InfoLab

IC612: Data Warehousing and Data Mining

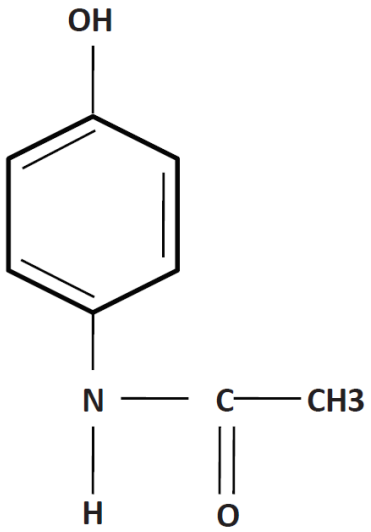
Lecture 11:
Mining Graph Data

Min-Soo Kim

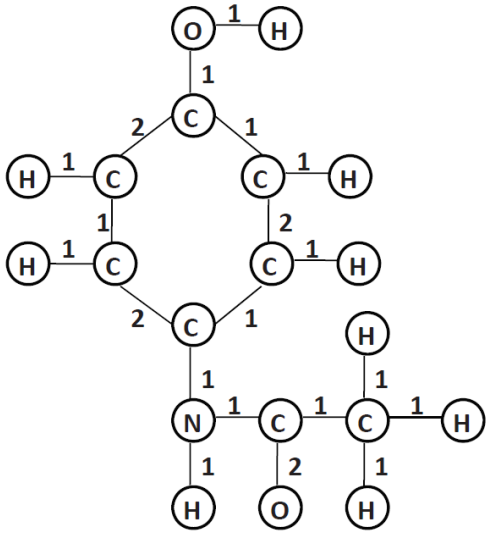
DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Two types of graphs

- **Many small graphs (in chemical and biological data)**
  - each node is associated with a label
  - label may or may not be unique to the node



(a) Acetaminophen



(b) Graph representation

node: atom
edge: bond

- **Single large graph (Web, social networks)**

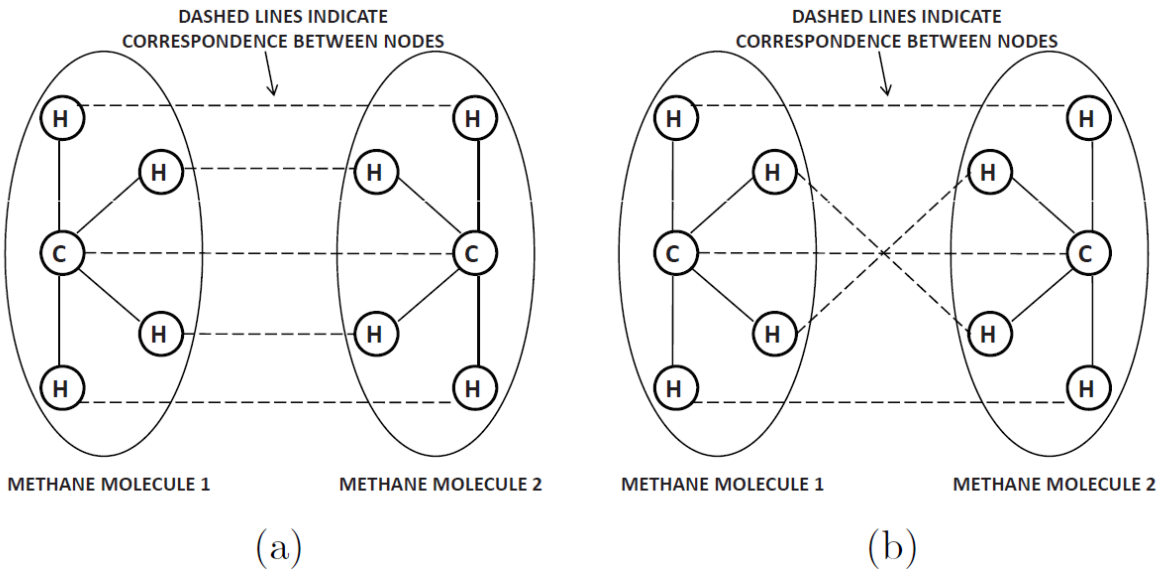
Graph database

Definition 17.1.1 (Graph Database) *A graph database $\mathcal{D}$ is a collection of n different undirected graphs, $G_1 = (N_1, A_1) \ldots G_n = (N_n, A_n)$, such that the set of nodes in the ith graph is denoted by N_i, and the set of edges in the ith graph is denoted by A_i. Each node $p \in N_i$ is associated with a label denoted by $l(p)$.*

- **Complexity 1: lack of the "flat" structure**
- **Complexity 2: repetition of labels among nodes**
  - it leads to problems of **isomorphism** in computing similarity between graphs (NP-hard)
  - both matching and distance computation are fundamental subproblems in graph mining applications

Matching computation in graphs

- **Two graphs match when**
  - a one-to-one correspondence can be established between the nodes of the two graphs
  - the edge presence between corresponding nodes match
  - their labels match
- **Matching graphs are said to be isomorphic**



matched up in 4! = 24 different ways

Definition 17.2.1 (Graph Matching and Isomorphism) *Two graphs $G_1 = (N_1, A_1)$ and $G_2 = (N_2, A_2)$ are isomorphic if and only if a one-to-one correspondence can be found between the nodes of N_1 and N_2 satisfying the following properties:*

1. *For each pair of corresponding nodes $i \in N_1$ and $j \in N_2$, their labels are the same.*

$$l(i) = l(j)$$

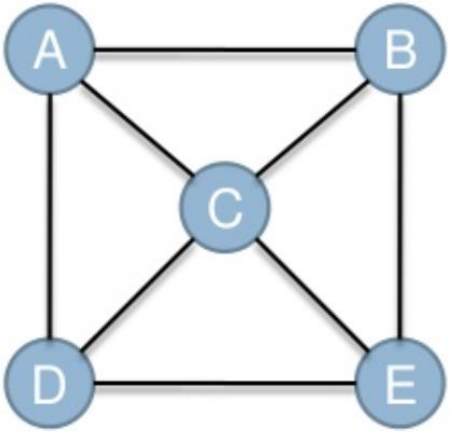
2. *Let $[i_1, i_2]$ be a node-pair in G_1 and $[j_1, j_2]$ be the corresponding node-pair in G_2. Then the edge (i_1, i_2) exists in G_1 if and only if the edge (j_2, j_2) exists in G_2.*

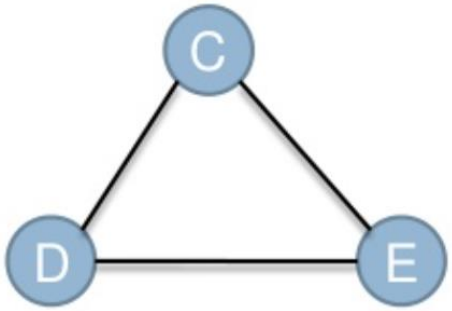
- **Computational challenges in graph matching arise because of the repetition in node labels**
- **For a pair of graphs containing *n* nodes each, the number of possible matchings can be as large as n!**

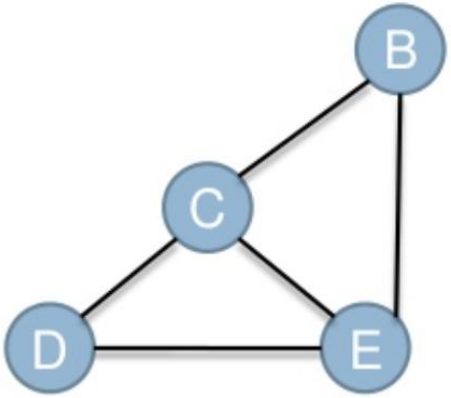
Definition 17.2.2 (Node-Induced Subgraph) *A node-induced subgraph of a graph* $G = (N, A)$ *is a graph* $G_s = (N_s, A_s)$ *satisfying the following properties:*

1. $N_s \subseteq N$
2. $A_s = A \cap (N_s \times N_s)$

In other words, all the edges in the original graph G *between nodes in the subset* $N_s \subseteq N$ *are included in the subgraph* G_s.





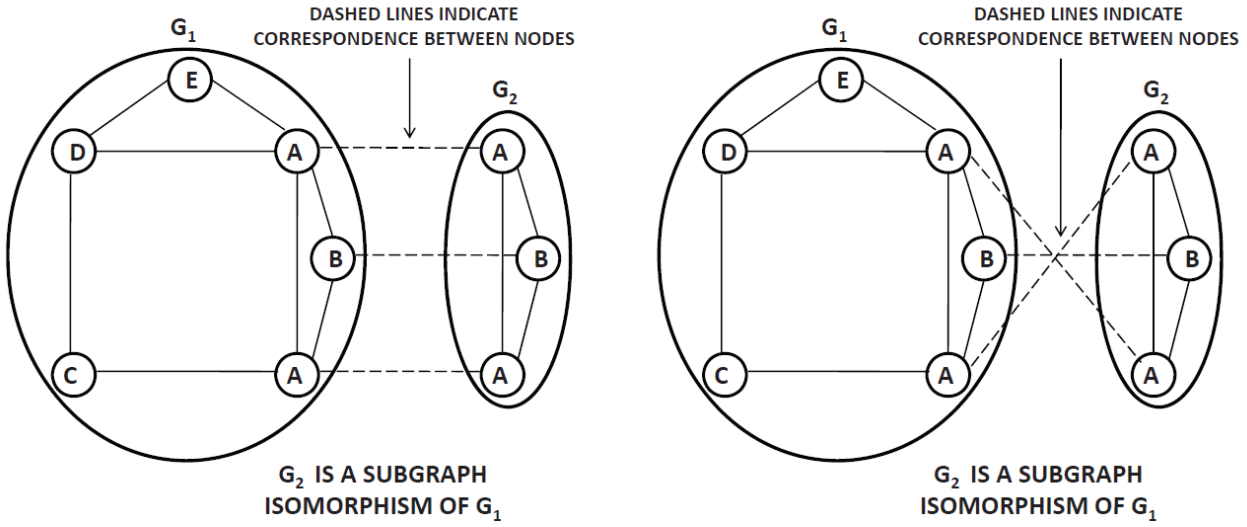


Definition 17.2.3 (Subgraph Matching and Isomorphism) *A query graph $G_q = (N_q, A_q)$ is a subgraph isomorphism of the data graph $G = (N, A)$ if and only if the following conditions are satisfied:*

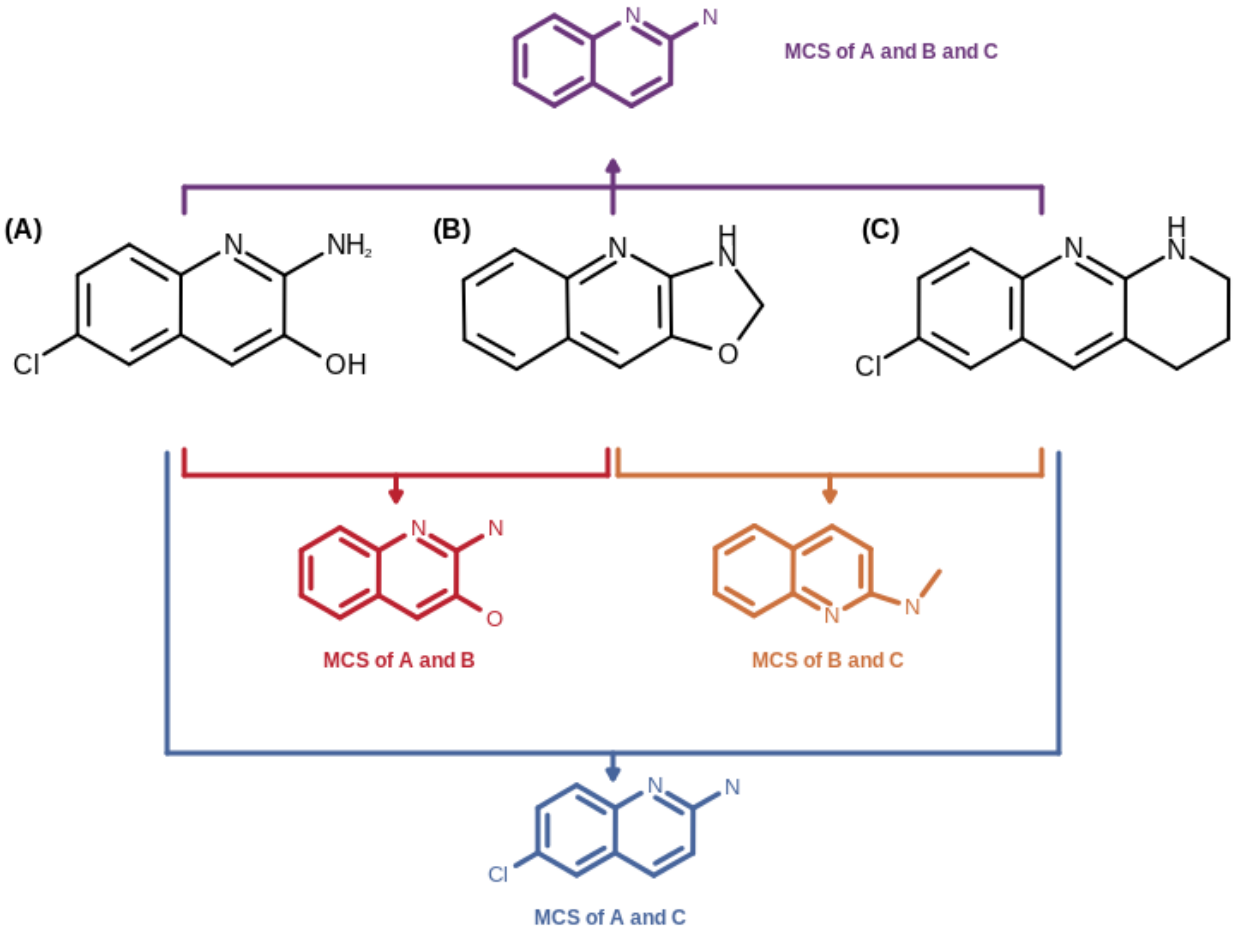
1. *Each node in N_q should be matched to a unique node with the same label in N, but each node in N may not necessarily be matched. For each node $i \in N_q$, there must exist a unique matching node $j \in N$, such that their labels are the same.*

$$l(i) = l(j)$$

2. *Let $[i_1, i_2]$ be a node-pair in G_q, and let $[j_1, j_2]$ be the corresponding node-pair in G, based on the matching discussed above. Then, the edge (i_1, i_2) exists in G_q if and only if the edge (j_1, j_2) exists in G.*



Definition 17.2.4 (Maximum Common Subgraph) *A maximum common subgraph between a pair of graphs $G_1 = (N_1, A_1)$ and $G_2 = (N_2, A_2)$ is a graph $G_0 = (N_0, A_0)$ that is a subgraph isomorphism of both G_1 and G_2, and for which the size of the node set N_0 is as large as possible.*



Ullman's algorithm for subgraph isomorphism

- **Determine all possible subgraph isomorphisms between a query graph and a data graph**

$$\mathcal{M} = \{(i_1^q, i_1), (i_2^q, i_2), \ldots (i_m^q, i_m)\}$$

Algorithm *SubgraphMatch*(Query Graph: G_q, Data Graph: G,
Current Partially Matched Node Pairs: $\mathcal{M}$)
begin
if ($|\mathcal{M}| = |N_q|$) **then return** successful match $\mathcal{M}$;
(Case when $|\mathcal{M}| < |N_q|$**)**
$\mathcal{C}$ = Set of all label matching node pairs from (G_q, G) not in $\mathcal{M}$;
Prune $\mathcal{C}$ using heuristic methods; **(Optional efficiency optimization)**
for each pair $(i_q, i) \in \mathcal{C}$ **do**
if $\mathcal{M} \cup \{(i_q, i)\}$ is a valid partial matching
then *SubgraphMatch*$(G_q, G, \mathcal{M} \cup \{(i_q, i)\})$;
endfor
end

- The value of the matching set parameter *M* is initialized to the empty set at the top-level recursive call
- The number of matched nodes in *M* is exactly equal to the depth of the recursive call
- The recursion backtracks when either the subgraphs cannot be further matched or when G_q has been fully matched
- The procedure has exponential complexity in terms of its input size (especially, the query graph size)

Maximum common subgraph (MCG) problem

Algorithm MCG(Graphs: G_1, G_2, Current Partially Matched Pairs: $\mathcal{M}$, Current Best Match: $\mathcal{M}_{best}$)
begin
  $\mathcal{C}$ = Set of all label matching node pairs from (G_1, G_2) not in $\mathcal{M}$;
  Prune $\mathcal{C}$ using heuristic methods; **(Optional efficiency optimization)**
  for each pair $(i_1, i_2) \in \mathcal{C}$ **do**
    if $\mathcal{M} \cup \{(i_1, i_2)\}$ is a valid matching
      then $\mathcal{M}_{best} = MCG(G_1, G_2, \mathcal{M} \cup \{(i_1, i_2)\}, \mathcal{M}_{best})$;
  endfor
  if $(|\mathcal{M}| > |\mathcal{M}_{best}|)$ **then return**$(\mathcal{M})$ **else return**$(\mathcal{M}_{best})$;
end

- Both *M* and M_{best} are initialized to *null* in the initial call
- Largest common subgraph found so far is tracked in M_{best}

Distance computation for graph

- **Metric: nonnegative, symmetric, triangle inequality**
- **Maximum common subgraph-based distances**
  - Unnormalized non-matching measure: number of non-matching nodes between the two graphs

$$U(G_1, G_2) = |G_1| + |G_2| - 2 \cdot |MCS(G_1, G_2)|$$

  - Union-normalized distance: range of (0, 1)

$$UDist(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{|G_1| + |G_2| - |MCS(G_1, G_2)|}$$

  - Max-normalized distance : range of (0, 1)

$$MDist(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{\max\{|G_1|, |G_2|\}}$$

Graph edit distance

- analogous to the string edit distance

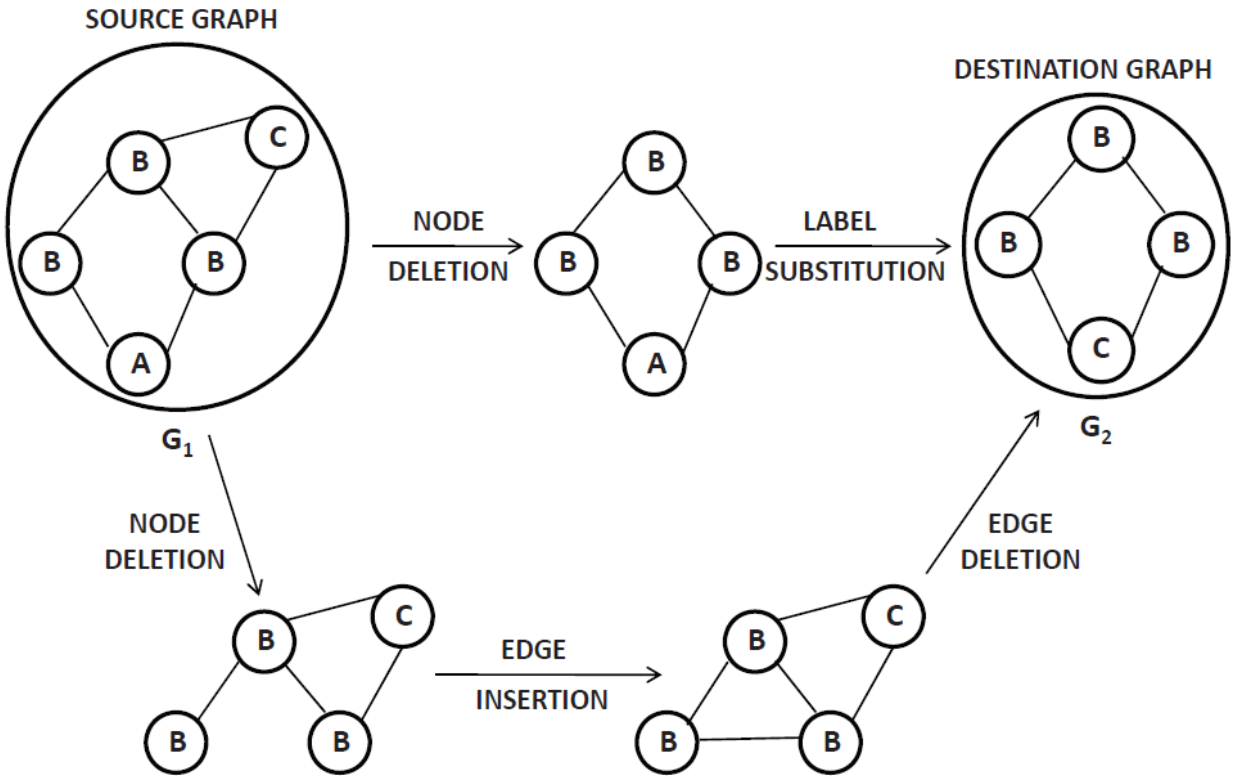
- edit operations are specific to the graph domain: edit distances can be applied to the nodes, the edges, or the labels
  a. the insertion of nodes
  b. the deletion of nodes (includes automatic deletion of all its incident edges)
  c. the label substitution of nodes
  d. the insertion of edges
  e. the deletion of edges

- problem of learning edit costs is a challenging issue

- edit distance is not necessarily symmetric
  - Edit(G_1,G_2) can be different from Edit(G_2,G_1)

Edit distance computation for graphs is NP-hard

- edit distance can be shown to be equivalent to distance measures based on the maximum common subgraph
- edit distance can be viewed as the cost of an error-tolerant graph isomorphism (“errors” are quantified in the cost of edit operations)
  - edit-distance computation for strings and sequences can be solved polynomially using dynamic programming



Edit distance algorithm

- Edit-distance algorithm maintains a series of edits ε that are the operations to be applied to graph G_1 to transform it into a subgraph isomorphism G'_1 of the graph G_2
  - e.g.,

$$\mathcal{E} = \text{Delete}(i_1),\ \text{Insert}(i_2, i_5),\ \text{Label-Substitute}(i_4, A \Rightarrow C),\ \text{Delete}(i_2, i_6)$$

- Value of ε is *null* at the beginning of the algorithm
- New edits are appended to ε in each recursive call
- Value of the parameter ε_{best} at the top-level call
  - trivial sequence of edit operations: delete all nodes of G_1 and then add all nodes and edges of G_2

Optimal edit distance

Algorithm *EditDistance*(Graphs: G_1, G_2, Current Partial Edit Sequence: $\mathcal{E}$, Best Known Edit Sequence: $\mathcal{E}_{best}$)
begin
  if (G_1 is subgraph isomorphism of G_2) **then begin**
    Add insertion edits to $\mathcal{E}$ that convert G_1 to G_2;
    return($\mathcal{E}$);
  end;
  $\mathcal{C}$ = Set of all possible edits to G_1 excluding node-insertions;
  Prune $\mathcal{C}$ using heuristic methods; **(Optional efficiency optimization)**
  for each edit operation $e \in \mathcal{C}$ **do**
  begin
    Apply e to G_1 to create G_1';
    Append e to $\mathcal{E}$ to create $\mathcal{E}'$;
    $\mathcal{E}_{current} = EditDistance(G_1', G_2, \mathcal{E}', \mathcal{E}_{best})$;
    if $(Cost(\mathcal{E}_{current}) < Cost(\mathcal{E}_{best}))$ **then** $\mathcal{E}_{best} = \mathcal{E}_{current}$;
  endfor
  return($\mathcal{E}_{best}$);
end

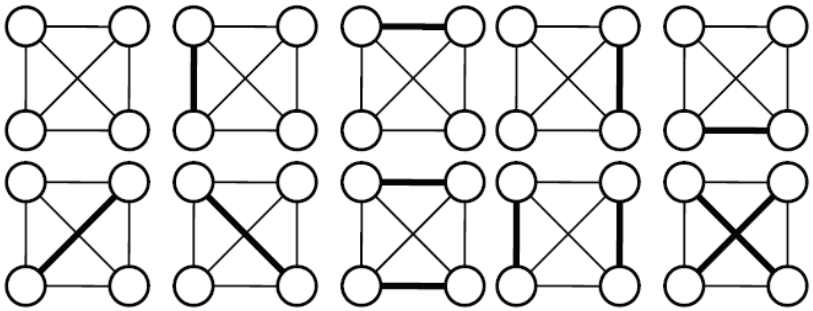
Transformation-based distance

- **Graph edit distance is computationally impractical**
  - use heuristic and kernel-based methods to transform the graphs into a space in which distance computations are more efficient

- **Topological Descriptors**
  - convert structural graphs to multidimensional data
    - use quantitative measures of important structural characteristics as dimensions
    - highly domain-specific
  - use multidimensional data mining algorithms on the transformed representation
  - drawback: structural information is lost

Examples of topological descriptors

- **Morgan index**
- **Wiener index**
- **Hosoya index (Z-index)**
  - the number of valid pairwise node matchings in the graph



Hosoya index: 10
for a clique of four nodes

- **Estrada index**
- **Circuit rank**
- **Randic index**

Kernel-based transformations

- **Can be used directly with SVM classifiers**

- **$K(G_i, G_j)$**
  - kernel similarity between a pair of graphs G_i and G_j
  - dot product of the two graphs after hypothetically transforming them to a new space by the function $\Phi(\cdot)$

$$K(G_i, G_j) = \Phi(G_i) \cdot \Phi(G_j)$$

  - value of $\Phi(\cdot)$ is defined indirectly in terms of the kernel similarity function $K(\cdot, \cdot)$

- **Kernels: random-walk kernels, shortest-path kernels**

Random-walk kernels

- **Compare the label sequences induced by random walks in the two graphs**
  - two graphs are similar if many sequences of labels created by random walks between pairs of nodes are similar as well
- **Computational challenge: exponential number of possible random walks**
  - define a primitive kernel function k(s1, s2) between a pair of node sequences s1 (from G1) and s2 (from G2)

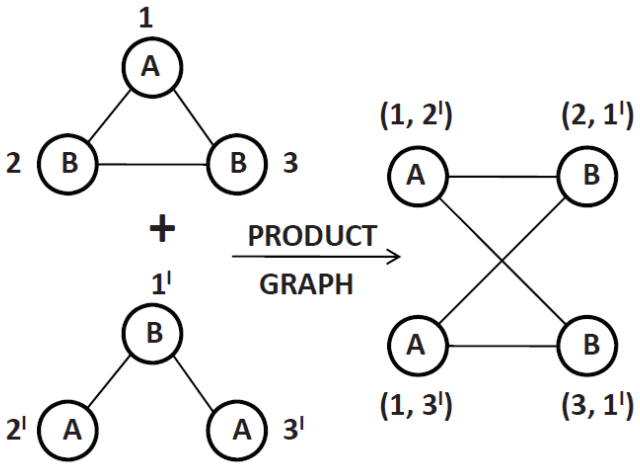
$$k(s_1, s_2) = I(s_1 = s_2)$$

  - K(G1,G2) : defined as the sum of the probabilities of all the primitive sequence kernels over all possible walks

$$K(G_1, G_2) = \sum_{s_1, s_2} p(s_1|G_1) \cdot p(s_2|G_2) \cdot k(s_1, s_2)$$

Product graph

- **Random-walk kernel is computed using the notion of a product graph G_X between G_1 and G_2**



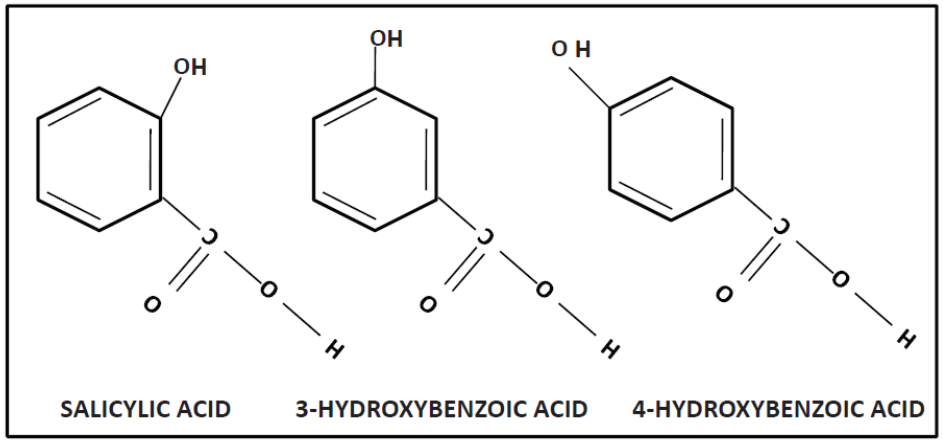
- $\overline{e}$: $|GX|$-dimensional column vector of 1s
- $\lambda \in (0, 1)$: a discount factor

$$K(G_1, G_2) = \sum_{i,j} \sum_{k=1}^{\infty} \frac{\lambda^k}{k!} [A^k]_{ij} = \overline{e}^T \exp(\lambda A) \overline{e}$$

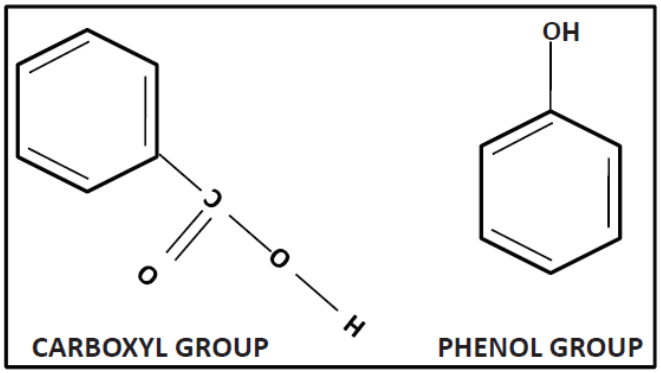
$O(n^6)$

Frequent subgraph mining

- **Fundamental building block for graph mining algorithms**
  - many of the clustering, classification, and similarity search techniques use frequent graph mining as an intermediate step
  - e.g., phenolic acids (a family of organic compounds with similar chemical properties)
    - many complex variations of this family act as signaling molecules and agents of defense in plants



DATABASE OF PHENOLIC ACIDS



FREQUENT SUBSTRUCTURES OF PHENOLIC ACIDS

- **Definition of a frequent subgraph**
  - identical to the case of association pattern mining
  - except that a subgraph relationship is used to count the support rather than a subset relationship

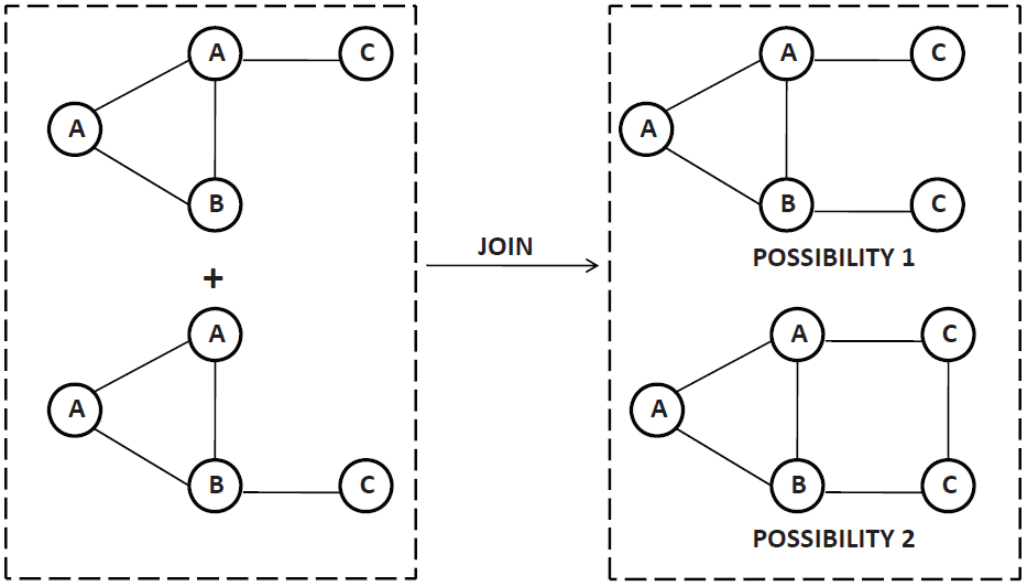
- **Many well-known frequent substructure mining algorithms are based on the enumeration tree principle**
  - joins are used to create candidate patterns of size (k + 1) from frequent patterns of size k
  - candidate frequent patterns can be generated by either node extensions or edge extensions
    - how frequent substructures of size k are defined
    - how they are joined together to create candidate of size (k + 1)

Frequent subgraph mining algorithm

Algorithm *GraphApriori*(Graph Database: $\mathcal{G}$,
Minimum Support: *minsup*);
begin
$\mathcal{F}_1 = \{$ All Frequent singleton graphs $\}$;
$k = 1$;
while $\mathcal{F}_k$ is not empty **do begin**
Generate $\mathcal{C}_{k+1}$ by joining pairs of graphs in $\mathcal{F}_k$ that
share a subgraph of size $(k-1)$ in common;
Prune subgraphs from $\mathcal{C}_{k+1}$ that violate downward closure;
Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \mathcal{G})$ and retaining
subgraphs from $\mathcal{C}_{k+1}$ with support at least *minsup*;
$k = k + 1$;
end;
return$(\cup_{i=1}^{k} \mathcal{F}_i)$;
end

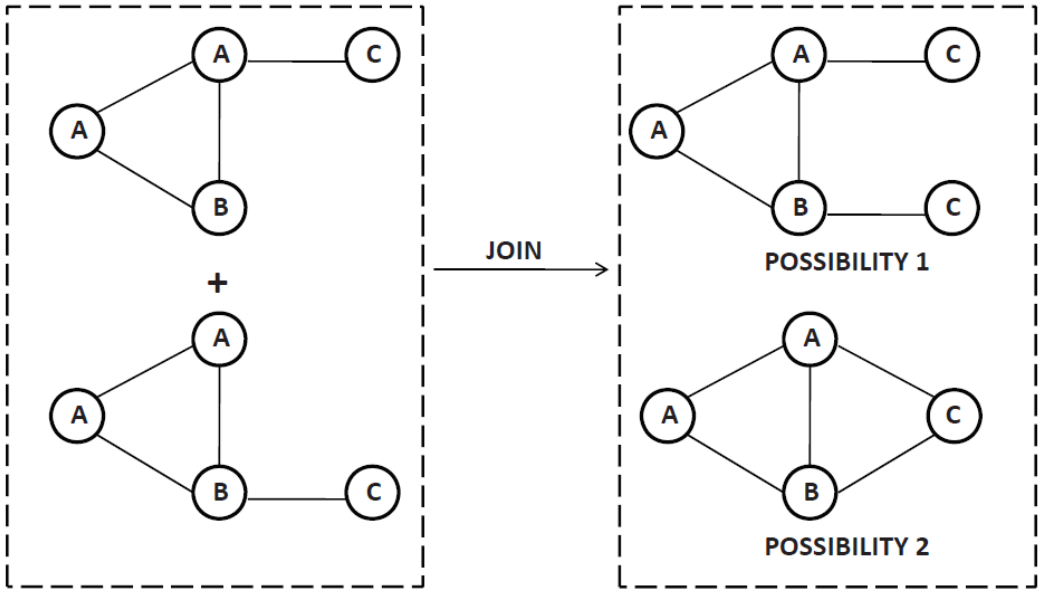
Node-based join growth

- **Core**
  - for two graphs from F_k to be joined, a matching subgraph with (k−1) nodes must exist between the two graphs
- **Ambiguity exists when node-based join**
- **All possible (k − 1) common subgraphs need to be discovered to generate the candidates completely**



Edge-based join growth

- **Resulting candidate will contain exactly (k +1) edges**
  - the number of nodes in the candidate may not necessarily be greater than that in the individual subgraphs that are joined
- **Edge-based join growth tends to generate fewer candidates**
  - therefore generally more efficient



InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology