Knowledge Discovery in Databases II

Winter Term 2014/2015

Chapter 5: Linked Data

Lectures : PD Dr Matthias Schubert
Tutorials: Markus Mauder, Sebastian Hollizeck

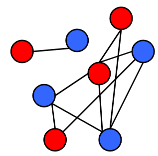
http://www.dbs.ifi.lmu.de/cms/Knowledge_Discovery_in_Databases_II_(KDD_II)

Chapter Overview

1. Graphs, Networks and Linked Data
2. Similiarity and Distance Measures for Graph Data
3. Frequent Subgraph Mining
4. Ranking Nodes and Centrality
5. Link Prediciton
6. Graph Clustering

An introduction to graphs

- **Definition**: A graph is a tuple $G=(V,E)$ where V is a set of vertices and $E \subseteq V \times V$ a set of edges.



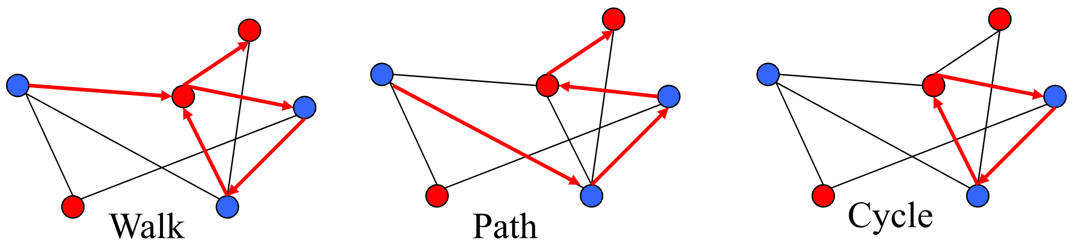
- Usually: vertices = objects, edges =relationships between objects
- A graph is representable as a quadratic matrix where each objects corresponds to a row and a column (Adjacency Matrix)
- Comparing graphs is expensive because there are

An introduction to graphs

- **node degree:** The degree of a node v_i in G=(V,E) denoted as $d_G(v_i)$ is number of adjacent edges: $d_G(v_i) = \left|\{v_j \mid (v_i, v_j) \in E\}\right|$
- **adjacency matrix** : The adjacency matrix of a graph G=(V,E) is defined as:

$$[A]_{i,j} = \begin{cases} 1 & \text{if} \quad (v_i, v_j) \in E \\ 0 & \text{else} \end{cases}$$

- **Walk**: A walk $w=(v_1,v_2,..,v_k)$ *is a sequence of nodes* $v_i \in V$ *where* $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$.
- **Path**: w is a path if $v_i \neq v_j$ with $i \neq j$.
  (=> no node is allowed to appear twice.)
- **Cycle:** Let $w=(v_1,..,v_k)$, $v_1 = v_k$ and for all 1 < i,j < k it hold that $v_i \neq v_j$ then w is called cycle.



An introduction to graphs

Directed or undirected graphs:

undirected graph: $(v_k, v_l) \neq (v_l, v_k)$, adjacency matrix is not symmetric

labeled graphs: Let F_V and F_E be Feature Spaces.

node labels: for every node $v \in V$ there is a label $l_v \in F_E$.

edge labels: for each edge $e \in E$ there is a edge label $l_e \in F_E$.

Remarks:

- Labels can be arbitrary types of information
- In most cases, labels are symbols from a given alphabet

Examples

- Molecule structures
- Protein interaction networks
- Social Networks
- WWW and other social media
- Spatial Networks

Comparing Graphs

Input: 2 Graphs G and G‘.

Output: Mapping $s:(V \times E) \times (V \times E) \rightarrow IR$ computing the similarity of G and G‘.

Approaches:

Isomorphism: 2 Graphs are equal if there exists a bijection between nodes inducing a bijection of edges.

=> Similarity decreases with the non-isomorphic parts

Edit-Distance: Similarity is computing by counting the minimal amount of operations transforming one graph into the other.

Topological Descriptors: Two Graphs are similar if the have similar values w.r.t. topological properties, e.g. number of edges, nodes, node degrees, label distributions,...

Graph Isomorphism

Graph-Isomorphism:

Let *G=(V,E)* and *G‘=(V‘,E‘) be two* graphs. *G* and *G*‘ are isomorph ($G \cong G'$) if there exists a bijection $f: V \rightarrow V'$ where $(v,v') \in E \Leftrightarrow (f(v),f(v')) \in E'$ fo all node pairs $v,v' \in V$.

Subgraph: Let *G =(V,E)* be a graph then *G‘=(V‘,E‘)* is a subgraph of *G*, if $V' \subseteq V$ and $E' \subseteq (V' \times V' \cap E)$.

Subgraph-Isomorphism: Let *G=(V,E) and G‘=(V‘,E‘) be graphs*. Then, *G‘* is subgraph isomorphic to *G* if there is a subgraph *G“* of *G* being isomorphic to G‘ ($G'' \cong G'$).

Maximal Common Subgraph : Let *G=(V,E)* and *G‘=(V‘,E‘)* be 2 Graphs. A graph *S* is maximal common subgraph *mcs(G,G‘)* if *S* is a subgraph of *G* and *G‘* and there is no other common subgraph *S*‘ having more nodes.

Minimal Common Super graph: Let *G=(V,E)* and *G‘=(V‘,E‘)* be 2 Graphs. A graph *S* is a minimal common super graph *MCS(G,G‘)* if *G* and *G‘* are subgraphs of *S* and there is no other graph containing G and G’ having less nodes.

Similarity based on Graph Isomorphism

mcs: Max Common Subgraph, MCS: Minimal Common Super Graph

- **Distance Measure 1**: Relative size of the minimal common subgraphs

$$d_1(G,G') = 1 - \frac{|mcs(G,G')|}{\max(|G|,|G'|)}$$

- **Distance Measure 2:** Difference of the size of MCS(G,G') and mcs(G,G')

$$d_2(G,G') = |MCS(G,G')| - |mcs(G,G')|$$

- Depends on the definition of the size:
  e.g. number of nodes => distance might be 0 for different graphs

- MCS and mcs require to solve the subgraph isomorphism problem (NP-hard).

Edit Distances for Graphs

Idea: Distance = minimal costs to transform *G* to *G'*.

- differences are removed by performing graph operations: Delete, Add, relabel nodes and edges
- Costs for each operation might vary depending on the labels
- Metric properties rely on the employed costs
- ***Graph Matching Distance*** between G and G' is defined as:

$$d(G,G') = \min_S \{c(S) | S \text{ sequence of operation transforming G into G'}\}$$

where *c(S)* is the sum of edit costs.

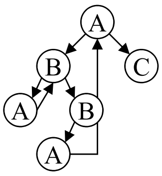
Problem:

- Problem still has to solve graph- and subgraph isomorphism problems
  => computation is very expensive

Edit Distances for Graphs

Performance:

- in general cases the complexity cannot be descreased
- for special cases faster methods are possible
  e.g. tree
  => unique serialisations are generall possible (order of subtrees)
  => Edit-distance for strings is in $O(n^2)$
  => Problem: Insertion costs have to selected to fit the change of topology



➡ [A[B[A][B[A]]][C]] ➡ [A[B[A][B[A]]][C]]

↕ Deletion of A in a leaf node

[A[B[A][B]][C]]

Conclusions

- Mathematically sound approach
- graphs can be compared on all of their properties
- Isomorphism-based methods depend on the definition of $|G|$
- Edit-Distance is a generalization of isomorphism-based methods
- computational complexity is very high (Subgraph Isomorphism is NP hard)
- limiting the problem to certain types of topologies can reduce the complexity

DATABASE SYSTEMS GROUP

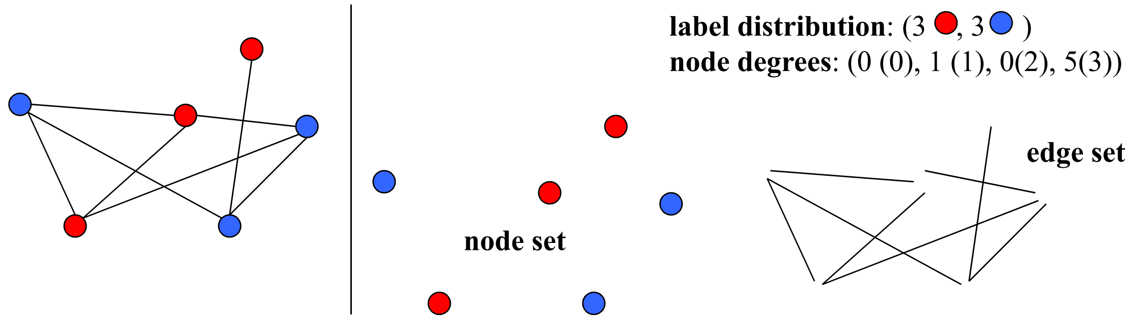
Topological Descriptors and Graph Kernels

Idea: Since isomorphism-based approaches are too expensive
=> compare topological graph properties

graph properties:

- Graph Summarization: Determine distribution of the edge costs, label frequencies, node degrees
- Consider graphs as sets of nodes and edges

  => 2 Views: Multi-Instance Object of nodes, Multi-Instance object of edges



DATABASE SYSTEMS GROUP

Topological Deskriptors

But: Graph Topology is still insufficiently represented

⇒ Topological Descriptors

e.g. properties of ways, paths, subgraphs,..

⇒ Topological descriptors decompose a graph into sets of simpler topological objects.

Example: Wiener Index

Let G=(V,E) be a graph. Then, the Wiener Index *W(G)* is defined as:

$W(G) = \sum_{v_i \in G} \sum_{v_j \in G} d(v_i, v_j)$ where $d(v_i, v_j)$ is the cost of the shortest path between v_i and v_j in *G*.

Remark: IF $G \cong G' \Rightarrow$ **W(G) = W(G‘).**
However, W(G) = W(G‘) does not imply $G \cong G'$

DATABASE SYSTEMS GROUP

Similarity Measures based on Topological Descriptors

LMU

Idea: Use topological descriptors and graph decompositions to define graph similarity measures.

Approaches:

- Derive feature spaces based on topological descriptors

- Integrate topological decomposition into similarity measures

DATABASE SYSTEMS GROUP

R-Convolution Kernels

LMU

- Generalization of convolution kernels for sets
- General framework for kernel functions for complex objects
- Allows the proving the kernel properties
- Let $o \in O$ be a composed object , $D(o) = (x_1, .. x_n)$ (=decomposition of o), where each component x_i is in the feature space F_i .
- $R: F_1 \times .. \times F_n \rightarrow \{True, False\}$ describes whether $(x_1, .. x_n)$ is a valid decomposition of o.
- $R^{-1}(o):=\{x|R(o,(x1,..,xn)=True\}$ is the set of all valid decompositions
- The R-convolution kernel of kernel function $K_1,..K_D$ where $K_i:X_i \times X_i \rightarrow IR$ is defined as:

$$K(x,x') = K_1 \cdot \ldots \cdot K_n(x,x') = \sum_{x \in R^{-1}(x), x' \in R^{-1}(x')} \prod_{i=1}^{n} K_i(x_i, x'_i)$$

Remark:

- All pairs of valid object decompositions are compared and summed up.
- For all elements of the objects the comparison between the corresponding parts are multiplied

R-Convolution Kernel

Simple Example: Comparing Graphs as Multi-Instance Objects

Two Labeled Graphs *G=(V,E) and G'=(V',E')* where *L: V→ IR[d]*.

Decomposition of G: D*(G)=V (set of nodes)*

Kernel K: $\langle x,y \rangle$ linear kernel of the node labels L(v).

$$K(G,G') = \sum_{\substack{v \in V \\ v' \in V'}} \prod_{i=1}^{1} \langle L(v), L(v') \rangle = \sum_{\substack{v \in V \\ v' \in V'}} \langle L(v), L(v') \rangle$$

Remark:

Multi-Instance Objects can be considered as graphs without edges.

R-Convolution Kernel and Topological Descriptors

- Let *S(G)* be the set of all subgraphs of *G*.
- **All Subgraph Kernel** fpr *G* and *G'*:

$$K_{Subgraph}(G,G') = \sum_{g \in S(G)} \sum_{g' \in S(G)} K_{isomorphism}(g,g')$$

where

$$K_{isomorphism}(g,g') = \begin{cases} 1 & falls \quad g \cong g' \\ 0 & sonst \end{cases}$$

Remark:

- compares all subgraphs for isomorphism
- NP-hard kernel due to subgraph-isomorphism

Product Graphs and Way-Based Kernels

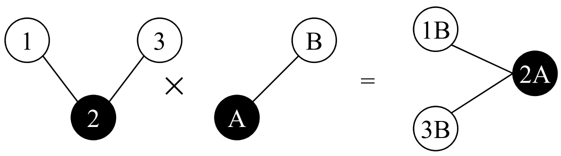
Idea: Find common ways *G* and *G'* to define graph similarity.

Product graphs simplify the search for common subgraphs.

Product Graph:

$G_\times = G \times G'$ for $G=(V,E,L)$ and $G=(V',E',L')$ is defined as:

$$V_\times = \{(v_i, v'_j) : v_i \in V \wedge v'_j \in V' \wedge L(v_i) = L(v'_j)\}$$
$$E_\times = \{((v_i, v'_j),(v_k, v'_l)) \in V \times V' : (v_i, v_k) \in E \wedge (v'_j, v'_l) \in E' \wedge L(v_i, v_k) = L(v'_j, v')\}$$



Random Walk Kernel

Idea: Count the number of common ways in both graphs.
(each way is given by its label sequence)

- **Computation**:
  Enumerate all ways in both graphs and count.
- **Problem**: Ways might infinitely extendable

- **Solution:** computation using the product graph

$$K_\times(G,G') = \sum_{i,j=1}^{|V_\times|}\left[\sum_{n=0}^{\infty} \lambda^n A_\times^n\right]_{ij} = \sum_{i,j=1}^{|V_\times|}\left[(I - \lambda A_\times)^{-1}\right]_{i,j}$$

  - Remark: parameter *0< λ < 1 is required for the convergence of the row*
  - if convergent random walk kernels are positive definite
  - *I* is the one matrix were $x_{i,i} = 1$ and $x_{i,j} = 0$ $i \neq j$

Random Walk Kernel

time complexity:

- let n = max(|V|,|V'|) for 2 graphs G and G'
- computation of the product graph:
  – compare all pairs of edges: n^2 potential edges
  – time complexity: $O(n^4)$
- Inversion of the adjacency matrix is cubic:
  – Invert a $n^2 \times n^2$ Matrix : $O(n^6)$
- Complexity of the complete kernel is : $O(n^6)$
- Later on it was shown that random walk kernels can be computed in $O(n^3)$ [Vishwanathan et al. 2006])

Problems with Random Walks

„Tottering"

- Walk-Kernel allow to visit the same nodes again and again
- multiple visits => evenm long walks can be very local
- the graph of the graph is insufficiently described

Solutions:

- Introduce additional labels
  $\Rightarrow$ less matching nodes
- disallow direct cycles.
  $\Rightarrow$ no real improvement
  $\Rightarrow$ Tottering can happend over multiple nodes

Shortest Path Kernel

Idea: Decompose graphs into the set of shortest paths.

⇒ no Tottering

⇒ less components

Method:

- compute all shortest paths between G and G'
- Compare the sets of paths based on the convolution kernel
  => sum of pairwise path similarities
- Needs some kernel to compare the paths

Shortest Path Kernel

Computation of all shortest paths:

- Use an all-pair shortest path algorithmn
  (Floyd-Warshal Algorithmus: *O(n³)*)
- Result is the distance matrix D:

$$M_{ShortestPath}(G)_{ij} = \begin{cases} d_{i,j} & if \quad v_i \text{ reachable from } v_j \\ \infty & else \end{cases}$$

- the set SD(G) of shortest paths describes the graph G
- Comparision by convolution kernel:

$$K_{shortestPath}(G, G') = \sum_{s_1 \in SD(G)} \sum_{s_2 \in SD(G')} k(s_1, s_2)$$

- Complexity is $O(n^4)$

Kernels and Distances

Something algorithms require distance measures:

1. Each kernel (scalar product) induces a metric:

$$D(G,G') = \sqrt{K(G,G) + K(G,G') - 2 \cdot K(G,G')}$$

2. Multiple distance measures are based on the same ideas:
   Example: employ SMD, Hausdorff or MMD on sets of shortest paths.

Conclusions

- Modelling objects as graphs is very general
- The complexity of graphs limits their usability
- topological descriptors are a trade-off between performance and exact comparisons
- Topological descriptors decompose a graph into simpler components
- Decomposition usually loses information

Frequent Subgraph Mining

Idea: Find all frequent subgraphs in a database of graphs

Applications:

- Common subgraphs can be used as topological descriptors
- Find typical subnetworks (cliques) in social networks
- Graph compression: Substitute frequent subgraphs by single nodes => reduces the size of the graphs
- Derive rules about social interaction
- find common motifs in protein interaction networks

Approaches to Frequent Subgraph Mining

- *Frequent Subgraph Mining is similar to Itemset mining*
  - Exploit monotonicity between subgraphs and super graphs
    => *k* Itemset *I* can only be frequent if all *k-1* Itemsets in *I* are frequent
    analogue: Subgraph *G* containing k nodes can only be frequent if all subgraphs of *G* containing *k-1* nodes are frequent
  - Generate candidates of size *k be combining pairs of frequent* subgraphs of size *k-1*.
- *Direct extension of frequent patterns*
  - Find all subgraph containing *k* nodes and extend them by an additional node => candiate for frequent subgraphs containing *k+1* nodes

Basic Problems

Subgraph-Isomorphism yields large problems

- Detecting occurrences of a candidate is very expensive
- Support Computation must consider all isomorphic subgraphs
- Candidates should only be generated once

⇒ All algorithms define a normal form for each isomorphic clas

⇒ Transforming a graph into the normal form is expensive

⇒ comparing normal forms is cheap

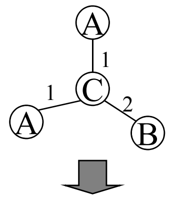
Algorithms for Frequent Subgraph Mining

FSG [Kuramochi, Karypis 2001]

for labeled and undirected graphs.

Idea: Apply apriori algorithm to subgraph mining.

- graphs are given as adjacency lists
- Isomorphic graphs can be considered as permutations of the adjacency lists

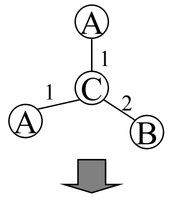


	1	2	3	4
	A	**A**	**C**	**B**
1	0	0	1	0
2	0	0	1	0
3	1	1	0	2
4	0	0	2	0

⇒ Canoncial Labelling

unique ordering to induce a normal form for each isomorphic class

Canonical Labeling



	1	2	3	4
	A	**A**	**C**	**B**
1	0	0	1	0
2	0	0	1	0
3	1	1	0	2
4	0	0	2	0

	1	2	3	4
	A	**A**	**B**	**C**
1	0	0	0	1
2	0	0	0	1
3	0	0	0	2
4	1	1	2	0

000112

- order the columns w.r.t. node degree
- generate all permutation for nodes having the same degree
- serialize the upper triangular matrix
- select the lexicographically smallest string

⇒ unique identifier for each isomorphic class

⇒ requires only permutation within a subset of the nodes

⇒ subgraph occurences and candidate testing can be based on the canonical labeling

FSG Algorithmus(1)


```
Vector<GraphSet> fsg(GraphSet D, double δ)
    GraphSet F1 = Set of frequent subgraphs having one edge
    GraphSet F2 = Set of frequent subgraphs having two edges
    int k=3
    Vector<GraphSet> frequentSubgraphs;
    frequentSubgraphs.add(F1)
    frequentSubgraphs.add(F2)
    while(frequentSubgraphs.getLastElement()!= {})
        Graphmenge Ck= fsg-gen(frequentSubgraphs.getLastElement());
        foreach Graph c ∈ Ck
                int anzahl_c_in_D =0;
                foreach Graph d ∈ D
                        if(d.includes(c))
                          anzahl_c_in_D ++;
                if(anzahl_c_in_D < δ*|D|)
                  ck.remove(c);
        frequentSubgraphs.add(Ck);
    return frequentSubgraphs;
```

FSG Algorithmus(2) (Candidate generation)

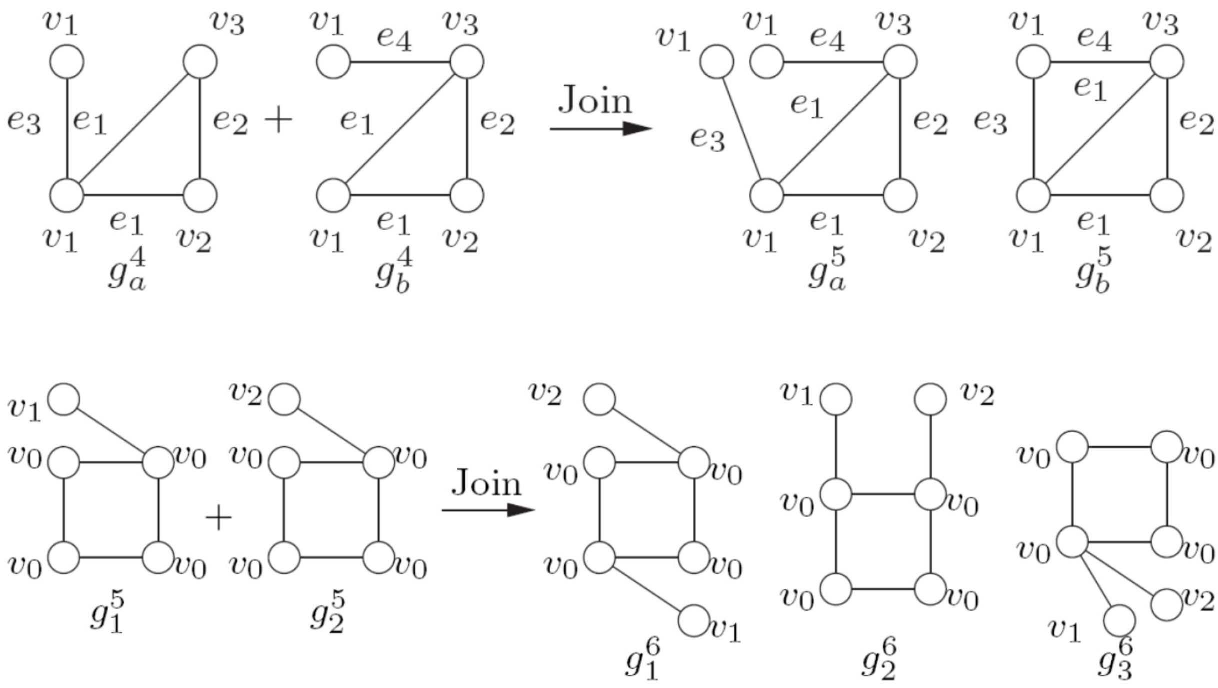

```
GraphSet fsg-gen(F^k)
    GraphSet Ck+1={};
    foreach Graph f1k ∈ F^k
          foreach Graph f2k ∈ F^k
            if(f1k.canonicalLabel <= f2k.canonicalLabel)
               foreach Edge e ∈ f1k
                     Graph f1k-1=f1k.remove(e);
                     if(f1k-1.isconnected && f2k.includes(f1k-1))
                       GraphSet Tk+1 =join(f1k, f2k)
                     forech Graph tk+1 ∈ Tk+1
                        boolean all_tk_frequent = true;
                       foreach Edge ed ∈tk+1
                           Graph tk = tk+1.remove(ed);
                            if(tk.isConnected && tk ∉ FK)
                                   all_tk_frequent = false;
                                   break;
                           if(all_tk_frequent)
                                   Ck+1.add(tk+1);
return Ck+1
```


Complexity of FSG

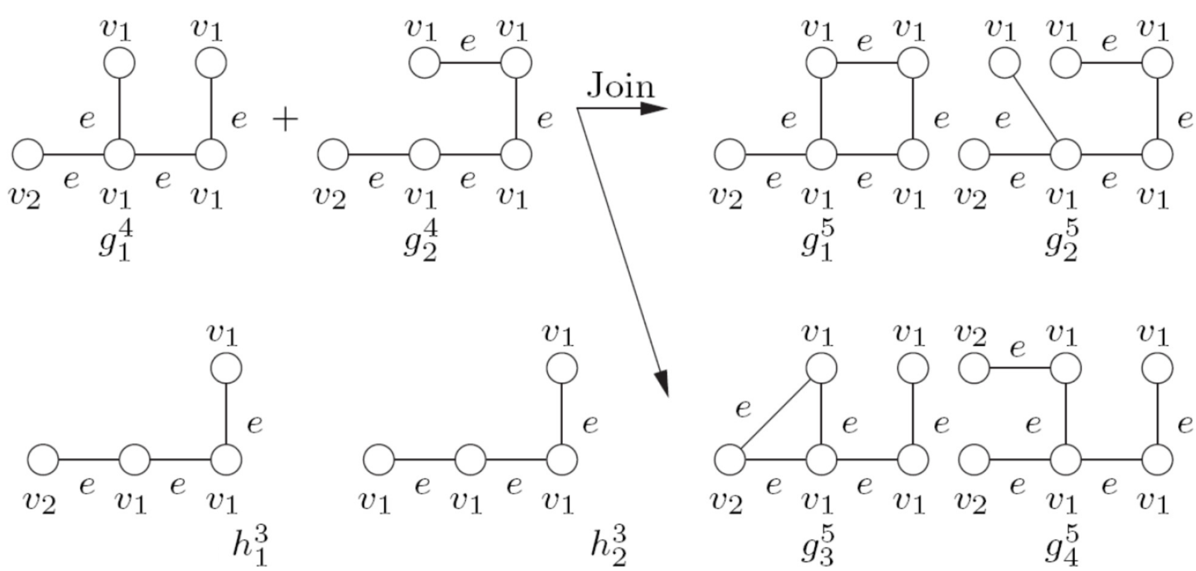
Complex parts of the algorithms:

1. Subgraph Isomorphism Testing (g.includes(s))
   - necessary when scanning the database
   - necessary during candidate generation:
     determine common *k-1* subgraph
2. Join two graph based on **k-1 s**ubgraphs
   ⇒ results in a set of candidates
   ⇒ all of the results must be tested for being real candidates

Possible Candidates(1)



Possible Candidates(2)



GSpan

Idea:

- candidate generation extend a single frequent subgraph by one edge
- desribe subgraphs by a depth first traversal (mininal DFS code)
- generate unique candidates by „right-most-only growth“

Aim:

- Avoid the generation of duplicate candidates
- Avoid isomorphism testing

Concepts:

- DFS lexivographical order
- minimal DFS code (canonical description of general subgraphs)

Pattern Growth

Naive Algorithms:

S : set of frequent graphs;
g : frequent subgraph,
DB: database
MinSup: minimal support for a subgraph in order to be frequent
S:={}
GrowPatterns(g,DB, S)

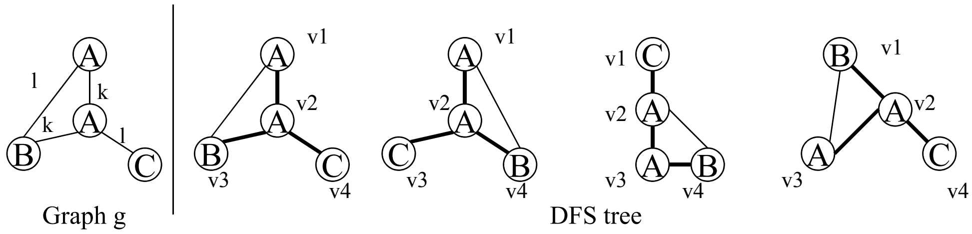
```
Function GrowPatterns(g,DB,S)
    if g ∈ S then return;
    else S.insert(g)
    EdgeSet E = findAdjacentEdges(DB,g MinSup); // find all edges in DB for extending g
    for each frequent e ∈ E DO   // only consider edges having mor edges than MinSup
      g‘ = extend(g,e)
      GrowPatterns(g‘,DB,S)
    end for
end function
```

Remark:

Finding all extensions is rather expensive and requires an isomorphism test for g ∈ S
Classen os isomorphic subgraphs should be found only once in findAdjacentEdges

DFS Codes

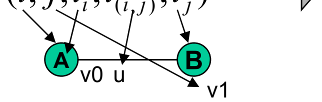
- canonical description of subgraphs belonging to one isomorphic class
- sequence of edges along a depth first traversal
  (***D***epth ***F***irst ***S***earch Tree)



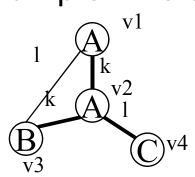
- Forward Edges: extend tree by one node
  backward edges: connect already visited nodes
- a DFS tree implies an order of the visited edges *G* (DFS-Code)
- Forward edges are ordered after visiting the start node
- Backward edges are odered corresponding to the order of the target nodes

DFS-Lexikographical Order

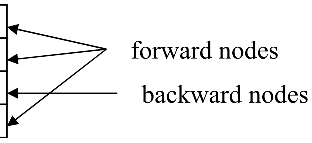
- a graph can be described as set of all DFS trees
- the DFS tree is uniquely described by the DFS-Code (sequence of edges)
- Description of an edge: $(i, j, l_i, l_{(i,j)}, l_j)$



Example: DFS code



1,2,A,k,A
2,3,A,k,B
3,1,B,l,A
2,1,A,l,C



- DFS lexicographical order: compare multiple DFS codes
- Lexicographical comparison between the codes
- edge comparison: start index, target index, start label, edge label, target label.
- Mininal DFS-Code (Min DFS-Code) w.r.t. DFS lexicographical order is unique for all graphs in the isomorphic class
  => 2 graphs *G,G´* have the same min. DFS code ⇔ *G* is isomorphic to *G´*

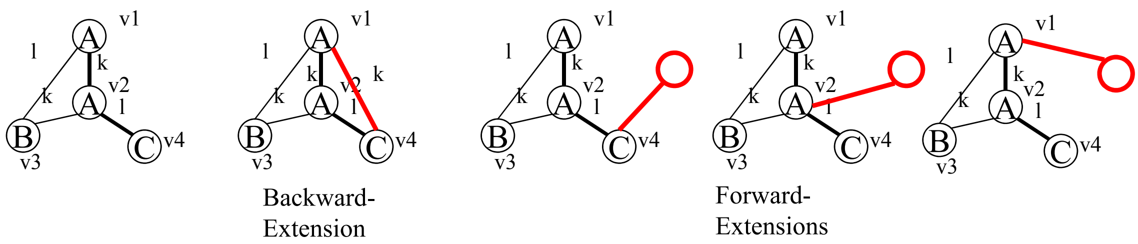
Right-Most-Only Extension

Idea: Avoid multiple generation of the same candidate

- ***Right-Most-Only Extension: only extension along the right most path are allowed***.
- **DFS-Tree**:
  - *Backward-Extension*

    connect nodes on the most right path
  - *Forward Extension*

    extend the graph beginning on the most right path



GSpan

Pattern Growth Algorithmus with right-most-only Extensions

GSpan

S :Set of frequent graphs;
s : a DFS Code
min_dfs(s): Mininmal DFS-Code of S.
DB: Graph database
MinSup: minimal support for frrequent Subgraph
S:={}
GSpan(s,DB, S)

```
Function GrowPatterns(g,DB,S)
    if s ≠ min_dfs(s) then return;
    else S.insert(s)
    C := {}
    EdgeSet E = findRightMostExtensions(DB,s, MinSup); // find all valid extensions of the minimal DFS tree
    C = extend(s,E);
    C.sortInLexDFSOrder;
    for each frequent s ∈ C DO
        GSpan(s,DB,S)
    end for
end function
```

Frequent Subgraph Mining

Frequent subgraph mining is similar to frequent itemset mining

But:

- set of isomorphic graphs is larger than the set of itemset permuations ⇒ Isomorphism testing is more complex than comparing Itemsets
- Finding canonical labeling is more difficult
- set of possible extension is far larger ⇒ candidate generation is more complex

- **FSG**: Apriori-based method with pairwise candidate geneation
- **GSpan**: Pattern-growth approach for general graphs