Knowledge Discovery in Databases II

Summer Term 2018

Lecture 5:
Graphs

Lectures : Prof. Dr. Peer Kröger, Yifeng Lu
Tutorials: Yifeng Lu

http://www.dbs.ifi.lmu.de/cms/studium_lehre/lehre_master/kdd218

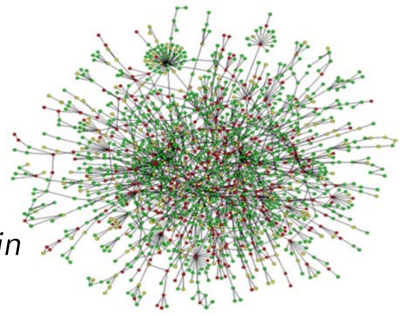
DATABASE SYSTEMS GROUP

Outline

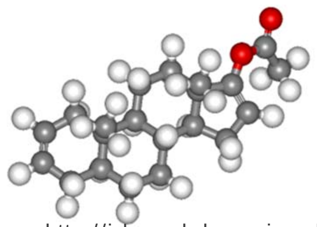
LMU

Introduction

- Graphs, graphs everywhere!
  - Chemical data analysis, proteins
  - Biological pathways/networks
  - Program control flow, traffic flow, work flow analysis
  - XML, Web, social network analysis
- Graphs form a complex and expressive data type
  - Trees, lattices, sequences, and items are degenerated graphs
  - Different applications result in different kinds of graphs and tasks
    - Diversity of graphs and tasks → diversity of challenges
  - Complexity of algorithms: many problems are of high complexity (NP-complete or even P-SPACE!)



Yeast Protein Interaction Network



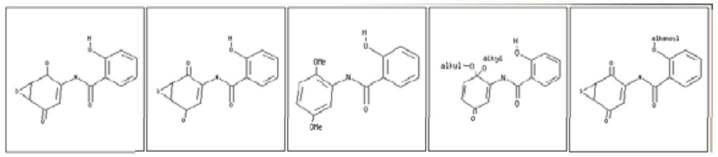
source: http://jchemed.chem.wisc.edu/

Social Network Graph (facebook, Dez 2010)

DATABASE SYSTEMS GROUP

Graph Data vs. Network Data

- Different applications result in different kinds of graphs and tasks
  - E.g. chemical graphs: relatively small, repeating vertex labels
  - E.g. large scale domains (web, computer networks, social networks): very big, vertex labels are distinct
- Diversity of graphs and tasks → diversity of challenges
- Graph mining can be divided into two fundamental settings:
  - Mining in a **set of graphs**, e.g.:
    - Finding similar graphs
    - Determining all frequent subgraphs
    - Classification of graphs
  - Mining in **one single large graph**, e.g.:
    - How does the network ´behave´?
    - Determine striking patterns,
      e.g. homogeneous and connected components



Social Network Graph (facebook, Dez 2010)

Basic Definitions

- Definition *Directed, Simple Graph*:

  A directed, simple graph is a tuple *g=(V,E)* comprising a set *V* of vertices and a set *E* of edges.

  Edges are 2-element subsets of the vertices ($E \subseteq V \times V$). The relation is represented as ordered pair of the vertices (directed). Loops and multiple edges are disallowed (simple).

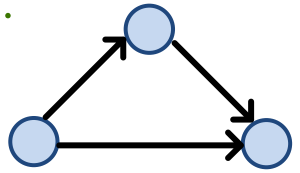
  - *V(g)* describes the set of vertices of the particular graph *g*.
  - *E(g)* describes the set of edges of the particular graph *g*.

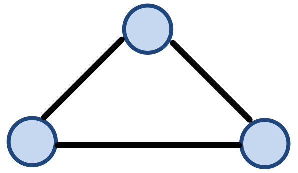
- Definition *Undirected, Simple Graph*:

  Default

  An undirected, simple graph is a tuple *g=(V,E)* comprising a set *V* of vertices and a set *E* of edges.

  Edges are 2-element subsets of the vertices ($E \subseteq V \times V$). The relation is represented as unordered pair of the vertices (undirected). Loops and multiple edges are disallowed.


directed, simple graph


undirected, simple graph

DATABASE SYSTEMS GROUP

Basic Definitions

LMU

- If not stated otherwise, we are dealing with *undirected, unlabeled, simple graphs*!
  - For simplicity we will write $e = (v_i, v_j)$ also for undirected edges!
- Definition *Labeled Graph*:

  A labeled graph is a triplet *g=(V,E,l)* with a set of vertices *V*, a set of edges *E*, and a label function *l*, which maps a vertex or an edge to the label set:

  $$\Sigma \ (l: V \cup E \rightarrow \Sigma).$$
- The (infinite) set of all graphs will be denoted as:

  $$\mathcal{G} \ (\mathcal{G} \subseteq \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N} \times \mathbb{N}))$$

DATABASE SYSTEMS GROUP

Basic Definitions

LMU

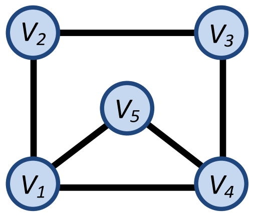
- *Walk/Path*: A walk or a path in a graph g is a sequence of vertices $p = (v_1, v_2, \dots, v_k)$ such that from each of its vertices there is an edge to the next vertex in the sequence ($\forall 1 \le i \le k-1: (v_i, v_{i+1}) \in E(g)$).
  - The length $len(p)$ of the walk/path is the number of edges traversed.
  - The set of vertices traversed by path p is denoted by $V(p) = \{v_1, \dots, v_k\}$
  - A walk/path is *closed* if its first and last vertices are the same, and *open* if they are different.
  - A *simple walk/path* is one where no vertices are repeated.
  - The first vertex of a walk/path is called its *start vertex*. The last vertex of a finite walk/path is called its *end vertex*. The intermediate vertices of the walk/path are called *internal vertices*.

Basic Definitions

- *Trail*: A trail is a walk in which all the edges are distinct. A closed trail is called *tour.*
- *Label sequence*: A label sequence in a labeled graph g is a sequence of vertex labels $ls = (l(v_1), l(v_2), \dots, l(v_k))$ such that from each of its vertices there is an edge to the next vertex in the sequence ($\forall 1 \leq i \leq k - 1: (v_i, v_{i+1}) \in E(g)$).
  - If also the edges are labeled, the label sequence expands to an alternating sequence of vertex and edge labels $ls = (l(v_1), l(e_1), l(v_2), l(e_2), \dots, l(e_{k-1}) l(v_k))$, s.t. $\forall 1 \leq i \leq k - 1: e_i = (v_i, v_{i+1}) \in E(g)$.
- Shortest path: The shortest path between two vertices v_i and v_j in a graph g is the path witch traverses the minimal number of edges $p_{min}(v_i, v_j) = \underset{p \in \{path=(v_1, \dots v_k) | v_1 = v_i \wedge v_k = v_j\}}{\operatorname{argmin}} len(p)$

Basic Definitions

- The *adjacency matrix* of a simple graph $M(g)$ is a $|V(g)| \times |V(g)|$ matrix with entries $M[i,j] = 1$ or $M[i,j] = 0$ according to whether $(v_i, v_j) \in E(g)$ or $(v_i, v_j) \notin E(g)$.

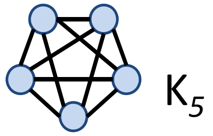
$g =$ 

$$M(g)=\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \quad M(g)^2=\begin{bmatrix} 3 & 0 & 2 & 1 & 1 \\ 0 & 2 & 0 & 2 & 1 \\ 2 & 0 & 2 & 0 & 1 \\ 1 & 2 & 0 & 3 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{bmatrix} \quad M(g)^3=\begin{bmatrix} 2 & 5 & 1 & 6 & 4 \\ 5 & 0 & 4 & 1 & 2 \\ 1 & 4 & 0 & 5 & 2 \\ 6 & 1 & 5 & 2 & 4 \\ 4 & 2 & 2 & 4 & 2 \end{bmatrix}$$

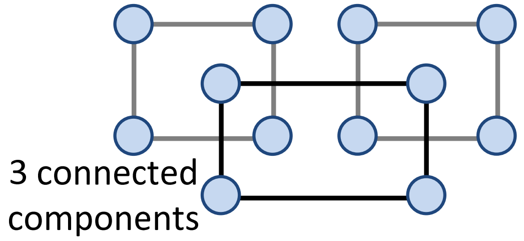
- The number of all paths of length n from v_i to v_j in a graph g is the (i,j) entry of $M(g)^n$
- *Adjacent*: Two vertices are adjacent if they are connected by an edge.

Basic Definitions

- If not stated otherwise, the *size/cardinality* of a graph is defined as: $|g| = |V(g)|$
- *Complete graph*: A complete graph or *clique* is a graph in which each vertex is adjacent to every other vertex ($\forall v_i, v_j \in V(g): (v_i, v_j) \in E(g)$).
  - A complete graph with n vertices is denoted by K_n.

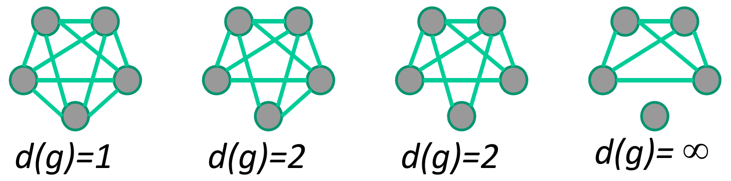


- *Connected*: A graph is connected if there is a path connecting every pair of vertices ($\forall v_i, v_j \in V(g): \exists p = (v_1, \dots, v_k)\ with\ v_1 = v_i\ \wedge v_k = v_j$)
  - A graph that is not connected can be divided into *connected components* (disjoint connected subgraphs).

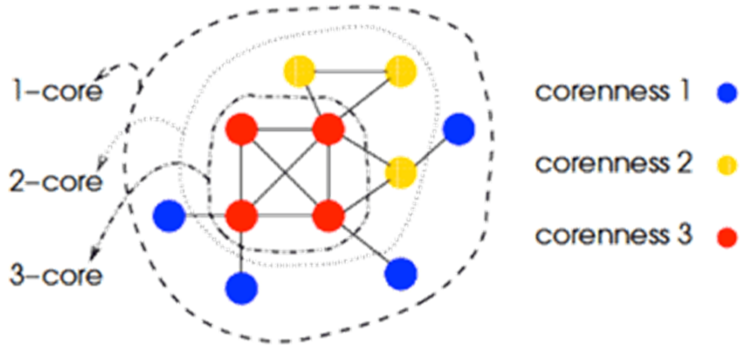


Basic Definitions

- *Diameter*: The diameter $d(g)$ of a graph is its '*longest shortest path*', i.e., the maximum among minimal paths between pairs of its vertices. $d(g) = \max\{len(p_{min}(v_i, v_j) | v_i, v_j \in V(g))\}$
  - $\mathrm{d}(g) = 1$ implies that g is complete.
  - $d(g) = \infty$ implies that g is not connected.



- *K-degenerate graph (k-core graph)*: An undirected graph in which every subgraph has a vertex of degree at most *k* is called a *k-core graph*.



Basic Definitions

- The *degree* of a vertex ($deg(v)$) is the number of edges incident to the vertex.
  - A vertex with degree 0 is called an *isolated vertex*.
  - A vertex with degree 1 is called a *leaf* or *end vertex*.
- *Indegree, Outdegree*: For directed graphs it is useful to differentiate between ingoing and outgoing edges:
  - The indegree $\deg^{+}(v)$ of a vertex is the number of head endpoints adjacent to it:

    $$\deg^{+}(v) = |\{(w, v) \in E(g)\}|$$

  - The outdegree $\deg^{-}(v)$ of a vertex is the number of tail endpoints adjacent to it:

    $$\deg^{-}(v) = |\{(v, w) \in E(g)\}|$$

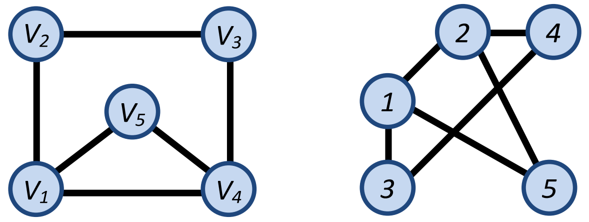
Isomorphism & Subgraphs

- Definition *Graph Isomorphism*:

  For two labeled graphs g and g', a graph isomorphism is a bijective function $f: V(g) \rightarrow V(g')$, such that:

  1. $\forall v \in V(g)$: $l(v) = l'\left(f(v)\right)$
  2. $\forall (u,v) \in E(g)$: $\left(f(u), f(v)\right) \in E(g')$ and $l(u,v) = l'\left(f(u), f(v)\right)$
  3. $\forall (u,v) \in E(g')$: $\left(f^{-1}(u), f^{-1}(v)\right) \in E(g)$ and $l'(u,v) = l\left(f^{-1}(u), f^{-1}(v)\right)$

- A graph g is isomorphic to g' ($g \cong g'$) if there exists a graph isomorphism from g to g'



$v_1 \rightarrow 1$
$v_2 \rightarrow 3$
$v_3 \rightarrow 4$
$v_4 \rightarrow 2$
$v_5 \rightarrow 5$

Isomorphism & Subgraphs

- Definition *Subgraph Isomorphism*:

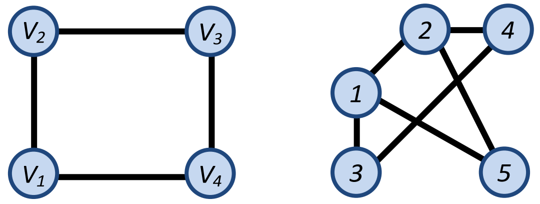
  For two labeled graphs g and g', a subgraph isomorphism is an injective function
  $f: V(g) \rightarrow V(g')$, such that:

  1. $\forall v \in V(g), l(v) = l'(f(v))$
  2. $\forall (u,v) \in E(g), (f(u), f(v)) \in E(g')$ and $l(u,v) = l'(f(u), f(v))$

  Where *l* and *l'* are the labeling functions of g and g' respectively.

  f is called an embedding of g in g'.

- A graph g is a *subgraph* of another graph g' ($g \subseteq g'$) if there exists a subgraph isomorphism from g to g'.

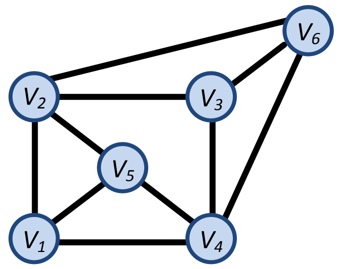


$e.g.: v_1 \rightarrow 2$
$v_2 \rightarrow 4$
$v_3 \rightarrow 3$
$v_4 \rightarrow 1$

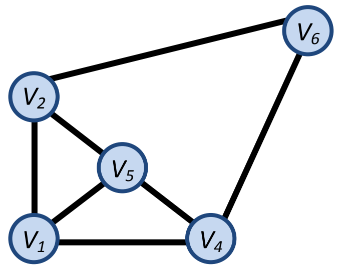
Induced Subgraphs

- Definition *Induced Subgraph*:

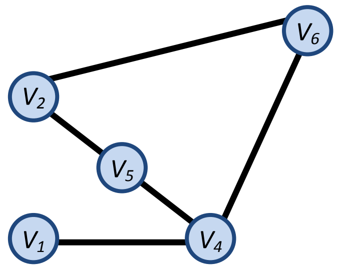
  A subgraph g' of a given graph g is an induced subgraph ($g' \subseteq_{ind} g$), iff $\forall v_i, v_j \in V(g'): (v_i, v_j) \in E(g) \Leftrightarrow (v_i, v_j) \in E(g')$. The graph g' is called the graph induced by the vertices $V(g')$ in g.



original graph



induced subgraph



not induced subgraph

g' contains all the edges of g that connect elements of the given subset of the vertex set of g, and only those edges.

DATABASE
SYSTEMS
GROUP

Literature and References

LMU

- References for each method are provided in the corresponding chapters

- An overview of the area is given by the textbook

  Managing and Mining Graph Data

  Charu C. Aggarwal, Haixun Wang

  Springer, 2010

- The book is available in our „Handapparat“

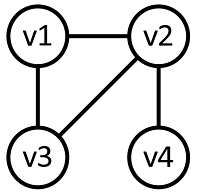
Outline

- Graph Introduction
  - Basic Definitions
- **Graph Similarity**
  - **Exact Graph Matching**
  - **Error-tolerant Graph Matching**
- Frequent Subgraph Mining

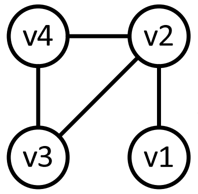
Similarity between Graphs

- Similarity between objects basic requirement for mining and exploration
  - Retrieval, Clustering, Classification, …
    - Many techniques (cf. Data Mining I) rely on similarity/distance measures
- Traditional vector data: several distance functions introduced
  - Euclidean Distance, Cosine Distance, Mahalanobis Distance, …
- Similarity between graphs more complex
  - Arbitrary permutation of nodes still results in same graph

  → Computing, e.g., Frobenius norm („entrywise" Euclidean Distance) between two adjacency matrices not meaningful

g_1 

$$M_1 = \begin{bmatrix} 0&1&1&0 \\ 1&0&1&1 \\ 1&1&0&0 \\ 0&1&0&0 \end{bmatrix}$$

g_2 

$$M_2 = \begin{bmatrix} 0&1&0&0 \\ 1&0&1&1 \\ 0&1&0&1 \\ 0&1&1&0 \end{bmatrix}$$

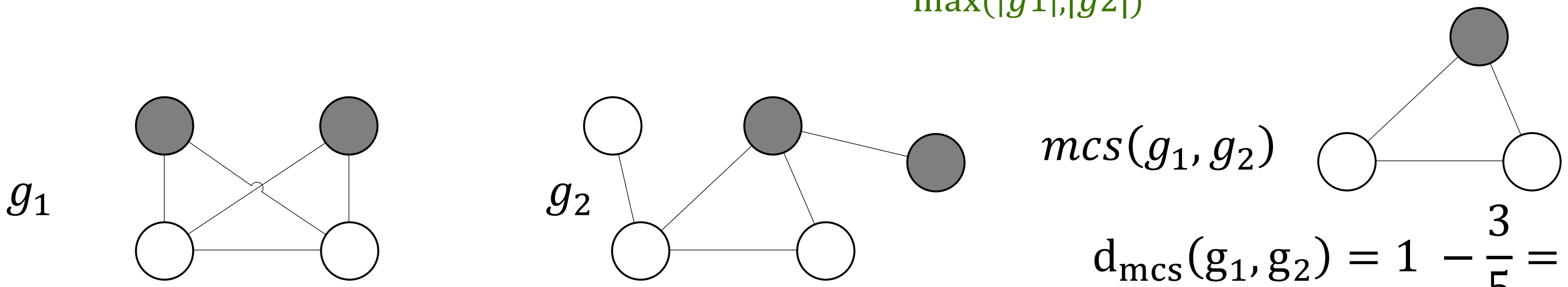
$g_1 \cong g_2$
but
$\|M_1 - M_2\|_F = 2$

DATABASE SYSTEMS GROUP

Exact Graph Matching

LMU

- The most simple similarity measure: Isomorphism
  - $dist_{iso}(g1, g2) = \begin{cases} 0, & if\ g1 \cong g2 \\ 1, & else \end{cases}$
  - Obviously: too restrictive/sensitive, just binary decision
    - graphs have to be completely identical
- Better solution: use of Maximum Common Subgraph
  - Largest part of two graphs that is identical
  - Common (induced) subgraphs $cs(g1, g2) = \{x \in \mathcal{G} | x \subseteq_{ind} g1 \wedge x \subseteq_{ind} g2\}$
  - Maximum common subgraph $mcs(g1, g2) = \underset{g \in cs(g1,g2)}{\operatorname{argmax}} |g|$
  - Distance function: $\mathrm{d_{mcs}}(\mathrm{g1}, \mathrm{g2}) = 1 - (\frac{|mcs(g1,g2)|}{\max(|g1|,|g2|)})$

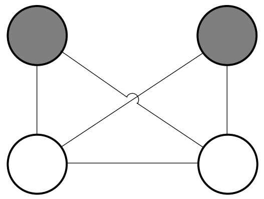


$$\mathrm{d_{mcs}}(\mathrm{g_1}, \mathrm{g_2}) = 1 - \frac{3}{5} = \frac{2}{5}$$

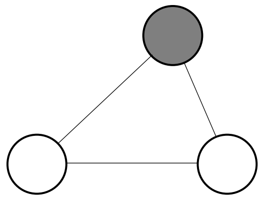
Exact Graph Matching

- Extension: Consider also the Minimum Common Supergraph
  - Smallest supergraph that „contains“ both other graphs
  - Common supergraphs: $\mathrm{CS}(\mathrm{g1},\mathrm{g2}) = \{x \in G \mid g1 \subseteq_{ind} x \wedge g2 \subseteq_{ind} x\}$
  - Minimum common supergraph $MCS(g1, g2) = \underset{g \in CS(g1,g2)}{\operatorname{argmin}} |g|$
  - Distance function: $d_{MMCS}(g1, g2) = |MCS(g1, g2)| - |mcs(g1, g2)|$

g_1

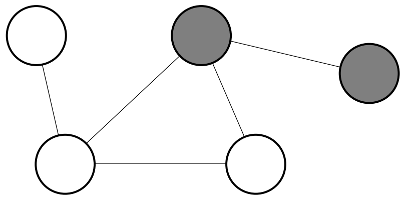


$mcs(g_1, g_2)$

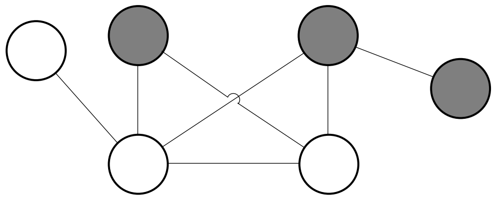


$\mathrm{d}_{\mathrm{MMCS}}(\mathrm{g}_1, \mathrm{g}_2) = 6 - 3 = 3$

g_2

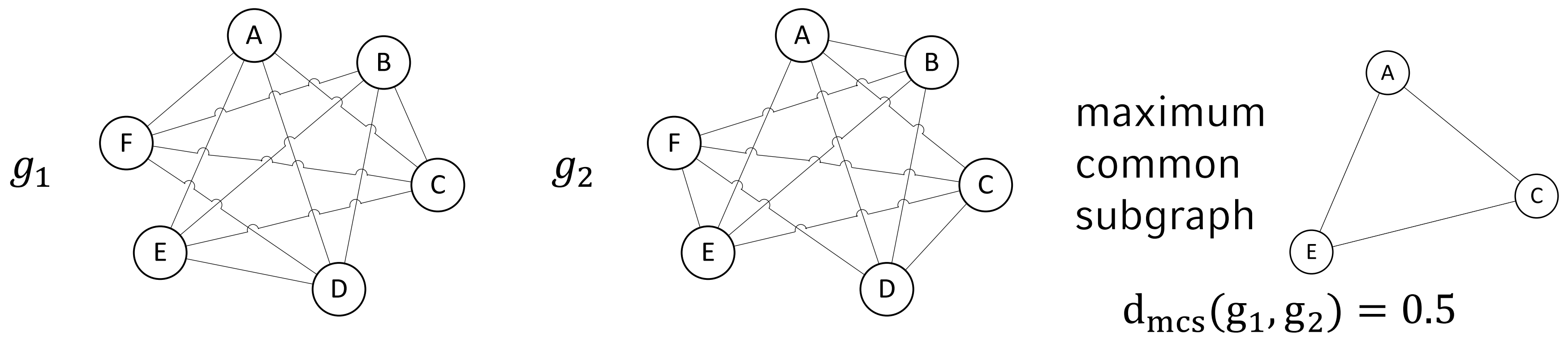


$\mathrm{MCS}(g_1, g_2)$



Problems of Exact Graph Matching

- Problem of previous distance measures
  - To obtain high similarity, a significant part of the topology and of the labels need to be *identical*
  - Just a few missing edges or slightly different labels lead to low similarity
    - Real world data, however, is often noisy and contains some errors
  - If labels are selected from continuous domains (e.g. $\mathbb{R}$) very unlikely to detect identical (sub)graphs



$$d_{mcs}(g_1, g_2) = 0.5$$

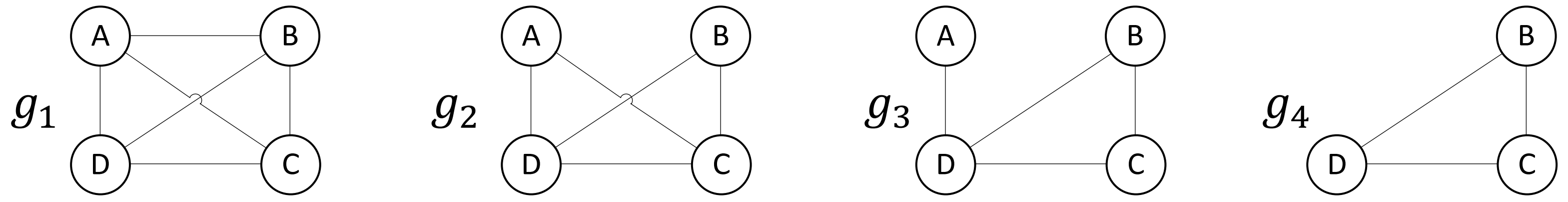
- → Error-tolerant graph matching

DATABASE SYSTEMS GROUP

Error-tolerant Graph Matching

LMU

- Idea: Do not enforce identical patterns, but „just“ penalize deviations
  - E.g. the more dissimilar the labels, the higher the penality
  - E.g. two missing edges worse than one missing edge



$\mathrm{d_{mcs}}(\mathrm{g_i}, \mathrm{g_j}) = \frac{3}{4} \quad \forall i \neq j$ since g_4 is maximum common subgraph

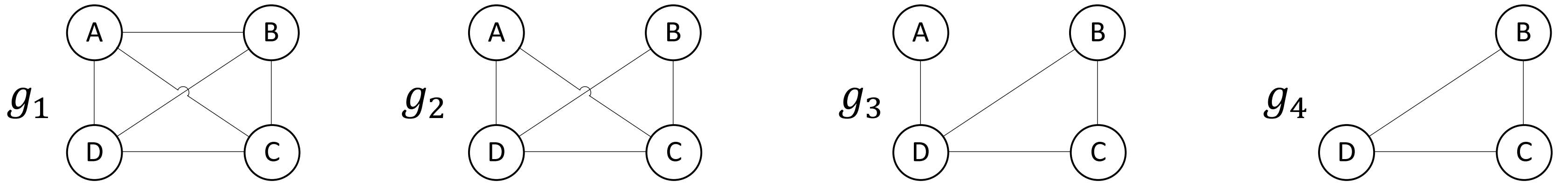
However, intuitively $g1$ more similar to $g2$ than to $g4$

- We briefly discuss two paradigms
  - Vector space embeddings of graphs
  - Graph Edit Distance

Vector Space Embeddings of Graphs

- Idea: Extract characteristic (and numerical) features of the graph
  - E.g. number of nodes, number of edges, …
  - In chemistry such features are called „topological indices“
- Each feature corresponds to one dimension in a vector space
- Similarity of graphs = similarity of vectors in feature space
  - E.g. using Euclidean Distance, Dot Product, …

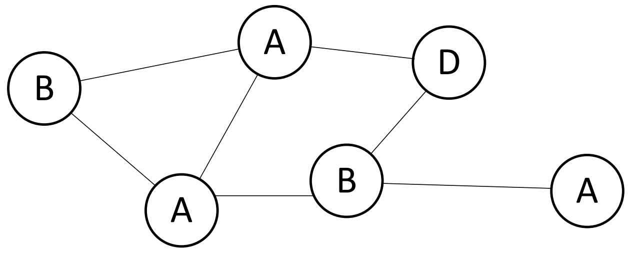
Example for $f(g) = [\,|V(g)|\,, |E(g)|\,]$



$f(g_1) = [4,6]$ $f(g_2) = [4,5]$ $f(g_3) = [4,4]$ $f(g_4) = [3,3]$

Multi-dimensional Features

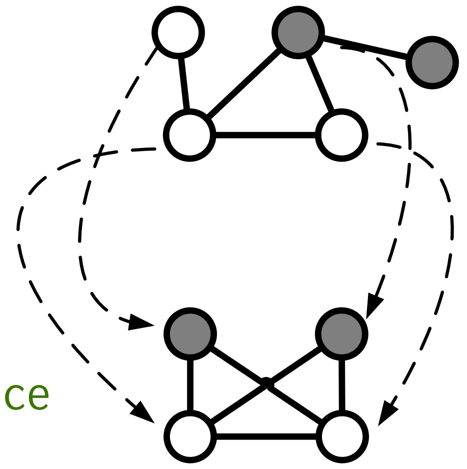
- Previous indices „compress“ the graph to a single value
- Easy interpretation but potentially too rough for measuring similarity
- → Extract multi-dimensional features
- Label histogram
  - Each bin of the histogram represents a label $l \in \Sigma$ and stores the number of nodes with label l
  - Feature space: $h_{label}(g) \in \mathbb{R}^{|\Sigma|}$
  - $h_{label}(g)[i] = |\{v \in V(g) | l(v) = l_i\}|$ with $\Sigma = \{l_1, \dots, l_{|\Sigma|}\}$
  - Limited to discrete label domains, i.e. finite Σ



$h_{label}(g) = [3,2,0,1]$ with $\Sigma = \{A, B, C, D\}$

Graph Similarity based on Edit Distance

- Previous approaches map graphs to vector spaces
  - Similarity of graphs is distance in novel vector space
- Now: Try to transform graph g1 into graph g2
  - „The smaller the transformation costs, the similar the graphs“
  - Idea: Adapt *String* Edit distance to *Graph* Edit Distance

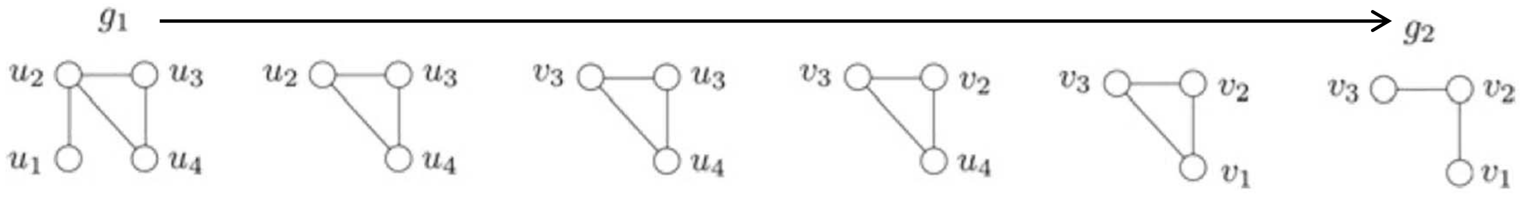


- String Edit Distance:
  - Minimal number of editing operations (insertions, deletions, substitutions) for transforming sequence *s* into sequence *q*.
  - Example: D(“TÜRSCHLOSS”, “ABSCHUSS”) = D(s,q)= 5
    - two deletions (◊) and three substitutions (:) are necessary.
    - Five symbols are unmodified (|)

```
s  =  T Ü R S C H L O S S
      ◊ : : | | | ◊ : | |
q  =    A B S C H   U S S
```

Graph Edit Distance

- Graph Edit Operators:
  - Vertex insertion, deletion, substitution
  - Edge insertion, deletion, substitution
  - A list of operations edit one graph to another is a Edit Path $\mathcal{P}(g_1, g_2)$



Node operations:

$$\{u_1 \to \phi, u_2 \to v_3, u_3 \to v_2, u_4 \to v_1\}$$

Edge operations:

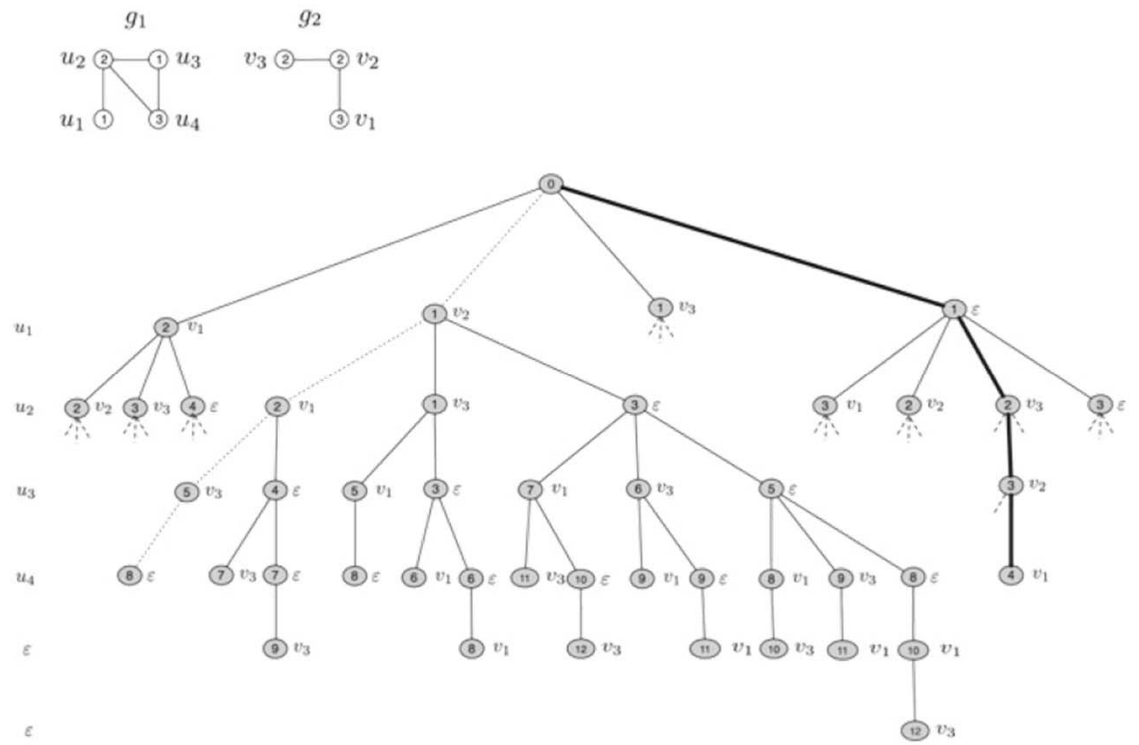
$$\{(u_1, u_2) \to \phi, (u_2, u_3) \to (v_3, v_2), (u_3, u_4) \to (v_2, v_1), (u_4, u_2) \to \phi\}$$

- Graph Edit Distance:

$$GED(g_1, g_2) = \min_{(e_1, e_2, \ldots, e_k) \in \mathcal{P}(g_1, g_2)} \sum_{i=1}^{k} c(e_i)$$

Graph Edit Distance

- All possible edit path form a edit tree
- Computing Graph Edit Distance is equivalent to finding the shortest path in the tree (e.g.: A*-algorithm)



- Possible number of Edit Path grows exponentially (NP-hard)

Summary

- Different approaches to measure similarity between graphs
- Exact graph matching
  - Isomorphism, maximum common subgraph, …
- Error-tolerant graph matching
  - Mapping of graphs to feature vectors (degree histogram, …)
  - Graph Edit Distance

- Given a similarity/distance measure, many interesting mining tasks can already be performed for graph data!
  - e.g. *Graph Clustering* by using k-Medoid and Graph Edit Distance

Outline

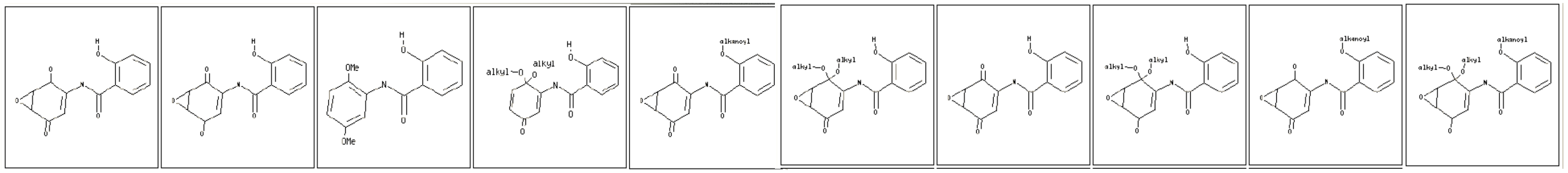
DATABASE SYSTEMS GROUP

Introduction

LMU

- Input: collection of graphs $DB = (g_1 \ldots , gn)$ consisting of undirected, labeled graphs $g_i = (V_i, E_i, l_i)$, where $'l$ is a labeling function mapping an edge or a vertex to a label
- Aim: determine all connected graphs that occur as subgraph in at least a given percentage (*support*) or number (*frequency*) of all graphs in *DB*
- Applications:
  - As preprocessing: characterizing graph sets, discriminating different groups of graphs, classifying graphs, clustering graphs, building graph indices, facilitating similarity search
  - Bioinformatics, computer vision, video indexing, chemical informatics

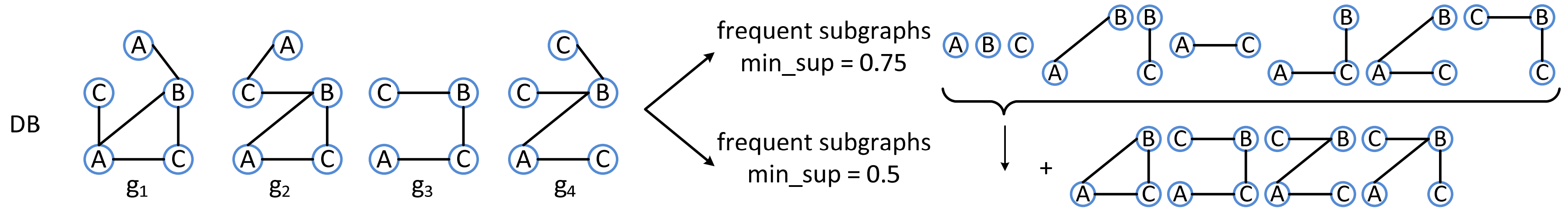
E.g. frequent molecular fragments (e.g. in drug discovery)



Basics

- Analogy to "traditional" frequent itemset mining:
  - Each graph g_i of the graph database DB represents a transaction
  - Each subgraph represents an itemset
- Formal definitions:
  - A graph g' is a *subgraph* of another graph g ($g' \subseteq g$) if there exists a subgraph isomorphism from g' to g
  - $support(g') = \frac{|D_{g'}|}{|DB|}$, with *supporting graphset* $\mathrm{D}_{g'} = \{g_i | g' \subseteq g_i, g_i \in DB\}$.
  - A subgraph g' is *frequent* if its support is no less than a threshold min_sup.

  Example:



  - *Anti-Monotonicity*: A size-k subgraph is only frequent if all of its subgraphs are frequent.

General Approach

- Naive approach: test frequency of all possible subgraph patterns
  - Frequency calculations require subgraph isomorphism test (NP-complete)

  → Try to early exclude some patterns from further considerations

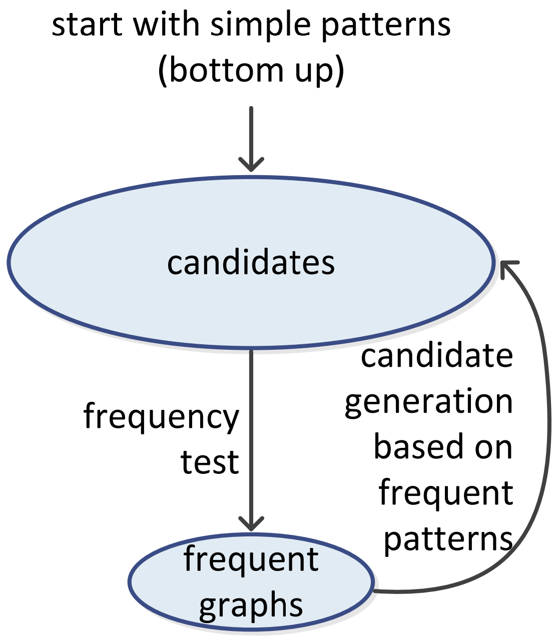
- General (iterative) approach for discovering frequent subgraphs:

  1st step: generate frequent subgraph candidates

  2nd step: check the frequency of each candidate

  → Goal: try to keep the candidate set small!

- Two basic approaches (exploiting the anti-monotonicity criterion):
  - Apriori-based approach
  - Pattern-growth approach


start with simple patterns (bottom up)
candidates
frequency test
candidate generation based on frequent patterns
frequent graphs

Apriori-based Approach

- Works analogously to Apriori-based frequent itemset mining
- Exploiting the anti-monotonicity in a bottom-up algorithm:
  - Start with small-size subgraphs (e.g. single nodes)
  - In each iteration:

    Candidate generation:
    - Increase the size of new frequent subgraph candidates by one
    - Generate new candidates by *joining* two similar but slightly different frequent subgraphs of the previous iteration

    Check the frequency of the just built candidates
- How to join two graphs of size *k* to a graph of size *k+1* ?
  How is the size of a graph defined?
  - typical approaches: AGM[IWM00], FSG[KK01], edge-disjoint path-join method[VGS02]

[IWM00] A. Inokuchi, T. Washio, H. Motoda. *An apriori-based algorithm for mining frequent substructures from graph data*. In PKDD'00, pp.13-23.
[KK01] M. Kuramochi and G. Karypis. *Frequent Subgraph Discovery*. In ICDM'01, pp. 313-320.
[VGS02] N. Vanetik, E. Gudes, S.E. Shimony. *Computing frequent graph patterns from semistructured data*. In ICDM'02, pp. 458-465.

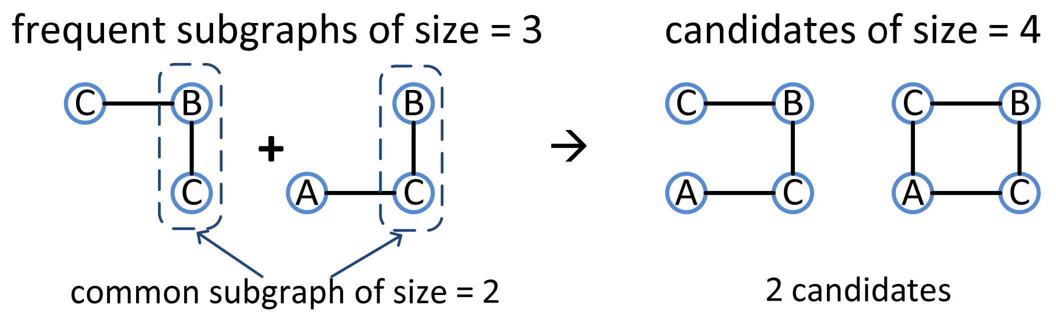
DATABASE SYSTEMS GROUP

AGM Candidate Generation

LMU

- AGM: vertex-based candidate generation:
  - The "size" of a graph *g* is the number of vertices in *V(g)*
  - 2 size-*k* subgraphs are joined iff they share the same size-*(k-1)* subgraph

  Example:



- In each *iteration* potentially a large amount of candidates is generated
  - AGM generates disconnected frequent subgraphs
  - Joining two patterns always just generates 2 candidates, BUT:
  - 1 pattern can have multiple representations
    - Due to the representation of a graph by its adjacency matrix, the common subgraph has no unique representation!

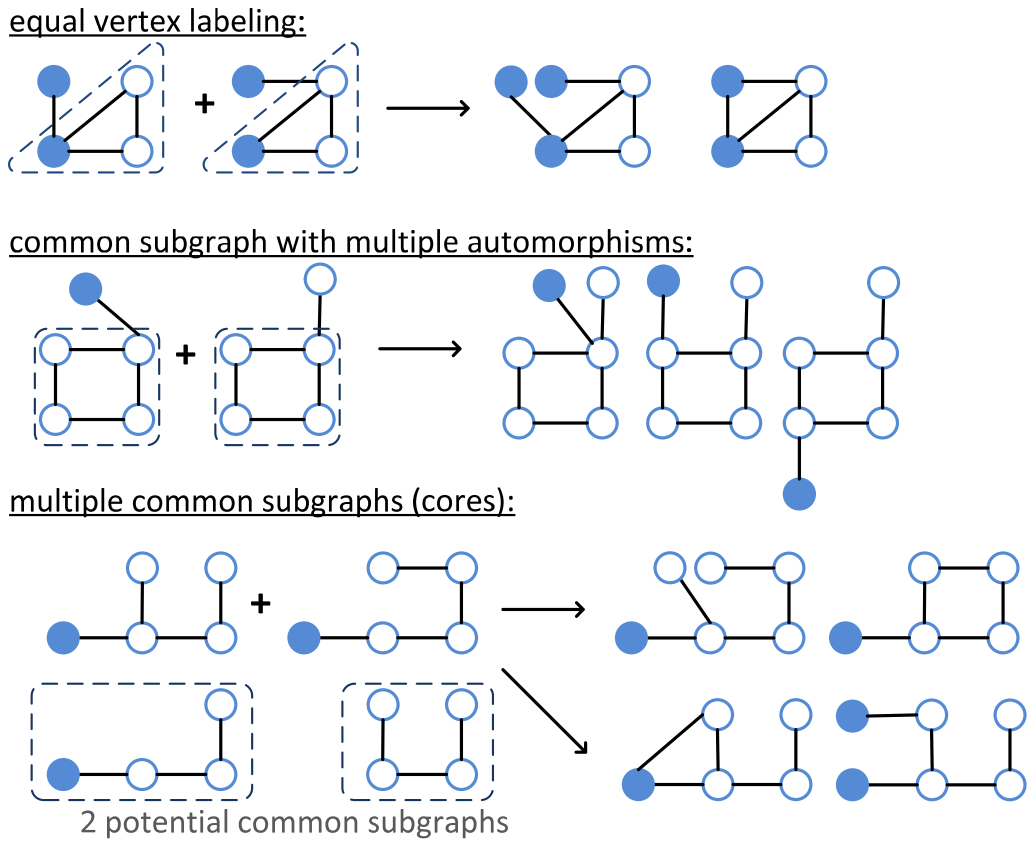
Apriori-based Approach – FSG Candidate Generation

- FSG: edge-based candidate generation:
  - The "size" of a graph *g* is the number of *edges* in E*(g)*
  - 2 size-*k* patterns are joined iff they share the same subgraph having *k*-1 edges

  Example:

  3 reasons for large candidate sets

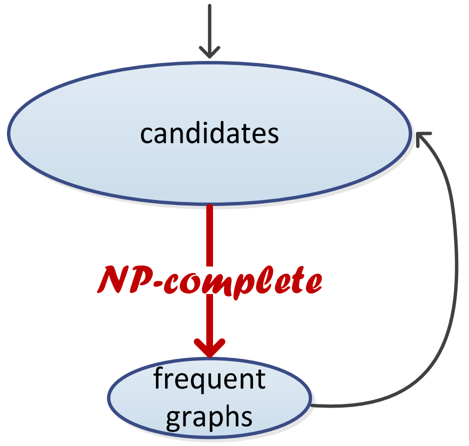
  - Still a potentially large amount of candidates
  - But avoids disconnected frequent subgraphs


equal vertex labeling:
common subgraph with multiple automorphisms:
multiple common subgraphs (cores):
2 potential common subgraphs

Apriori-based Approach vs. Pattern Growth Approach

Apriori-based approach:

- Join methods can be expensive
- Has considerable overhead when size-*k* patterns are joined to generate patterns of size *(k+1)*
- Has to use a breadth-first search (BFS) strategy because of level-wise candidate generation



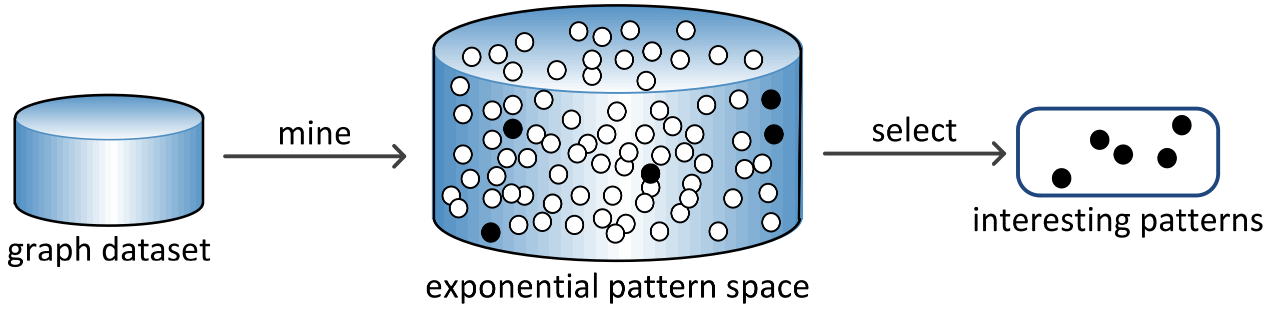
Alternative: pattern growth approach (e.g. *gSpan*[YH02])

- Extends a frequent graph by directly adding a new edge; no expensive joins
- More flexible: can use BFS or DFS
- Critical point: extension of a graph; how to avoid duplicates?

[YH02] X. Yan, J. Han. *gSpan: Graph-based substructure pattern mining.* In ICDM'02, pp.721-724.

Restricted Frequent Subgraphs

- Problem: the mining process often generates a huge number of patterns
  - Anti-monotonicity: a frequent pattern with *n* edges has O(2^n) frequent subgraphs
- Solution: Restrict the frequent patterns based on objective functions
  - Closed subgraphs, maximal subgraphs
  - General constraints (e.g., geometric constraints, density, etc.)
  - Significant graph patterns (e.g., information gain, p-value, G-score, etc.)
- Intuitively:



Summary

- Frequent Subgraph Mining
  - Extension of traditional itemset mining to graph databases
  - Apriori-Methods: Join step is expensive (many duplicates)
  - Pattern-Growth method (gSpan): No duplicates due to DFS code

- Problem of redundancy
  - Set of frequent subgraphs is exponentially large and contains very many similar patterns
  - Solution: Restrict the set of frequent subgraphs
    - Closed, maximal subgraphs
    - Representative subgraphs