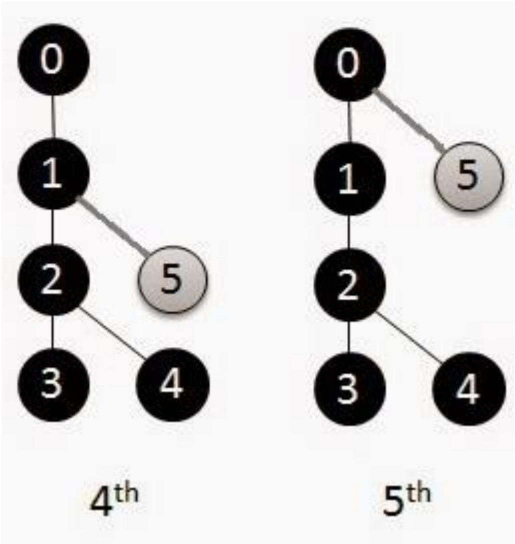
- ..and the vertices of the rightmost path…



Building subgraphs

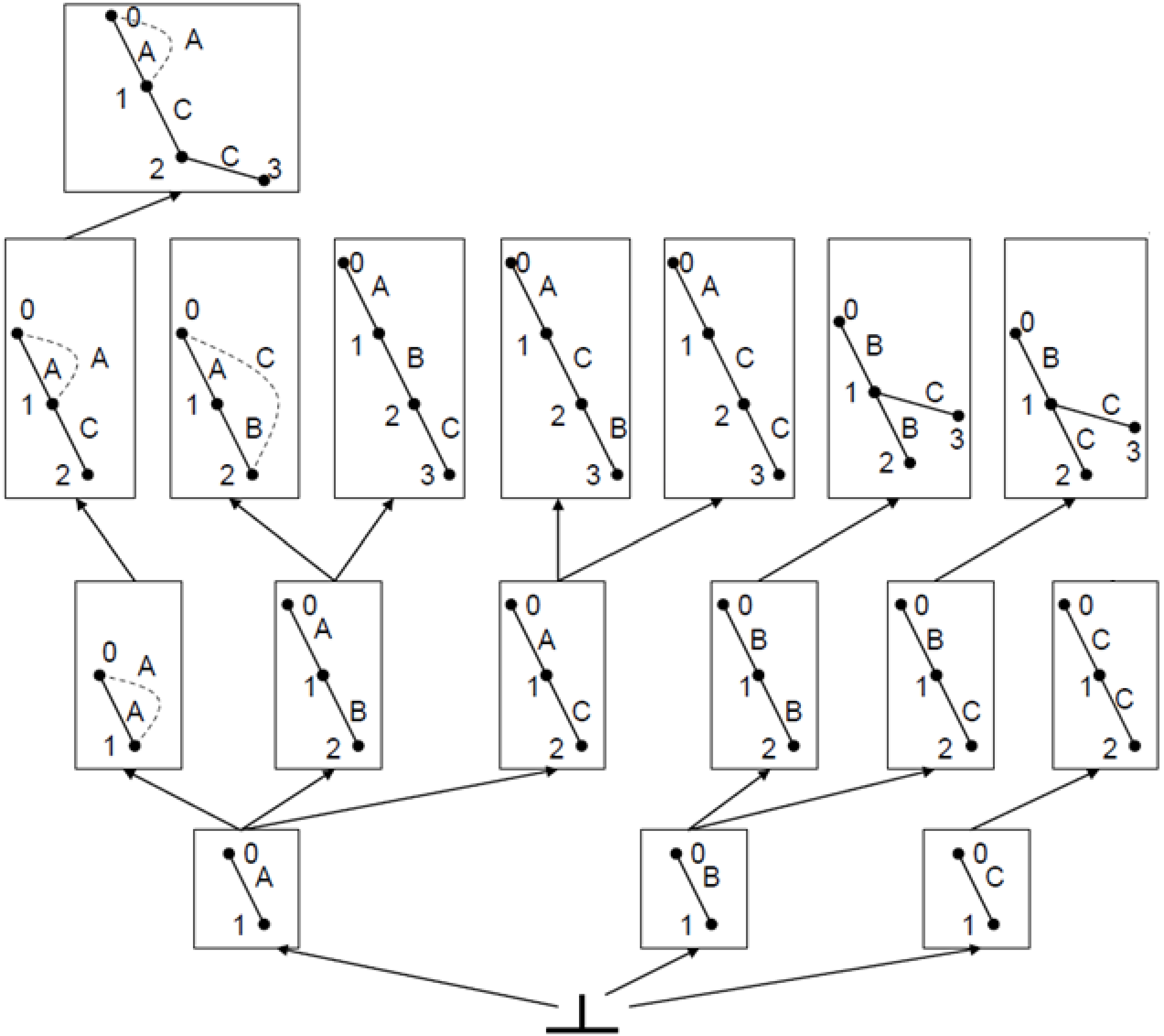


- …in the reverse order



Tree search space

- …finally, we have



Frequent subgraphs & Tree pruning

- Procedure gSpan(D,minS)→ S
  - compute frequent one-edge subgraphs in D → S1
  - sort S1 in lexicographic order
  - S ← S1
  - for each edge $e \in S1$
    - initialize s:<e>
    - add_new_edges(D,s,S,minS,S1)
  - remove s from all graphs in D (only consider subgraphs not already enumerated)

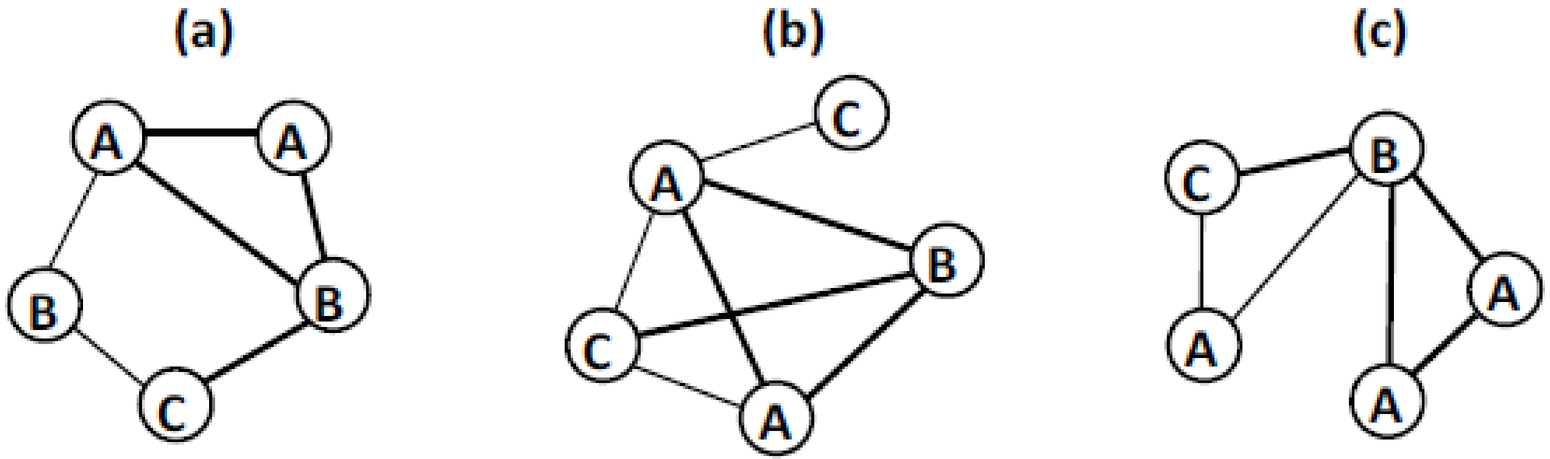
Frequent subgraphs & Tree pruning

- Procedure add_new_edges(D,s,S,minS,S1)
    - add s to S
    - for each extension x:<s,e>, e $\in$ S1
        - if supp(x) then add_new_edges(D,x,S,minS,S1)
        - else prune x

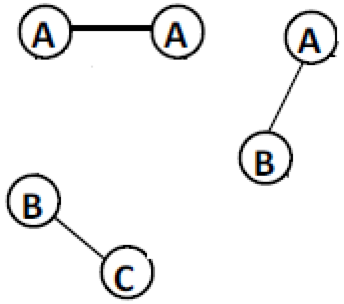
Frequent subgraphs & Tree pruning

- Example (simplified, without edge labels):

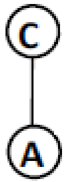
minS=3



Frequent:



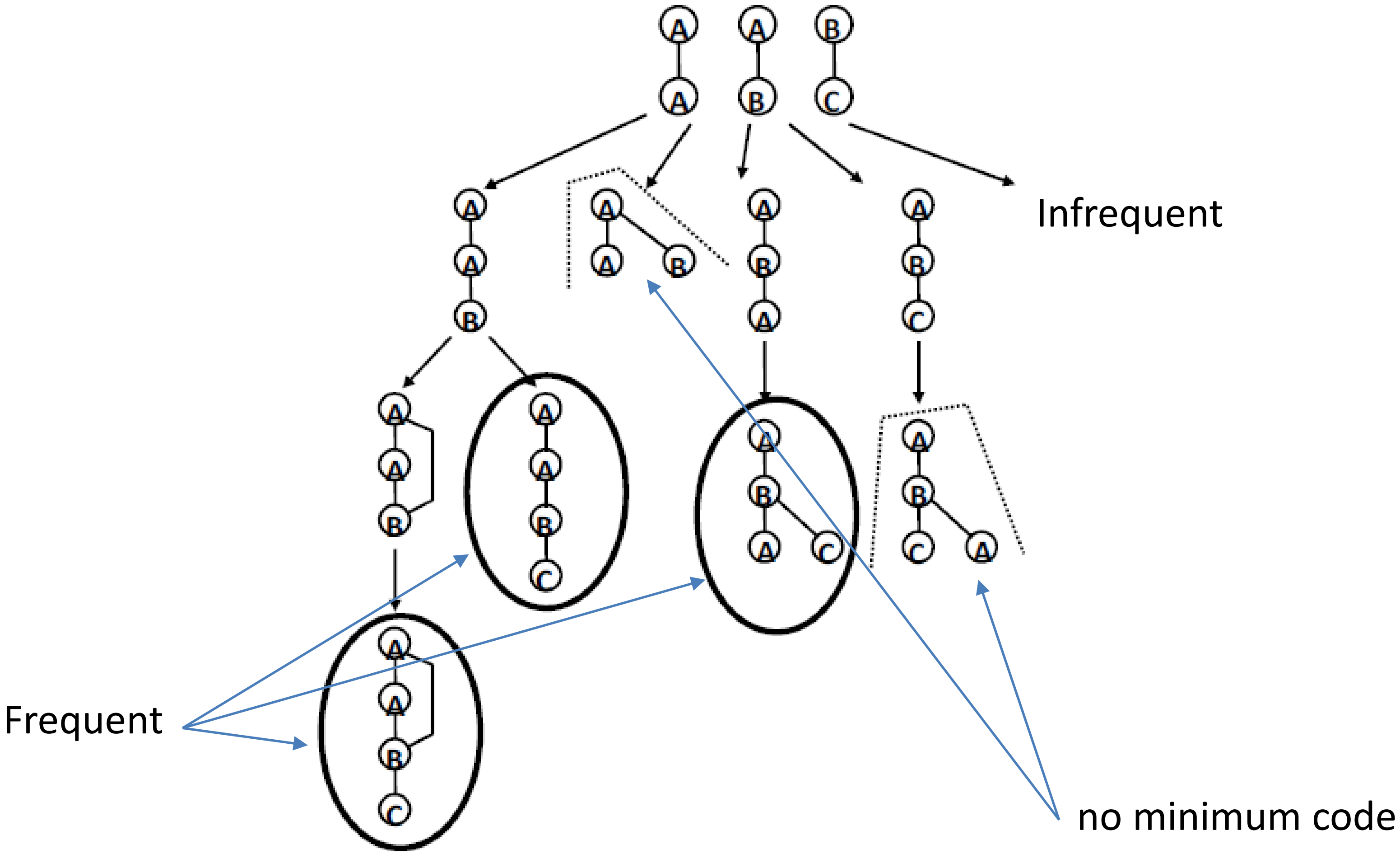
Infrequent:



Frequent subgraphs & Tree pruning

- Example (simplified, without edge labels):

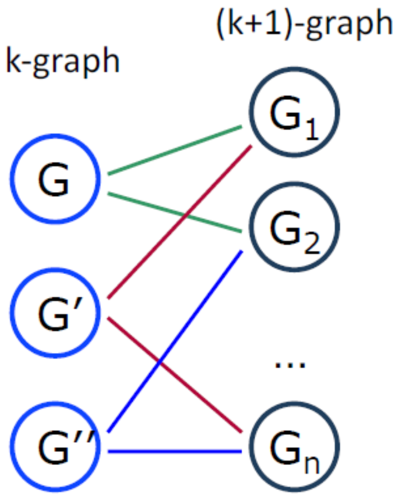
minS=3



Apriori-based approaches

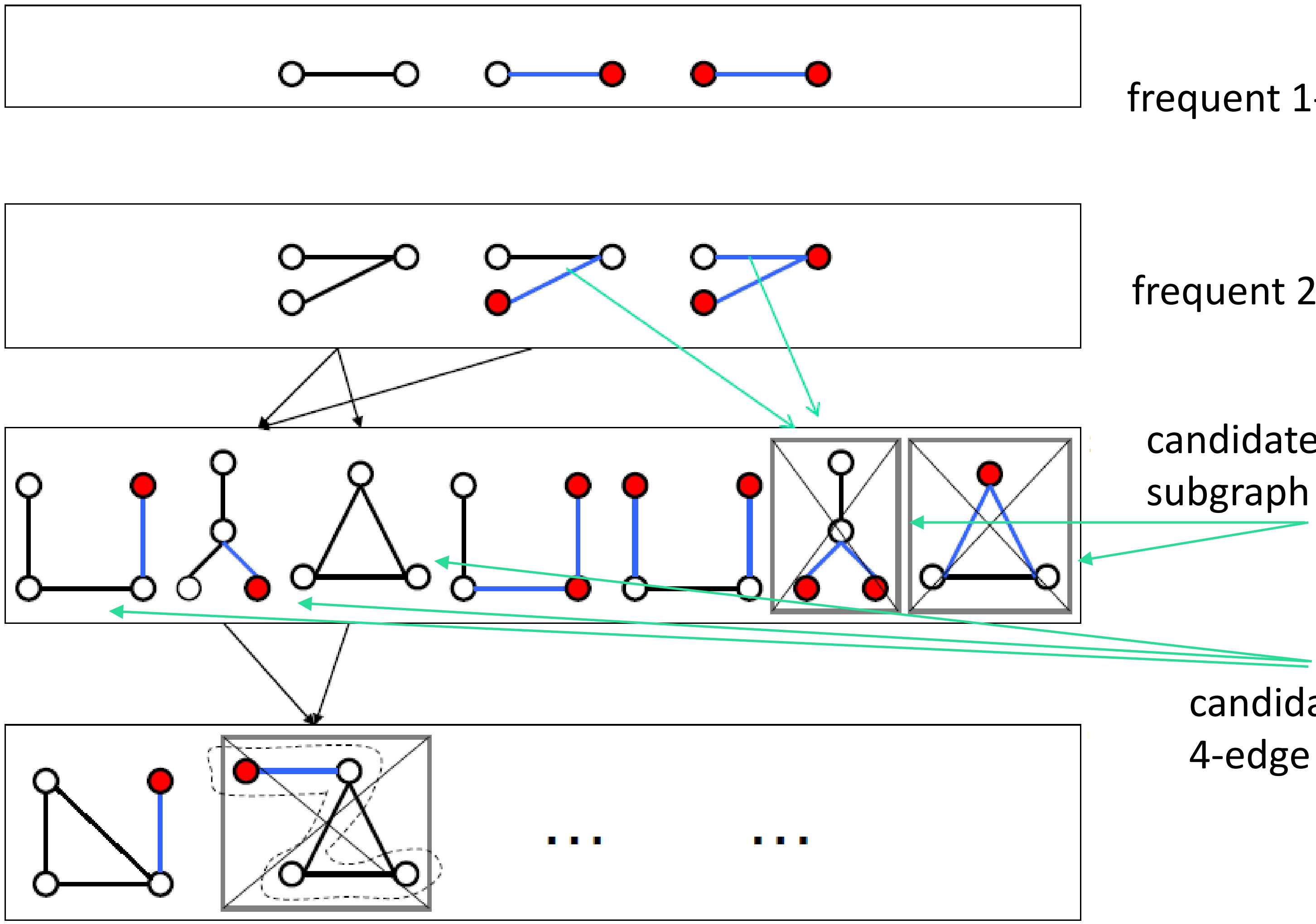
- Basic Idea
- level-by-level structure and breadth first search
- generate-and-test: candidate generation and then evaluation
- FSG (Kuramochi, 2002) algorithm
  - generates candidates by joining 2 frequent subgraphs to obtain one with one more edge.
  - then evaluates it and prunes it if the i) downward closure property is not satisfied, ii) support constraint is not satisfied
  - candidate generation uses sub-graph isomorphism
  - candidate evaluation uses graph-isomorphism

(removal duplicates) and sub-graph isomorphism (frequency)



Apriori-based approaches

- Basic Idea



frequent 1-edge subgraphs

frequent 2-edge subgraphs

candidate 3-edge subgraphs pruned: no subgraph isomorphism!

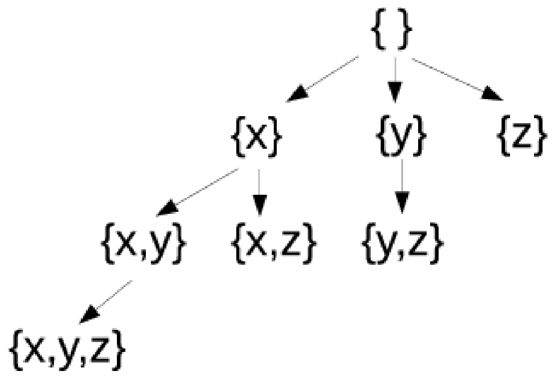
candidate 3-edge subgraphs to generate 4-edge subgraphs

candidate 4-edge duplicate pruned

Apriori-based approaches

- Basic Idea
- Set-enumeration tree (partial order) and breadth first search
- lexicographic order on vertices and edges labels
- generate-and-test: candidate generation and then evaluation
- @DIB algorithm
  - generates candidates (k+1)-edge patterns by joining 2 frequent k-edges patterns that satisfy the downward closure property
  - then evaluates and prunes it if the support constraint is not satisfied
  - candidate generation uses ~~sub-graph isomorphism~~ (uses heuristic: at least one k-edge patterns has to be a sub-graph, but necessary checking whether (k+1)-edge patterns is sub-graph)
  - candidate evaluation uses ~~graph isomorphism~~ (removal duplicates, no necessary by lex. order) and ~~sub-graph isomorphism~~ (frequency, no necessary by intersection TID-lists)

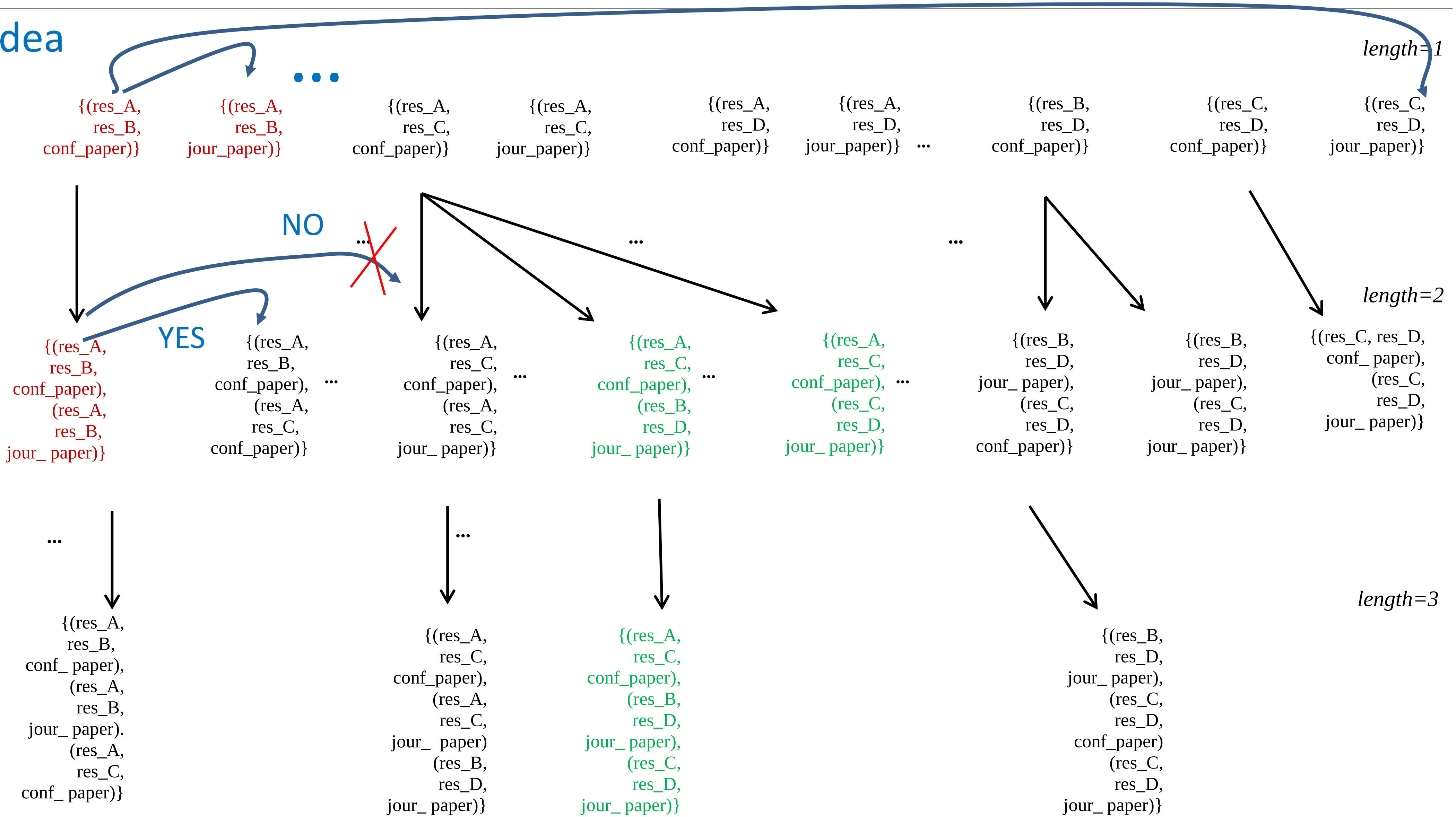
Apriori-based approaches



- Basic Idea
- Set-enumeration tree (partial order) and breadth first search
- lexicographic order on vertices and edges labels
- generate-and-test: candidate generation and then evaluation
- @DIB algorithm
  - generates candidates (k+1)-edge patterns by joining 2 frequent k-edges patterns that satisfy the downward closure property
  - then evaluates and prunes it if the i) ~~downward closure property is not satisfied~~(no necessary) ii) support constraint is not satisfied
  - candidate generation uses ~~sub-graph isomorphism~~ (uses heuristic: at least one k-edge patterns has to be a sub-graph, but necessary checking whether (k+1)-edge patterns is sub-graph)
  - candidate evaluation uses ~~graph isomorphism~~ (removal duplicates, no necessary by lex. order) and ~~sub-graph isomorphism~~ (frequency, no necessary by intersection TID-lists)

Apriori-based approaches

- Basic Idea



length=1

{(res_A, res_B, conf_paper)}

{(res_A, res_B, jour_paper)}

...

{(res_A, res_C, conf_paper)}

{(res_A, res_C, jour_paper)}

{(res_A, res_D, conf_paper)}

{(res_A, res_D, jour_paper)} ...

{(res_B, res_D, conf_paper)}

{(res_C, res_D, conf_paper)}

{(res_C, res_D, jour_paper)}

NO

... ...

length=2

{(res_A, res_B, conf_paper), (res_A, res_B, jour_ paper)}

YES

{(res_A, res_B, conf_paper), (res_A, res_C, conf_paper)} ...

{(res_A, res_C, conf_paper), (res_A, res_C, jour_ paper)} ...

{(res_A, res_C, conf_paper), (res_B, res_D, jour_ paper)} ...

{(res_A, res_C, conf_paper), (res_C, res_D, jour_ paper)} ...

{(res_B, res_D, jour_ paper), (res_C, res_D, conf_paper)}

{(res_B, res_D, jour_ paper), (res_C, res_D, jour_ paper)}

{(res_C, res_D, conf_ paper), (res_C, res_D, jour_ paper)}

... ...

length=3

{(res_A, res_B, conf_ paper), (res_A, res_B, jour_ paper). (res_A, res_C, conf_ paper)}

{(res_A, res_C, conf_paper), (res_A, res_C, jour_ paper) (res_B, res_D, jour_ paper)}

{(res_A, res_C, conf_paper), (res_B, res_D, jour_ paper), (res_C, res_D, jour_ paper)}

{(res_B, res_D, jour_ paper), (res_C, res_D, conf_paper) (res_C, res_D, jour_ paper)}

References

- Xifeng and H. Jiawei, gSpan: Graph-Based Substructure Pattern Mining, Tech. report, University of Illinois at Urbana-Champaign, 2002.
- M. Kuramochi and G. Karypis, An Efficient Algorithm for Discovering Frequent Subgraphs, Tech. report, Department of Computer Science/Army HPC Research Center, 2002.

Neighborhood restriction rules

- *If the first vertex of the current edge is less than the 2nd vertex of the current edge (forward edge)*
  - *If the first vertex of the next edge is less than the 2nd vertex of the next edge (forward edge)*
    - *If the first vertex of the next edge is less than or equal to the 2nd vertex of the current edge*
    - *AND If the 2nd vertex of the next edge is equal to the 2nd vertex of the current edge plus one <u>this is an acceptable next edge</u>*
    - *Otherwise <u>the next edge being considered isn't valid</u>*
  - *Otherwise (next edge is a backward edge)*
    - *If the first vertex of the next edge is equal to the 2nd vertex of the current edge*
    - *AND If the 2nd vertex of the next edge is less than the 1st vertex of the current edge <u>this is an acceptable next edge</u>*
    - *Otherwise <u>the next edge being considered isn't valid</u>*
- *Otherwise (the current edge is a backward edge*
  - *If the first vertex of the next edge is less than the 2nd vertex of the next edge (forward edge)*
    - *If the first vertex of the next edge is less than or equal to the 1st vertex of the current edge*
    - *AND If the 2nd vertex of the next edge is equal to the 1st vertex of the current edge plus one <u>this is an acceptable next edge</u>*
    - *Otherwise <u>the next edge being considered isn't valid</u>*
  - *Otherwise (next edge is a backward edge)*
    - *If the first vertex of the next edge is equal to the 1st vertex of the current edge*
    - *AND If the 2nd vertex of the current edge is less than the 2nd vertex of the next edge <u>this is an acceptable next edge</u>*
    - *Otherwise <u>the next edge being considered isn't valid</u>*