Knowledge Discovery in Databases II

Winter Term 2015/2016

Lecture 13 & 14:
Variety: Linked data

Lectures : Dr Eirini Ntoutsi, PD Dr Matthias Schubert
Tutorials: PD Dr Matthias Schubert

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

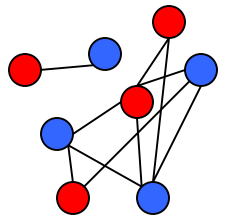
Chapter Overview

Graphs everywhere

- Molecule structures
- Protein interaction networks
- Social networks
- WWW
- Spatial networks
- Sensor networks

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.



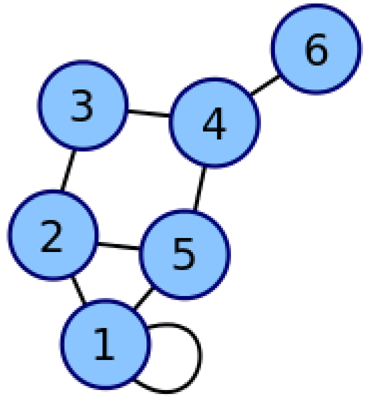
- Usually: vertices = objects, edges =relationships between objects
- Graphs provide a lot of flexibility for data modeling as one can define what are the *objects* (nodes) and the *relationships* (edges) between the objects.
- From objects to graphs
  - Objects → vertices
  - Object properties → vertex labels
  - Relation between two objects → edge
  - Type of relation → edge label

An introduction to graphs

- A graph is representable as a square matrix (Adjacency Matrix)
  - Rows/ columns correspond to the objects
  - Entries correspond to the edges between the corresponding objects
- Typically, the adjacency matrix of a graph G=(V,E) is defined as:

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

- In general, different "mappings" from edges to entry values are possible:
  - An example (source: Wikipedia)



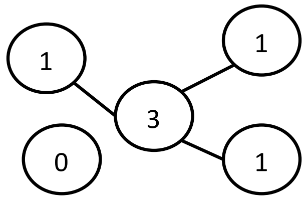
each edge adds 1 to the appropriate cell in the matrix, and each loop adds 2

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

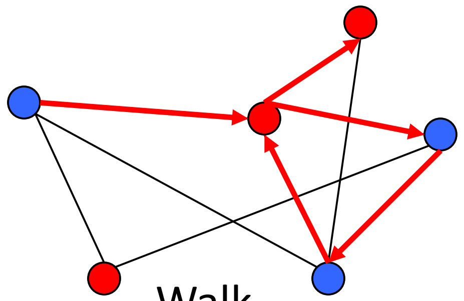
An introduction to graphs

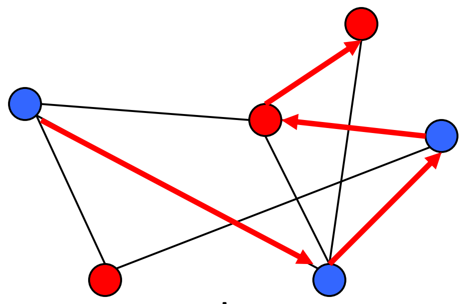
- **Node degree:** The degree of a node v_i in G=(V,E) denoted as $d_G(v_i)$ is number of adjacent edges:

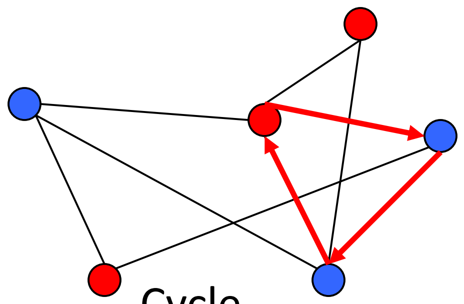
$$d_G(v_i) = \left| \left\{ v_j \middle| (v_i, v_j) \in E \right\} \right|$$



- **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 $i \neq j$. (i.e., 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.

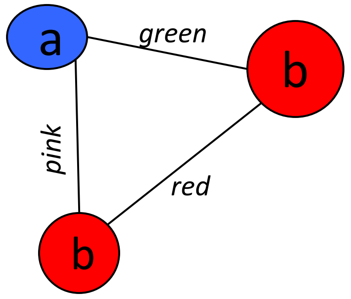

Walk


Path


Cycle

An introduction to graphs

- **Directed** vs **undirected** graphs:
  - directed graph: $(v_k,v_l) \neq (v_l,v_k)$, adjacency matrix is not symmetric
- **Labeled** vs **unlabeled** graphs
  - labeled graphs: both nodes and edges.
    - 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 $l_e \in F_E$.
  - Labels can be arbitrary types of information
  - In most cases, labels are symbols from a given alphabet



- **Subgraph**: Let $G =(V,E)$ be a graph then $G'=(V',E')$ is a subgraph of G,

  if $V' \subseteq V$ and $E' \subseteq (V' \times V' \cap E)$.

Chapter Overview

Comparing Graphs

Input: Two graphs G and G‘ from the graph space $\mathbb{G}$.

Output: A mapping *s:* $\mathbb{G} \times \mathbb{G} \rightarrow$ *IR* computing the similarity of G and G‘.

Different comparison approaches:

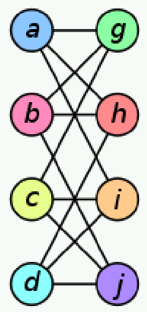
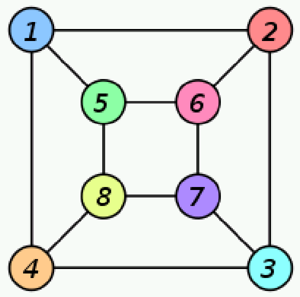
- **Isomorphism**: 2 graphs are equal if there exists a bijection between nodes inducing a bijection of edges.
- **Edit-distance**: Similarity is computing by counting the minimal amount of operations transforming one graph into the other.
- **Topological descriptors**: Two graphs are similar if the have similar values w.r.t. topological properties like 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 isomorphic ($G \simeq G'$) if there exists a bijection *f: V*$\rightarrow$*V‘* such that *(v,v‘)*$\in$*E* $\Leftrightarrow$ *(f(v),f(v‘))*$\in$ *E‘* for all node pairs *v,v‘*$\in$ *V*.

  - i.e., there is a 1-1 correspondence between the nodes that preserves the edge structure

Graph G	Graph H	An isomorphism between G and H
		*f*(*a*) = 1
		f(*b*) = 6
		f(*c*) = 8
		f(*d*) = 3
		f(*g*) = 5
		f(*h*) = 2
		f(*i*) = 4
		f(*j*) = 7

Source: https://en.wikipedia.org/wiki/Graph_isomorphism

- **Subgraph-Isomorphism:**

Let *G=(V,E)* and *G'=(V',E') be two* graphs. An injective function *f:V*$\rightarrow$*V'* is a subgraph isomorphism if there exists a subgraph *G''* of *G'* such that *f* is a graph isomorphism between *G* and *G''*.

Graph Isomorphism

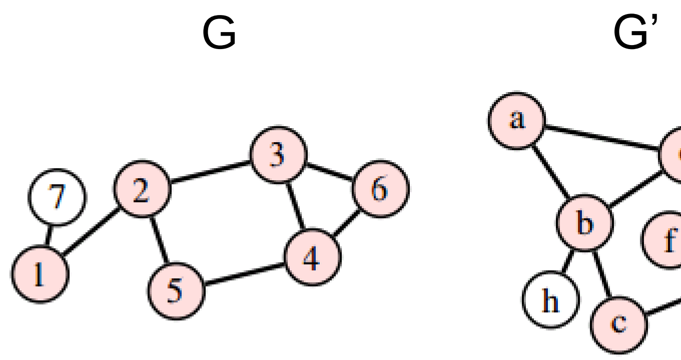
- **Common Subgraph**

Let *G, G'* be two graphs. Let *g* a subgraph of *G* and let *g'* subgraph of *G'*. If there exists a graph isomorphism between *g* and *g'*, then both *g* and *g'* are called a common subgraph of *G* and *G'*.

- **Maximum Common Subgraph**

Let *G,* G' be two graphs and let *g, g'* are a common subgraph of *G, G'*. If there is no other subgraph of *G, G'* that has more nodes than *g* and *g'*, then *g* and *g'* are called maximum common subgraph *mcs(G, G')*.

  - Represents the maximal part of both graphs that is identical



Bijection

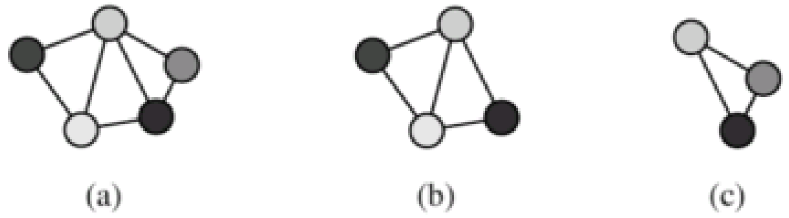
$a \leftrightarrow 6$
$b \leftrightarrow 4$
$c \leftrightarrow 5$
$d \leftrightarrow 2$
$e \leftrightarrow 3$
$f \leftrightarrow 1$

Source: http://www.lsis.org/tuples/workshop/wscp_bgbtp_1.pdf

Graph Isomorphism

- **Minimum Common Supergraph**

Let *G, G‘* be two graphs. A graph S is a minimum common supergraph *MCS(G,G‘)* if *G* and *G‘* are common subgraphs of *S* and there is no other graph containing *G* and *G’* having less nodes.



Graph (a) is a minimum common supergraph of graph (b) and (c)
Source: [AggWan10]

Similarity based on graph isomorphism

Notation: *mcs*: maximum common subgraph, MCS: minimum common supergraph

- **Distance Measure 1**: Relative size of the maximum common subgraph

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

  - The larger the mcs, the larger the similarity
  - Value range: [0,1]
    - 0, if G, G’ are isomorphic
    - 1, if G, G’ have no part in common

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

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

  - mcs provides a lower bound on the similarity, MCS an upper bound
  - 0 if G and G’ are isomorphic
  - As G and G’ become more dissimilar,|mcs| decreases and |MCS| increases
  - The normalized version of d_2:

$$d_{2'}(G,G') = 1 - \frac{|mcs(G,G')|}{|MCS(G,G')|}$$

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

Edit distances for graphs

Idea: Distance = minimum cost to transform *G* to *G'*.

- Differences are removed by performing different graph operations:
  - *Delete*, *Add*, *Relabel*, for both nodes and edges
- Costs for each operation might vary depending on the labels
- Metric properties rely on the employed costs

- **Graph Edit Distance:** The shortest or the least cost sequence of elementary graph edit operations that transform one graph into the other

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

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

Problem:

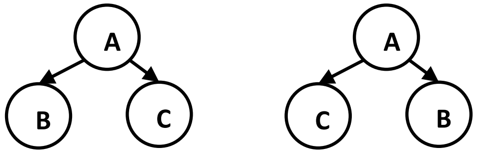
- Still has to solve (sub)graph isomorphism => computation is very expensive.
- Choosing cost function for different operations is difficult

Edit distances for graphs

Performance:

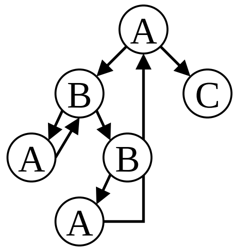
- in general cases the complexity cannot be descreased
- for special cases faster methods are possible
- e.g. trees

  => unique serialisations are generall possible (order of subtrees)



  => Edit-distance for strings is in *O(n²)*

  => Problem: Insertion costs have to be 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]]

Isomorphism- and edit distance-based overview

- 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 complete)
- Limiting the problem to certain types of topologies can reduce the complexity

Topological Descriptors and Graph Kernels

Idea: Since the aforementioned approaches are too expensive

- Map each graph to a feature vector
- Compare these vectors

Pros: reuse known and efficient tools for feature vectors

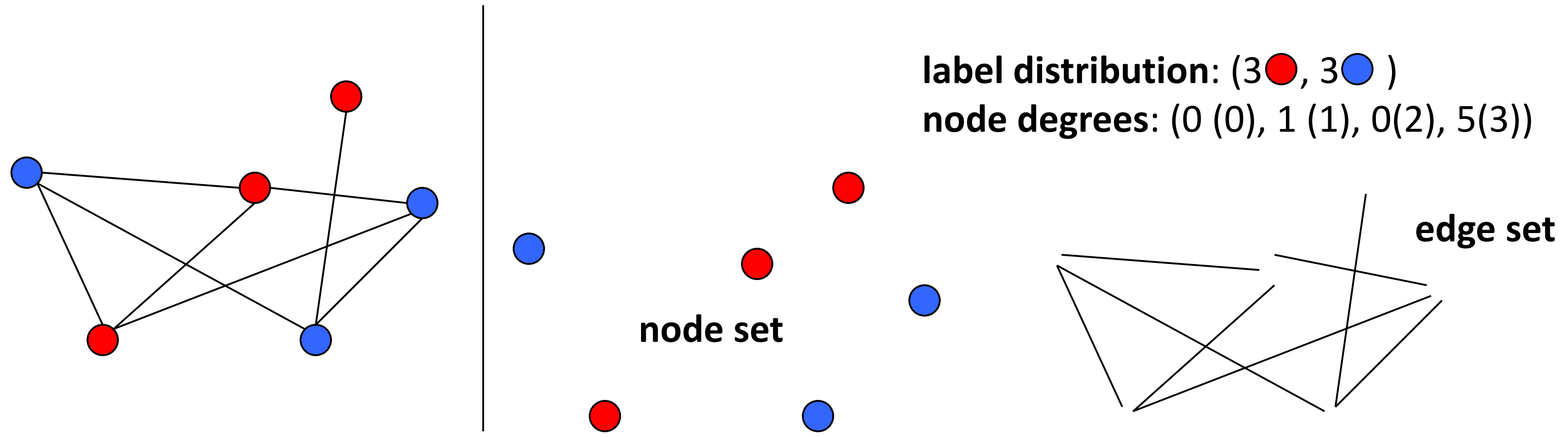
Cons: Efficiency comes at a price: feature vector transformation leads to loss of topological properties (or includes subgraph isomorphism as one step)

Topological Descriptors and Graph Kernels

Basic descriptors:

- Graph summarization: Distribution of 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


label distribution: (3🔴, 3🔵)
node degrees: (0 (0), 1 (1), 0(2), 5(3))
edge set
node set

Topological Descriptors

But: Graph topology is still insufficiently represented

$\Rightarrow$ Topological descriptors

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

$\Rightarrow$ Topological descriptors decompose a graph into sets of simpler topological objects.

Example: **Wiener Index**

The Wiener Index *W(G)* for a graph G=(V,E) is defined as the sum of distances between all distinct pairs of nodes

$$W(G) = \sum_{v_i \in G} \sum_{v_j \in G} d(v_i, v_j)$$

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 $G \cong G'$

Similarity Measures based on Topological Descriptors

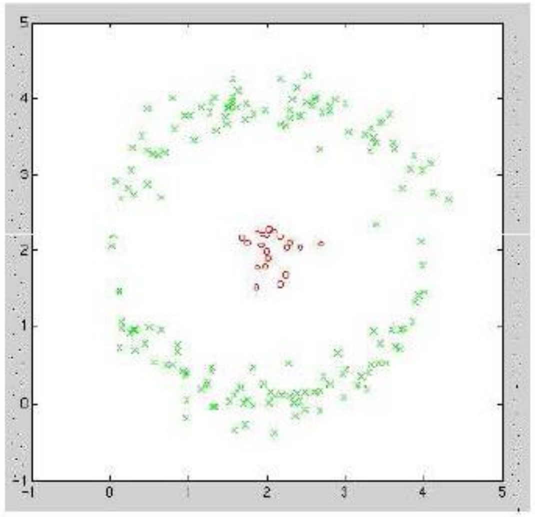
Idea: Use topological descriptors and graph decompositions to define graph similarity measures.

Approaches:

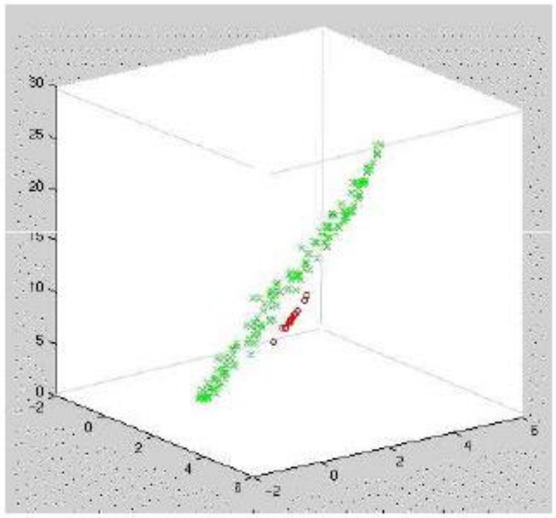
- Derive feature spaces based on topological descriptors that are computable in polynomial time
- Integrate topological decomposition into similarity measures
- Use graph kernels

What is a kernel?

- A kernel is a transformation/ mapping ϕ of the input data x, x’ into a feature space H.
- Measure the similarity in H as <ϕ(x), ϕ(x‘)>
- Kernel trick: Compute inner product in H as kernel in input space: K(x, x‘) = <ϕ(x), ϕ(x‘)>



$$\Phi : \mathbf{R}^2 \to \mathbf{R}^3$$
$$(x_1, x_2) \mapsto (x_1, x_2, x_1^2 + x_2^2)$$



These classes are linearly inseparable in the input space

We can make the problem linearly separable by a simple mapping

R-Convolution Kernels

- Compare decompositions of structured objects
- Let X be a set of composite objects (e.g., cars), and $\bar{X}_1, \dots, \bar{X}_D$ be sets of parts (e.g., wheels, brakes, etc.). All sets are assumed countable.
- Let R denote the relation "being part of" (i.e., if the decomposition is valid):

$$R(\bar{x}_1, \dots, \bar{x}_D, x) = 1, \text{ iff } \bar{x}_1, \dots, \bar{x}_D \text{ are parts of } x$$

- The inverse relation R^{-1} is defined as: $R^{-1}(x) = \{\bar{\mathbf{x}} : R(\bar{\mathbf{x}}, x) = 1\}$
- In other words, $R^{-1}(x)$ contains valid decompositions for x.

Let $x, y \in X$ and $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ be the corresponding sets of parts. Let $K_d(\bar{x}_d, \bar{y}_d)$ be a kernel between the d-th parts of x and y $(1 \leq d \leq D)$. Then the convolution kernel between x and y is defined as:

$$K(x, y) = \sum_{\bar{\mathbf{x}} \in R^{-1}(x)} \sum_{\bar{\mathbf{y}} \in R^{-1}(y)} \prod_{d=1}^{D} K_d(x_d, y_d)$$

Remarks:

- 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

Input: Two labeled graphs *G=(V,E) and G‘=(V‘,E‘)*

Node labels *L: V* $\rightarrow$ *IR*^{d}.

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

Linear Kernel of the node labels *K: K(v,v‘)=*$\langle$*L(v),L(v‘)*$\rangle$

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

Remark:

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

DATABASE SYSTEMS GROUP

R-Convolution Kernel and Topological Descriptors

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

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

where

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

Remarks:

- Compares all subgraphs for isomorphism
- NP-complete kernel due to subgraph-isomorphism

Product Graphs and Walk-based Kernels

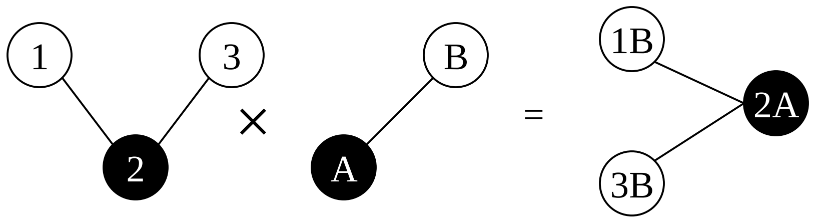
Idea: Find matching walks in *G* and *G‘* to define graph similarity.

Graph products simplify the search for common subgraphs.

Direct Graph Product :

$G_\times = G \times G'$ 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:** Given two graphs G and G', perform random walks on both and count the number of matching paths.
  - Match if: they have the same length and the label sequences are the same.
- **Solution:** computation using the direct product graph
  - It has been proven that: A random walk on the direct product graph G_x is equivalent to performing a simultaneous random walks on *G* and *G'*.
  - Construct direct product graph of G and G', $G_x=(V_x,E_x)$
  - Count walks in this product graph
  - It holds that: Walks of length *k* can be computed by looking at the *k*-th power of the adjacency matrix , i.e., A^k_x

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

- A_x: the adjacency matrix of G_x
  - Remark: parameter $0 < \lambda < 1$ is required for the convergence
  - if convergent random walk kernels are positive definite

Problems with Random Walks

- **Complexity**
  - Complexity of the complete kernel is: O(n^6)
- **Tottering**
  - Walks allow for repetition of nodes
  - A walk can visit the same cycle of nodes again and again
  - Kernel measures similarity in terms of common walks
  - Hence a small structural similarity can cause a huge kernel value

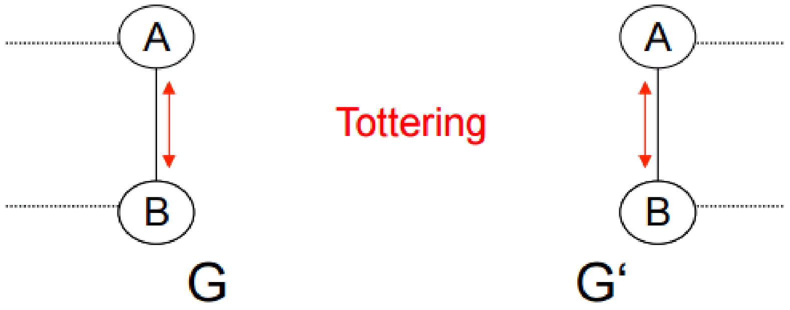
Solutions to tottering:

- Introduce additional labels

  $\Rightarrow$ less matching nodes
- disallow direct cycles.

  $\Rightarrow$ no real improvement

  $\Rightarrow$ Tottering can happen over multiple nodes



Shortest Path Kernel

Idea: Decompose graphs into sets of shortest paths.

$\Rightarrow$ no tottering

$\Rightarrow$ 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 Algorithm: *O(n³)*)

- Result is the distance matrix D:

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

- Comparision by convolution kernel:

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

  – the set SD(G) of shortest paths describes the graph G

- Complexity is $O(n^4)$

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

Chapter Overview

Frequent Subgraph Mining

The problem: Find all frequent subgraphs in a database of graphs

Applications:

- Common subgraphs can be used as topological descriptors
- Find typical subnetworks 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

Problem definition 1/2

The problem:

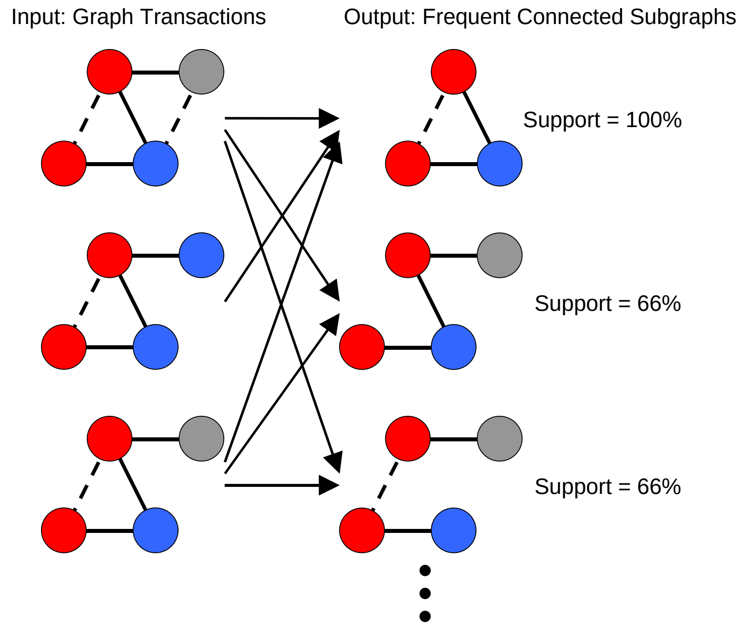
Given a graph dataset *D*, find all frequent subgraph *g* w.r.t. a frequency threshold.

- To reduced the complexity, only frequent connected subgraphs are considered
- A subgraph is connected if there are paths between every pair of vertices.

- **Input:**
  - A dataset of transactions D.
  - Each transaction is a simple graph (undirected, no loops)
  - Both nodes and edges have labels.
  - A minSupport threshold σ

- **Output:**

All connected undirected subgraphs that occur in at least $\sigma|D|$ transactions.


Input: Graph Transactions
Output: Frequent Connected Subgraphs
Support = 100%
Support = 66%
Support = 66%

Problem definition 2/2

The problem (definition using the concept of isomorphism)**:**

Given a graph dataset *GS* and a minSupport threshold *σ*, let *σ(g,GS)* be the occurrence frequency of *g* in *GS*:

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

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

Frequent Subgraph Mining is to find every graph g in GS such that *σ(g,GS)≥σ*.

Approaches to Frequent Subgraph Mining

Frequent Subgraph Mining is an extension of Frequent Itemset Mining (FIM)

- Exploit monotonicity between subgraphs and supergraphs
  - A *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*.
  - analogue: Find all subgraph containing k nodes and extend them by an additional node => candidate 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 be generated only once

$\Rightarrow$ All algorithms define a normal form for each isomorphic class

$\Rightarrow$ Transforming a graph into the normal form is expensive

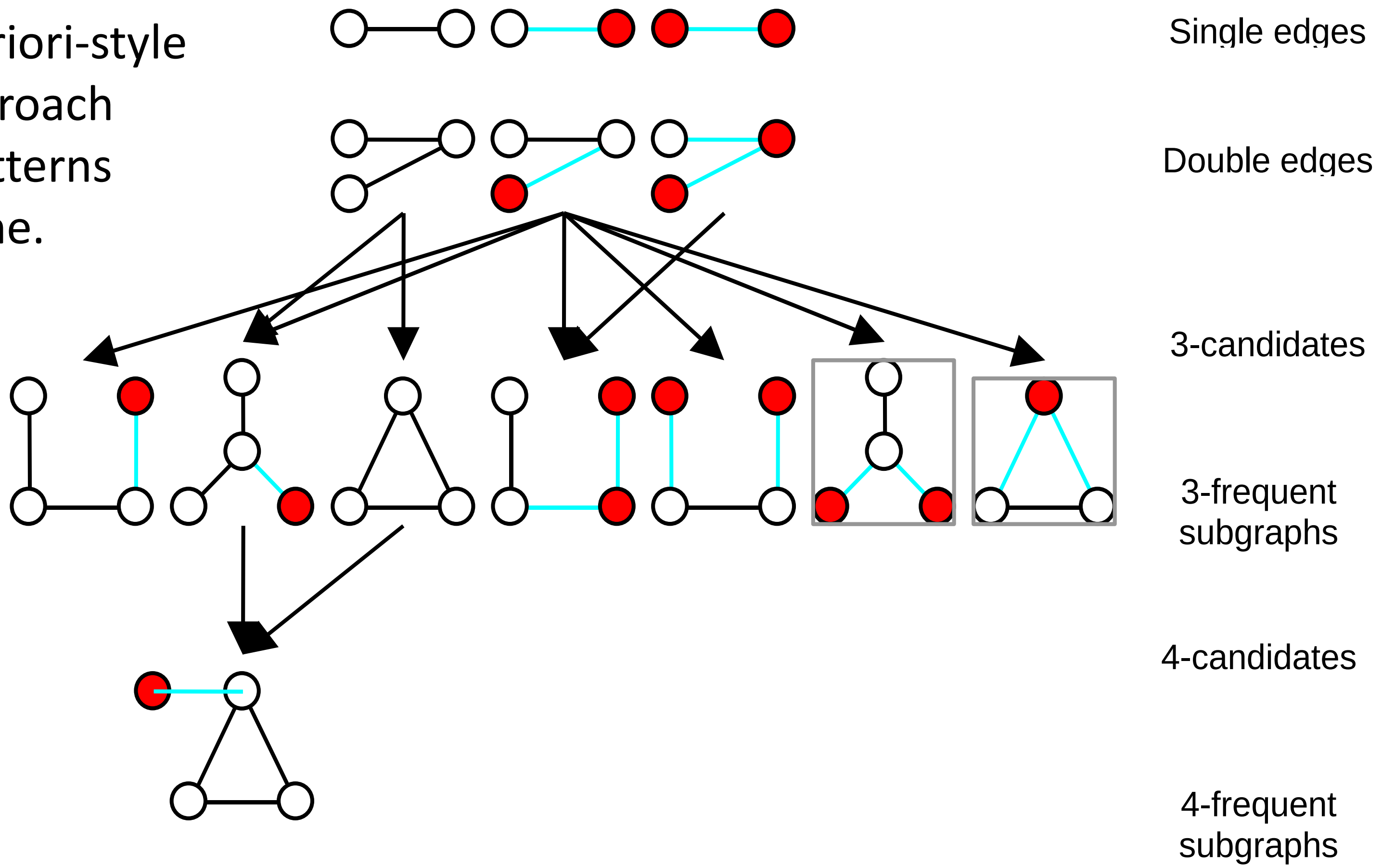
$\Rightarrow$ But, comparing normal forms is cheap

- 2 types of algorithms
  - Apriori-based approaches: FSG [KurKar01]
  - Pattern growth-based approaches: gSpan [YanHan02]

Algorithms for Frequent Subgraph Mining: FSG

- FSG (frequent subgraph) [KurKar01]
  - Edges correspond to items

– Follows an Apriori-style level-by-level approach and grows the patterns one edge-at-a-time.



FSG algorithm psedocode

```
**Algorithm fsg(GraphSet D, double σ)**
//D: dataset of transactions (graphs)
//σ: minSupport threshold
    $F^1$ ← Set of frequent subgraphs having one edge
    $F^2$ ← Set of frequent subgraphs having two edges
    k ← 3
    while($F^{k-1}$ != {})
        //Candidate generation
        $C^k$ ← fsg-gen($F^{k-1}$);
        foreach candidate $g^k \in C^k$ do
            $g^k$.count ← 0;
            foreach graph d ∈ D
                if(d.includes($g^k$)) //Inclusion check
                    $g^k$.count ← $g^k$.count+1;
        //Pruning by support count
        $F^k$←{$g^k \in C^k$| $g^k$.count≥σ|D|}
        k ←k+1
    return $F^1$, $F^2$, ..., $F^k$;
```

FSG basics

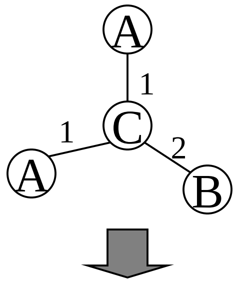
- Graph representation
  - Uses sparse graph representation to store input transactions, intermediate candidates and frequent subgraphs
  - Stored using adjacency lists
- Canonical labeling
  - In FIM, items are sorted by lexicographical order
  - Graphs can be represented in different ways depending on the order of their edges or vertices.
  - Use canonical labeling
    - A canonical label *cl(G)* is a unique code of a graph *G*
  - Canonical labeling is equivalent to finding isomorphism in graphs
    - If two graphs are isomorphic their canonical labels must be identical

Canonical Labeling for Graphs

- Idea: Try all vertex permutations to see which ordering of vertices gives the mininum adjacency matrix
  - Isomorphic graphs can be considered as permutations of the adjacency lists
- Methodology:
  - Narrow down the search through vertex invariants
    - First partition the vertices by their degrees and labels
    - Try all possible permutations within each partition
  - Serialize the upper triangular matrix
  - Select the lexicographically smallest string

⇒ requires only permutation within a subset of the nodes

⇒ unique identifier for each isomorphic class



	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

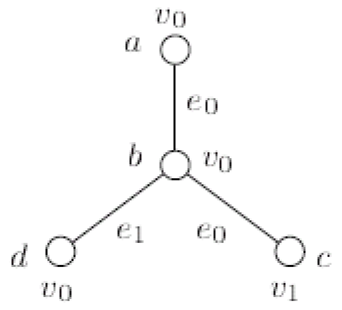
	1	2	3	4
	A	**A**	**B**	**C**
1	0	0	0	1
2	0	0	0	1
3	0	0	0	2
4	1	1	2	0

000112

DATABASE SYSTEMS GROUP

Canonical Labeling example

LMU



Graph

id	a	b	c	d
label	v_0	v_0	v_1	v_0
a	0	e_0	0	0
b	e_0	0	e_0	e_1
c	0	e_0	0	0
d	0	e_1	0	0

Adjacency matrix

id	a	c	d	b
label	v_0	v_1	v_0	v_0
partition		0		1
a	0	0	0	e_0
c	0	0	0	e_0
d	0	0	0	e_1
b	e_0	e_0	e_1	0

Vertice degree partitioning

id	d	a	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
d	0	0	0	e_1
a	0	0	0	e_0
c	0	0	0	e_0
b	e_1	e_0	e_0	0

Vertice label partitioning (v_0<v_1)

id	d	a	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
d	0	0	0	e_1
a	0	0	0	e_0
c	0	0	0	e_0
b	e_0	e_1	e_0	0

id	a	d	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
a	0	0	0	e_0
d	0	0	0	e_1
c	0	0	0	e_0
b	e_0	e_1	e_0	0

Test all possible permutations within each partition

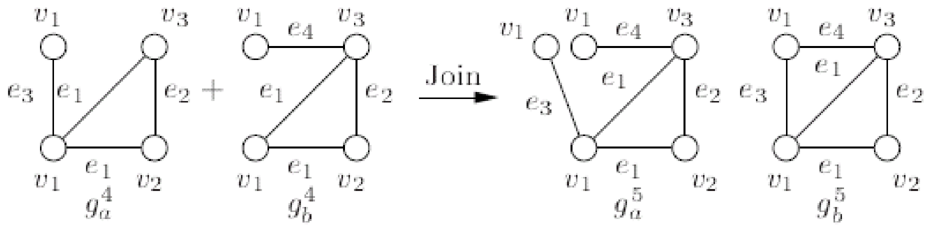
$000e_1e_0e_0$

Choose the smalles string

$000e_0e_1e_0$

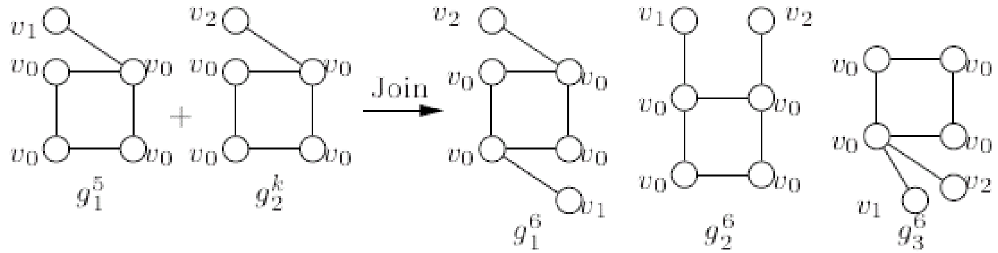
FSG: Candidate generation

- Join two frequent size-k subgraphs to get (k+1) candidates
  - Common connected (k-1) subgraph is necessary (called *core*)
  - Apriori doesn't suffer this problem due to lexicographic ordering of itemset
- Problem
  - Unlike FIM where a unique (k+1) itemset is created by joining two k-itemsets, the join of two subgraphs might led to multiple (k+1)subgraphs



v_1 v_3 e_3 e_1 e_2 v_1 e_1 v_2 g_a^4
+
v_1 e_4 v_3 e_1 e_2 v_1 e_1 v_2 g_b^4
Join
v_1 v_1 e_4 v_3 e_3 e_1 e_2 v_1 e_1 v_2 g_a^5
v_1 e_4 v_3 e_1 e_3 e_2 v_1 e_1 v_2 g_b^5

(a) By vertex labeling



(b) By multiple automorphisms of a single core

Candidate generation

Algorithm 2 fsg-gen(F^k) (Candidate Generation)

```
1:  $C^{k+1} \leftarrow \emptyset$;
2:  for each pair of $g_i^k, g_j^k \in F^k, i \leq j$ such that $\mathrm{cl}(g_i^k) \leq \mathrm{cl}(g_j^k)$ do
3:    for each edge $e \in g_i^k$ do {create a $(k-1)$-subgraph of $g_i^k$ by removing an edge $e$}
4:      $g_i^{k-1} \leftarrow g_i^k - e$
5:      if $g_i^{k-1}$ is included in $g_j^k$ then {$g_i^k$ and $g_j^k$ share the same core}
6:        $T^{k+1} \leftarrow$ fsg-join($g_i^k, g_j^k$)
7:        for each $g_j^{k+1} \in T^{k+1}$ do
8:          {test if the downward closure property holds for $g_j^{k+1}$}
9:          flag $\leftarrow$ true
10:         for each edge $f_l \in g_j^{k+1}$ do
11:           $h_l^k \leftarrow g_j^{k+1} - f_l$
12:           if $h_l^k$ is connected and $h_l^k \notin F^k$ then
13:             flag $\leftarrow$ false
14:             break
15:         if flag = true then
16:           $C^{k+1} \leftarrow C^{k+1} \cup \{g^{k+1}\}$
17: return $C^{k+1}$
```

Core identification

Join

Downward property

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
3. Canonical labeling
   ⇒ Used to efficiently detect subgraph occurences and for candidate testing

Algorithms for Frequent Subgraph Mining: gSpan

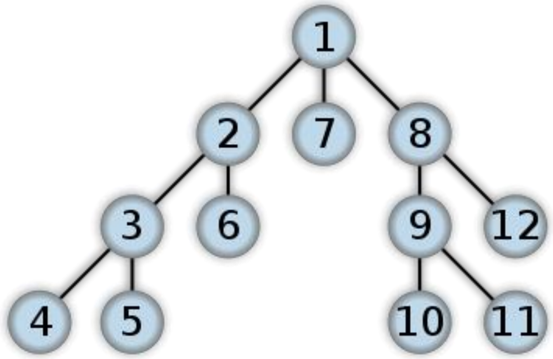
- Weakness of Apriori-based approach FSG
  - The generation of size (*k*+1) subgraph candidates from size *k* frequent subgraphs is too complicated and complex.
  - (k-1) core identification, joining, pruning false positives are expensive due to isomorphism
- gSpan: <u>G</u>raph-Based <u>S</u>ubstructure <u>Pa</u>tter<u>n</u> Mining [YanHan02]
  - Changes the way to represent a graph (Depth-first Search canonical labeling)
  - No candidate generation and false positive pruning
  - Combines growing and checking of frequent subgraphs into one procedure
  - “Connection” to traditional FIM: edges correspond to items

gSpan overview

- Map each graph into a DFS code (a sequence)
- Build a novel lexicographic ordering among these codes
- Construct a search tree based on this lexicographic order

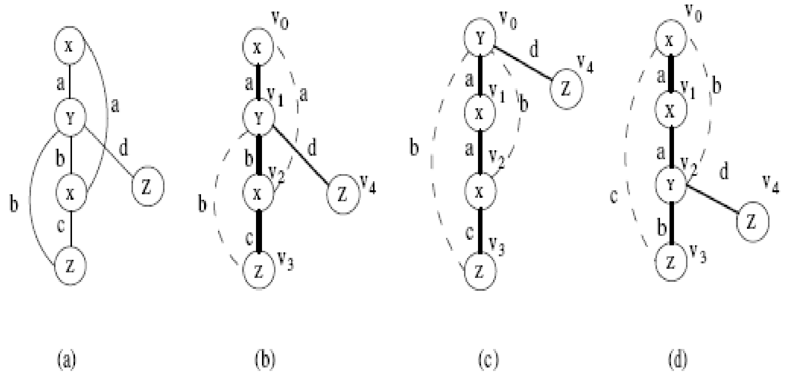
Depth-first Search (DFS) reminder

- Start at the root (select an arbitrary node as the root in case of graphs) and explore as far as possible along each branch before backtracking.



DFS Tree and Forward/Backward edge sets

- For each graph G, we perform DFS and we construct a DFS tree (G_T)
  - Mark vertices on the way thery are traversed: $v_i < v_j$ if v_i is traversed before v_j
  - DFS induces a linear order on vertices
  - v_0:*root*; v_n: *right-most vertex*; *rightmost path*: the direct path from v_0 to v_n.

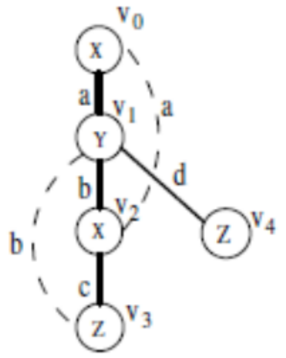


- One graph can have many DFS trees
  - E.g., by selecting different starting nodes
  - graphs (b), (c), (d) are isomorphic to graph (a)
- DFS divides edges in two sets
  - Forward edge set (bold line): *(v_i,v_j)* where $v_i<v_j$
  - Backward edge set (dashed line): *(v_i,v_j)* where $v_i>v_j$

Right most paths:
(b): (v0,v1,v4)
(c): (v0,v4)
(d): (v0,v1,v2, v4)

DFS linear order on the edges

- Linear order on the edges
  - Turn a DFS tree into a sequence of edges
  - Form the sequence in the following order:
    - Start with v_0
    - to extend one new node, add the *forward edge* that connect one node in the old graph with this new node.
    - Add all *backward edges* that connect this new node to other nodes in the old graph
    - repeat this procedure.



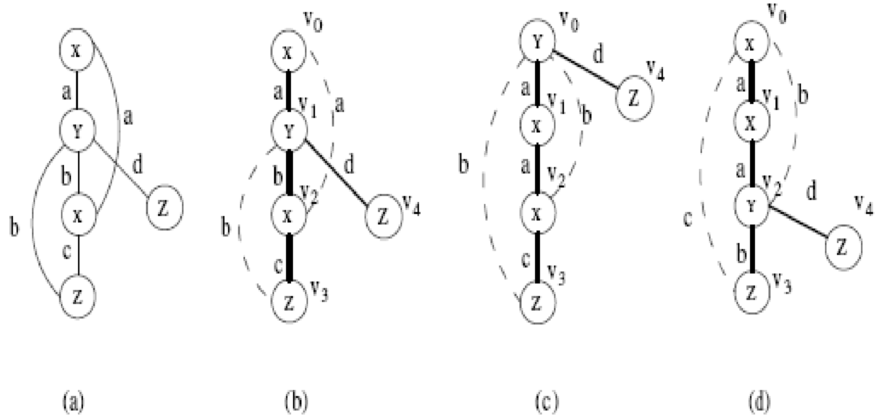
$\{ (v_0,v_1), (v_1,v_2), (v_2,v_0), (v_2,v_3), (v_3,v_1), (v_1,v_4) \}$

Recall: One graph can have many DFS trees➔ different DFS edge orderings

DATABASE SYSTEMS GROUP

DFS lexicographical order

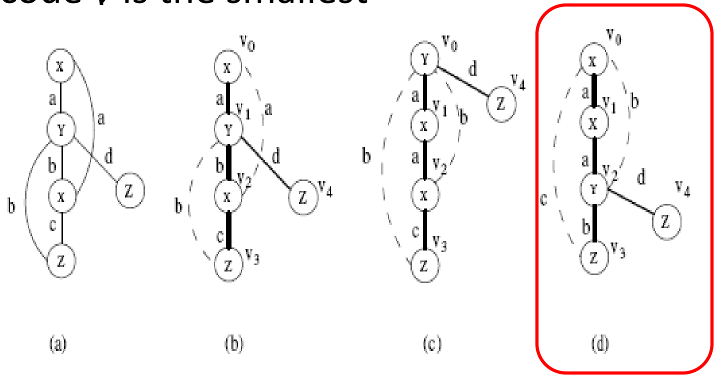
- **Idea:** Add label information as one of the ordering factors
- Each edge is modeled through a 5-tuple entry $(i, j, l_i, l_{(i,j)}, l_j)$
- **DFS code** is a sequence of 5-tuple entries corresponding to edges.
- Assume that there is an order on the labels
- This order together with the edge order defines an order for any two 5-tuple entries
- This extends to DFS code using a lexicographic encoding



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	$(1, 4, Y, d, Z)$	$(0, 4, Y, d, Z)$	$(2, 4, Y, d, Z)$

Canonical labeling

- Each graph *G* may have lots of DFS codes (why?):
- Let the canonical DFS code to be the smallest (based on lexicographic order) code that can be constructed from *G* (denoted *min(G)*) → *Canonical Label of G*
  - i.e., canonical description of subgraphs belonging to one isomorphic class
  - In our example, code γ is the smallest



- Theorem: Given two graphs G and H, they are isomorphic if and only if min(G)=min(H)
- Thus, mining frequent connected subgraphs is equivalent to mining their corresponding minimum DFS codes.

DATABASE SYSTEMS GROUP

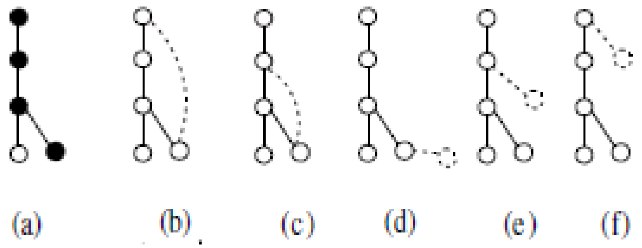
DFS code's parent and child

- Definition: DFS code's parent and child

$$\alpha=(a_0,a_1,..,a_m)$$
$$\beta=(a_0,a_1,..,a_m,\boldsymbol{b})$$

  - *α* is *β*'s parent and *β* is child of *α*
- To construct a valid DFS code, b must be an edge that grows only from the vertices of the rightmost path.
- DFS growth
  - Backward edges can grow only from the rightmost vertex
  - Forward edges from vertices on the rightmost path

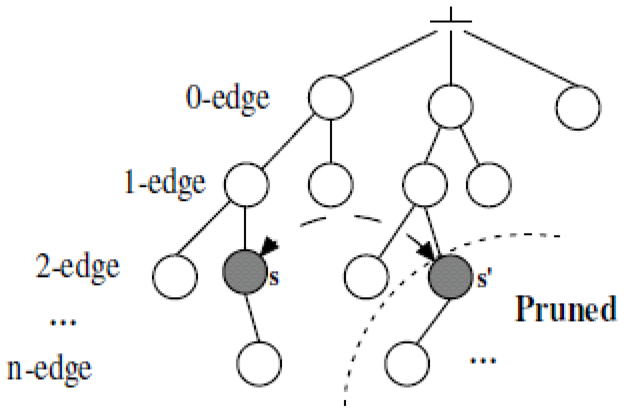
Right-Most-Only Extension is allowed!



Several potential children with one edge growth for graph (a)

DFS code tree

- DFS Code Tree:
  - each node represents a DFS tree and its children are DFS trees grown one edge
  - relations between parents and children complies with parent-child relation definition
  - siblings are consistent with DFS lexicographic order
- Given a label set L, a DFS code tree can be constructed following the above definition



A Search Space: DFS Code Tree

gSpan pseudocode

```
Algorithm    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;        //e.g. $(0,1,A,a,A) < (0,1,A,a,B) < ...$
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($\mathbb{GS}$, $\mathbb{S}$, $s$);        //Grow all nodes in the subtree rooted at this 1-edge graph
10:    $\mathbb{GS} \leftarrow \mathbb{GS} - e$;        //Shrink each graph in GS by removing edge e, after all descendants of
11:    if $|\mathbb{GS}| < minSup$;                      e have been searched.
12:        break;                                        //Successively the graph set becomes smaller → efficiency
```

gSpan pseudocode

- Generate all potential children of s with one edge growth and recursively run the same procedure on each child.

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

```
1: if $s \neq min(s)$          //Prune duplicate subgraphs and all their descendants
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($\mathbb{GS}$, $\mathbb{S}$, $s$);   //Recursion
```

Frequent Subgraph Mining overview

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 extensions is far larger ⇒ candidate generation is more complex

- FSG: Apriori-based method with pairwise candidate generation
- gSpan: Pattern-growth approach for general graphs

References

- Borgwardt K., Kriegel H.-P.: „Shortest-path kernels on graphs“. In Proc. Intl. Conf. Data Mining (ICDM 2005), 2005
- Borgwardt K.: „Graph Kernels“, Dissertation im Fach Informatik, Ludwig-Maximilians-Universität München, 2007
- Bunke, H. : „Recent developments in graph matching“. In ICPR, pages 2117–2124. 2000
- Gärtner, T., Flach, P., and Wrobel: „On graph kernels: Hardness results and efficient alternatives.“ Proc. Annual Conf. Computational Learning Theory, pages 129–143, 2003
- Wiener, H.: „Structural determination of paraffin boiling points“. J. Am. Chem. Soc., 69(1):17–20, 1947
- [AggWan10] C. Aggarwal, H. Wang, *Managing and Mining Graph Data*, Springer, 2010.
- [KurKar01] M. Kuramochi, G. Karypis, *Frequent Subgraph Discovery*, ICDM 2001.
- [YanHan02] X. Yan, J. Han, *Graph-Based Substructure Pattern Mining*, ICDM 2002.

Chapter Overview

Graph Databases and Linked Data

So far: Objects are considered as iid
(independent and identical distributed)

$\Rightarrow$ the meaning of objects depends exclusively on their description

$\Rightarrow$ objects do not influence each other

In the following: Link-Mining

Objects are *connected* and *dependent*.

Examples:

- Publications are evaluated based on citations.
- Webpages are evaluated based on other webpages linking to them.

$\Rightarrow$ objects might depend on any connected object

$\Rightarrow$ databases become large networks (knowledge graphs)

Node Ranking

Idea: Select and rank nodes in large networks w.r.t. their relevance or interestingness.

Interestingness might depend on :

- influence to the complete network
- key nodes for network flows

Applications:

- Ranking web sites and web pages
- Rank researchers in citation networks
- Rank importance of nodes representing crossing or routers in transportation networks

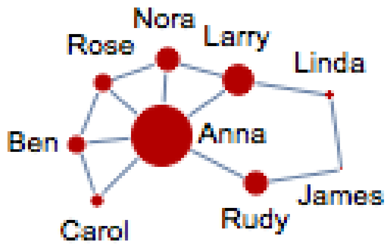
Centrality Measures

Idea: Centrality depends on the "position" of a node to the other nodes in the network w.r.t. networks distance (=cost optimal path between two nodes)

Let *d(v,t)* be the length of the shortest path from *v* to *t (v,t* $\in$*V)* in *G(V,E)*:

- **Closeness Centrality:**
  - Based on shorted distance to every other vertex

$$C_C(v) = \frac{1}{\sum_{t \in V} d(v,t)}$$



Source: https://reference.wolfram.com/language/ref/ClosenessCentrality.html

- **Graph Centrality:**
  - Based on max. shortest distance

$$C_G(v) = \frac{1}{\max_{t \in V}(d(v,t))}$$

Centrality Measures

Let σ_{st} be the number of shortest paths from *s* to *t*.

Let $\sigma_{st}(v)$ be the number of shortest path from *s* to *t* containing *v*.

- **Stress Centrality:**
  - Based on number of shortest paths passing through each node

$$C_S(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)$$

- **Betweenness Centrality:**
  - Normalize by the total number of shortest paths between *s* and *t*

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Centrality Measures

Example: Nodes represent routers in a computer network. If the router having the highest betweenness centrality goes offline, the most direct connections are affected.

Computation: Set of all-pair-shortest paths can be computed in $O(n^3)$ time and using $O(n^2)$ memory by the Floyd-Warshal algorithm.

Theorem: *v* is on the shortest path between *s* and *t* if and only if

$$d(s,t) = d(s,v) + d(v,t)$$

$$\Rightarrow \sigma_{st}(v) = \begin{cases} 0 & \text{if} \quad d(s,t) < d(s,v) + d(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{else} \end{cases}$$

$\Rightarrow$ to compute betweenness centrality it is not necessary to compute all paths

$\Rightarrow$ there are faster solution:

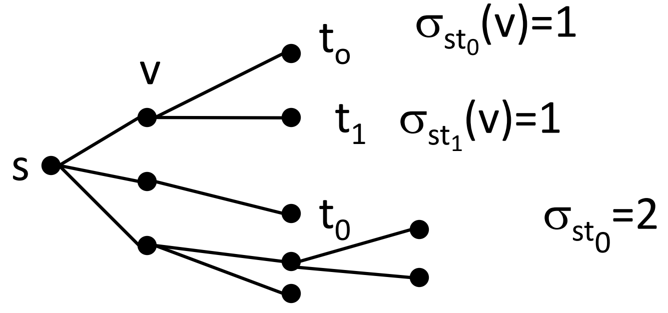
- $O(nm)$ without edge weights
- $O(nm+n^2 \log n)$ in graphs having edge weights

where $n = |V|$ and $m = |E|$ in the graph $G(V,E)$

Computing Betweeness Centrality

Basic idea:

- Start a single source all target search from each node s. The result is a tree (called Dijkstra tree) containing all shortest paths starting with s.
- The Dijkstra tree also induces a distance ranking of all nodes to s.
- Visit each node v with descending distance to s and count all nodes t lying behind v in the tree ($\sigma_{st}(v)$) and the set of shortest paths from s to t (σ_{st})



Algorithm for unweighted graphs(1)

Variables and expressions:

- S: Stack storing nodes w.r.t to their distance to s
- Q: Priority Queue for the Dijkstra search (ordered by the distance to s)
- P[v]: List storing all predecessors of v
- d[v]: distance of the shortest path from s to v
- σ[v]: number of shortest paths from s to v
- δ[v]: Given $\delta_{st}[v] = \frac{\sigma_{st}[v]}{\sigma_{st}}$ then $\delta[v] = \delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}[v] = \sum_{w: v \in P[w]} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \left(1 + \delta_{s\bullet}(w)\right)$

Workflow for each starting node s:

1. Phase: Algorithm computes the Dijkstra tree of s
2. Phase: traverse stack S and count the number of nodes behind each visited node v

DATABASE SYSTEMS GROUP

Algorithm for unweighted graphs(2)

```
CB[v] := 0 ∀ v∈V
for s ∈ V
  S:= empty Stack;
  P[w] := empty List ∀ w∈V;
  σ[t] :=0  ∀t∈V; σ[s]:=1;
  d[t] :=-1 ∀t∈V; d[s]:0;
  Q := empty Queue;
  Q.push(0,s);
  while Q not empty do
     v := Q.pop();
     S.push(v);
     foreach neighbor w of v do
        if d[w] < 0 then
           d[w]:=d[v]+1;
           Q.push(d[w],w);
        end if
        if d[w]=d[v]+1 then
          σ(w):=σ(w)+σ(v)
          P[w].add(v)
        end if
     end for
  end while
  δ[v]:=0; v∈V;
  while S not empty do
     w:=S.pop();
     for v∈P[w] do
        $\delta[v] := \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w]);$
     end for
     if w≠s then
        CB[w]:=CB[w]+δ[w];
     end if
  end while
end for
```

Ranking nodes in hyperlinked Text

PageRank: (S.Brin/B. Page 1996)

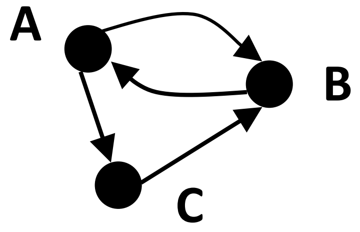
- important component in ranking algorithms of search engines (in combination with other features)
- Data is considered strongly connected, directed network G(V,E). (e.g., all HTML documents in a search engine)
- probabilistic surfer performs an infinite random walk. idea: visiting probability = importance of the page v.

PageRank

Computing the PageRank

start distribution: *p0(u) = 1 / |V |*

$$E = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$



adjacency matrix: E

transition prob.: $L[u,v] = \frac{E[u,v]}{\sum_{\beta} E[u,\beta]}$

$$L = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

probability of page v at time i : $p_i[v] = \sum_{u \in V} L[u,v] p_{i-1}(u)$

distribution vector over all pages: $\vec{p}_i = L^T \vec{p}_{i-1}$

Computation by „Power Iterations“: $\vec{p}_i \leftarrow L^T \vec{p}_{i-1}$

after ca. 20-30 iterations result should be stable

Solution for none strongly connected graphs: 1. Remove nodes without outlink

2. Allow jumps during traversal

Ranking Linked Objects

HITS (Kleinberg 1998): Hyperlink Induced Topic Search

- Consider only objects being relevant for *q* or being linked to relevant pages (in- and outlinks).

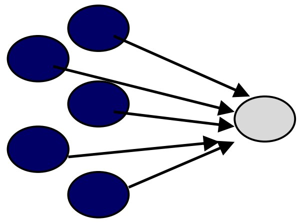
  $\Rightarrow$ $G_q(V_q,E_q)$ *for query q*
- There are two types of objects :

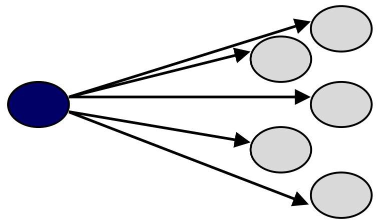
  Hubs: link to relevant objects (authorities)

  Authorities: relevant objects being linked by hubs.

  => each object has an authority score and a hub score

  for each object u, h[u] denotes its hub score and a[u] its authority score.


a good authority is linked by many good hubs


a good hub links to many good authorities.

Anwendungen im Web Mining

Computing HITS:

- $\vec{a}$ vector of authority scores over all objects v $\in V_q$
- $\vec{h}$ vector of hub scores over all objects v $\in V_q$

- Computation by mutual iterations: $\vec{a} = E^T \vec{h}$ (authority score)

  $\vec{h} = E\vec{a}$ (hub score)

Complete algorithm:

1. determine relevant objects (root set).
2. determine all pages linking relevant objects .(extended set)
3. iterate over all hub- and authority scores
4. Order the relevant pages by the authority scores

Link Prediction

Input: A graph G(V,E) and 2 nodes v,u $\in$ V where (v,u) $\notin$ E.

Output: Predict the existence of link (v,u) if:

- the existence is unknown.
- the link might develop at a future point in time

Examples:

- Links in social networks
- unknown protein interaction
- Customer product recommendations in bipartite graphs (Collaborative Filtering)

Feature-based Link Prediction

Idea: Use the features of pairs of objects to describe their relationship.

Example:

- Common interests in social networks
- Co-authors do research in the same area
- proteins have complementary active regions

$\Rightarrow$ Links do develop by accident, there are reasons which might be found in the feature values

$\Rightarrow$ Link Prediction: Learn a classifier that maps pairs of feature descriptions to link probabilities

$\Rightarrow$ Formal: Let u,v $\in$ V and let F(v),F(u) be their feature descriptions. Then, Link Prediction is the task to learn a function P: (F(v),F(u))-> L.

(L is either discrete {link, no link} or real-valued [0,..max_Strength])

Topology-based Link Prediction

Problem: Feature-based approach do not consider network proximity.

Example:

- Persons having similar interests might not have any contact
- Proteins might dock but do not appear in the same natural surrounding

Solution: Integrate the neighborhood of v and u in G.

$\Rightarrow$ common neighbors increase the likelihood of a link

$\Rightarrow$ describe a node by its adjacency list or the subnetwork being influenced by the node

Link Prediction and Matrix Factorization

Input: Graph G(V,E) with adjacency matrix A and let $E_u \subseteq E$ be the set of links with unknown existence or strength.

Method:

- Factorizing A allows to find a latent k-dimensional space (k is the rank of A) (Factorization can be done regardless of missing entries)
- nodes can be expressed in this latent space
- remapping of the nodes to the |V| dimensional space fills up the unknown entries E_u .

Procedure:

- Factorize A in the n×k Matrix B while minimizing L(B) the : $A' = BB^T$

$$L(B) = \sum_{a_{i,j} \in A \setminus U} \left| a_{i,j} - a'_{i,j} \right|^2 = \sum_{a_{i,j} \in A \setminus U} \left| a_{i,j} - \left\langle b_{i,*}, b_{*,j} \right\rangle \right|^2$$

Computation: Gradient descent on the derivate of L(B).

Remark: Also applicable to bipartite graphs (customer/ product)

Dense Subgraph Discovery

- Find „dense“ subgraphs in a network G(V,E).
- Definitions of „dense“:
  - cliques (complete subgraphs)
  - quasi-cliques (at least x % of the edges must exist)
  - relative density of the surrounding: in node in subgraph G‘ has more links to other node from G‘ than to nodes G \ G‘.
- …
- Problem: almost all definitions lead to NP-hard search problems
  => heuristic solutions
  => practical use is limited

Graph Clustering

- Class of clustering methods that treat the data set as graph
- Object= node; links distance, similarity, reachability distance...
- usually: only consider the k-nearest neighbors or an ε-range

  => directed and undirected network are considered

Clustering by weighted k-mincut:

Partition a graph G into k disjunctive subgraphs having similar size while minimizing the number of removed edges.

=> Weighted k-mincut is also an NP-hard problem.

Graph Clustering: Spectral Clustering

- built a symmetric adjacency matrix S: $S_{i,j} = sim(x_{i,} x_j)$

- Transform S into a graph Laplacian matrix L:

$$L = I - D^{-\frac{1}{2}} S D^{-\frac{1}{2}} \qquad D_{i,j} = \begin{cases} \sum_k sim(x_{i,} x_k) & if \quad i = j \\ 0 & else \end{cases}$$

- after eigenvalue decomposition of L:
  - Eigenvectors with eigenvalues = 0, represent connected components
  - Eigenvectors describe the linear weights to represent a cluster representative

$$r_k = \sum_{i=1}^{|DB|} EV_i \cdot o_i$$

Conclusions Graph Mining

- Graph-Mining includes new data mining tasks
  - Ranking nodes
  - Link prediction
  - Dense subgraph discovery and community detection
  - Frequent subgraph mining
- Clustering can be formulated as a graph problem
  - Density-based clustering: find all connected components where links denote a similarity predicate
  - Spectral clustering
  - Weighted *k*-mincut: Partition a graph into k subgraphs while minimizing the weights of the cut edges under size constraints w.r.t. the resulting subgraphs.

Literature

- Brin S., Page L.:“The anatomy of a large-scale hypertextual Web search engine“, Computer Networks and ISDN Systems, Vol 30, Nr.1-7, S.107-117,1998
- Kleinberg J. M.:“Authoritative sources in a hyperlinked environment“, Journal of the ACM,Vol.46, Nr. 5, S. 604-632,1999
- Brandes U.:“A faster Algorithm for Betweenness Centrality“, Journal of Mathematical Sociology, 25(2):163-177, 2001