UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

MASTER IN COMPUTER SCIENCE

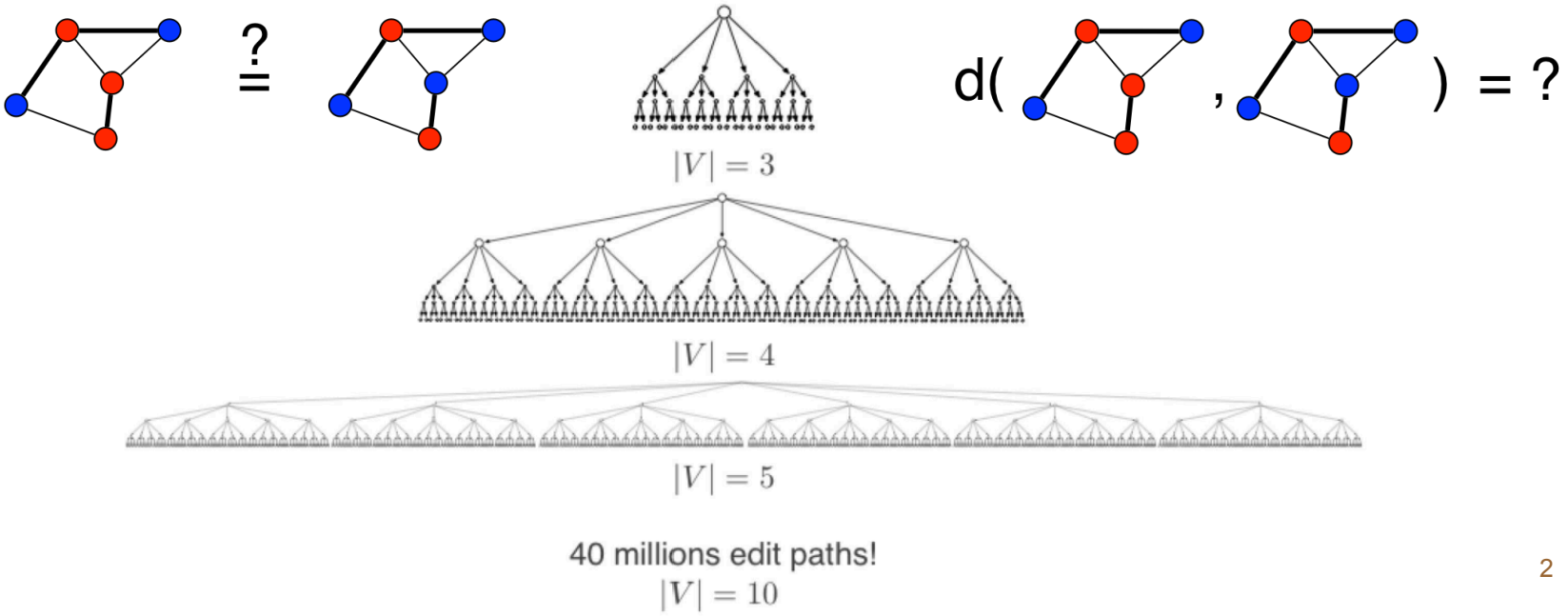
Pattern Recognition

Lecture 10 : Graph Matching II

Dr. Andreas Fischer
andreas.fischer@unifr.ch

Computational Complexity

- The high representational power of graphs comes at the cost of a high computational complexity for many matching algorithms.
- Efficient algorithms are needed:
  - for matching large graphs
  - for matching a large number of graphs
- Either improve the optimal methods or find approximate, suboptimal solutions to perform pattern recognition.



Exact Matching

Common Substructures

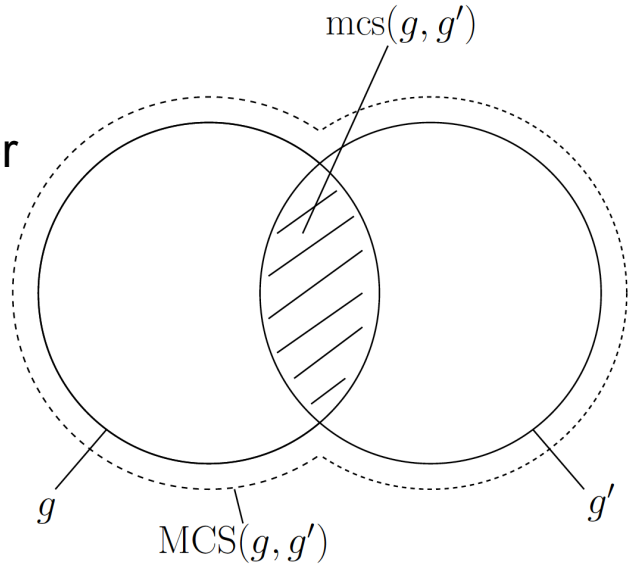
- Several metrics can be defined based on the maximum common subgraph (mcs) and the minimum common supergraph (MCS).
  - $d_1(g,g')$ = 0 ⇔ g and g' are isomorph; $d_1(g,g')$ = 1 ⇔ mcs(g,g') = ∅

$$d_1(g,g') = 1 - \frac{|mcs(g,g')|}{\max(|g|,|g'|)};\ 0 \le d_1(g,g') \le 1$$

  - $d_2(g,g')$ also takes into account the smaller graph for normalization.

$$d_2(g,g') = 1 - \frac{|mcs(g,g')|}{MCS(g,g')};\ 0 \le d_2(g,g') \le 1$$

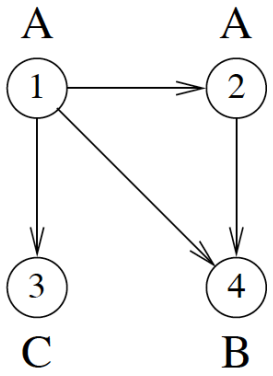
  - $d_3(g,g')$ corresponds to the symmetric difference of the graphs.



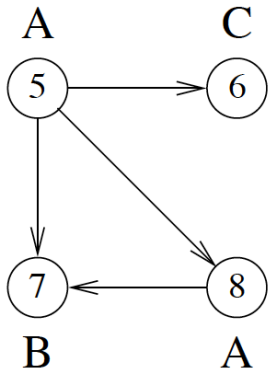
$$d_3(g,g') = |MCS(g,g')| - |mcs(g,g')|;\ 0 \le d_3(g,g') \le |g| + |g'|$$

Example

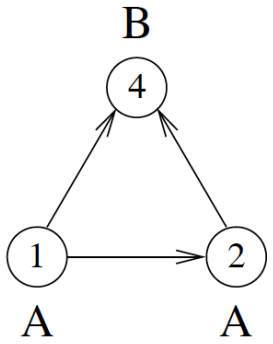
g_1 :



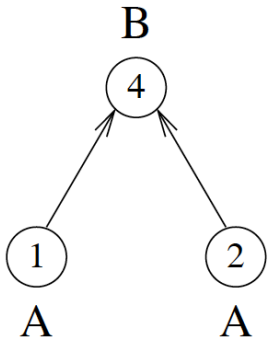
g_2 :



g_3 :



g_4 :



$$d_1(g_1, g_4) = 1 - \frac{2}{4} = \frac{1}{2}$$
$$d_2(g_1, g_4) = 1 - \frac{2}{5} = \frac{3}{5}$$
$$d_3(g_1, g_4) = 5 - 2 = 3$$

$$d_1(g_1, g_2) = d_2(g_1, g_2) = d_3(g_1, g_2) = 0$$

$$d_1(g_1, g_3) = 1 - \frac{3}{4} = \frac{1}{4}$$
$$d_2(g_1, g_3) = 1 - \frac{3}{4} = \frac{1}{4}$$
$$d_3(g_1, g_3) = 4 - 3 = 1$$

Computation of mcs and MCS

- Repetition:
  - Perform an exhaustive search via node assignments.
  - Find a maximum clique in the association graph.
  - |MCS(g,g')| = |g| + |g'| - |mcs(g,g')|
- Another alternative is to compute the graph edit distance with a special cost function:

$$
\begin{aligned}
c_{ns}(x) &= \left\{ \begin{array}{ll} 0, & \text{if } \alpha_1(x) = \alpha_2(f(x)), \\ \infty, & \text{otherwise,} \end{array} \right\} \text{for any } x \in \hat{V}_1, \\
c_{nd}(x) &= 1 \quad \text{for any } x \in V_1 - \hat{V}_1, \\
c_{ni}(x) &= 1 \quad \text{for any } x \in V_2 - \hat{V}_2, \\
c_{es}(e) &= \left\{ \begin{array}{ll} 0, & \text{if } \beta_1((x,y)) = \beta_2((f(x),f(y))), \\ \infty, & \text{otherwise,} \end{array} \right\} \text{for any } e = (x,y) \in \hat{E}_1, \\
c_{ed}(e) &= 0 \quad \text{for any } e = (x,y) \in E_1 - \hat{E}_1, \\
c_{ei}(e) &= 0 \quad \text{for any } e = (x,y) \in E_2 - \hat{E}_2.
\end{aligned}
$$

  - With this cost function: d(g,g') = |g| + |g'| - 2|mcs(g,g')|

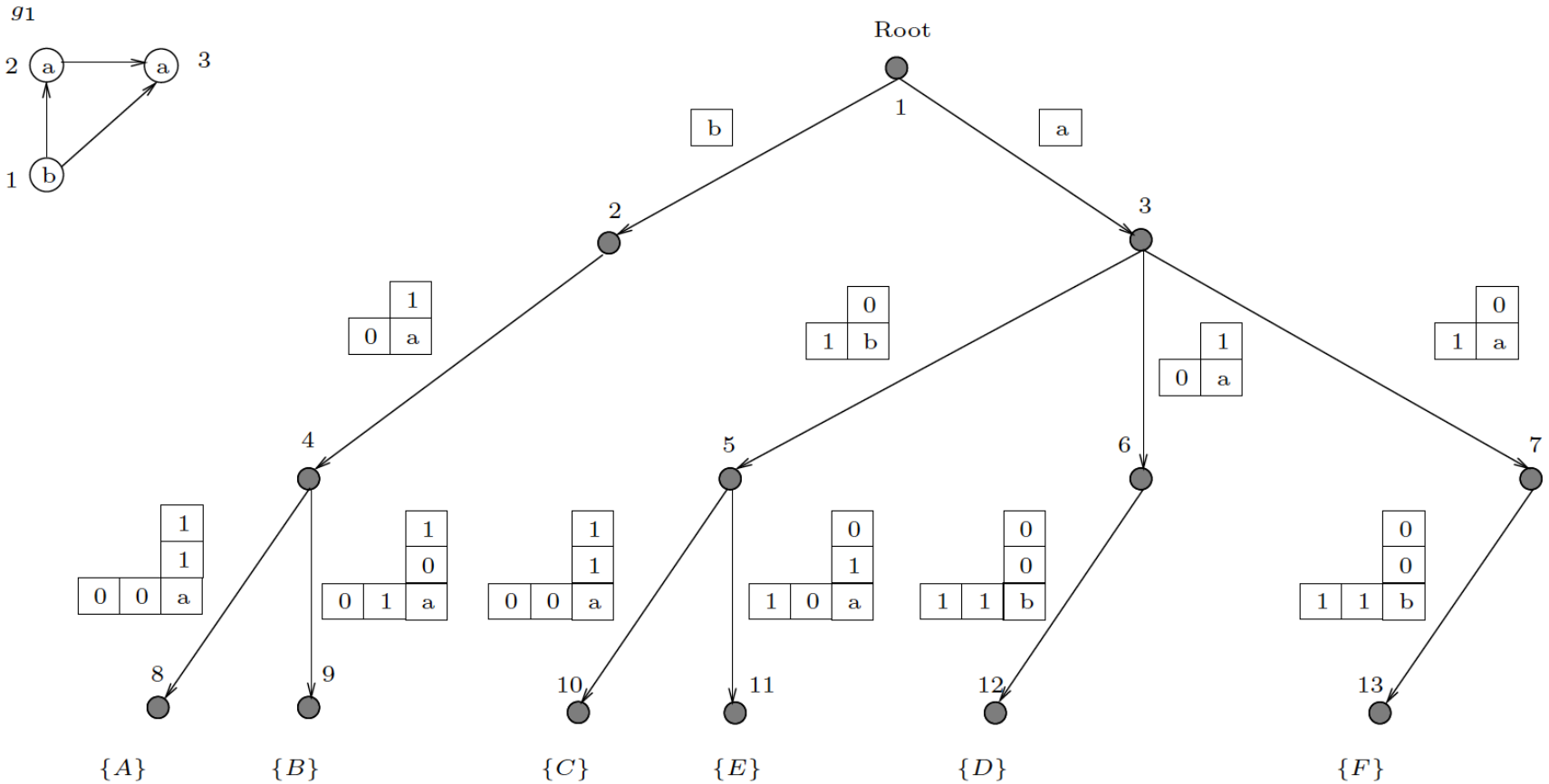
Fast Exact Matching

- The common substructure metrics can be used to classify an input graph, *e.g.* using KNN.
- Match large graphs:
  - Repetition: future match tables can speedup an exhaustive search for (sub)graph isomorphisms between two large graphs.
- Match a large number of graphs:
  - To compare the input with a database, the matching complexity is increased by an additional factor, *i.e.* the size of the database.
  - This overhead can be reduced with preprocessing methods that are applied off-line before the recognition stage.

$$g \leftrightarrow \{g_1, \ldots, g_L\}$$

Decision Tree Approach

- Question: is the input graph a subgraph of some database graphs?
- Off-line: store permutations of the adjacency matrices in a tree.
- On-line: compare the input graph with the tree.



Adjacency and Permutation Matrices

- For |V| = n, the *adjacency matrix* A = a_{ij} is an n x n matrix:
  - a_{ij} = 1 if there is an edge (v_i,v_j); i ≠ j
  - a_{ij} = 0 otherwise; i ≠ j
  - a_{ii} = α(v_i) in this application
- The *permutation matrix* P = p_{ij} is an n x n matrix:
  - all elements are either 0 or 1
  - each row contains exactly one 1
  - each column contains exactly one 1
- The permuted adjacency matrix B = PAP' represents the same graph. If p_{ij} = 1, the node v_j in A is the node v_i in B.
  - The question if g_1 and g_2 are isomorph is equivalent to the question if there exists a permutation matrix P such that $A_1 = PA_2P'$.
  - The question if g_1 is a subgraph of g_2 is equivalent to the question if there exists a permutation matrix P such that A_1 is the left upper part of the matrix PA_2P'.

Example

- Store permutations of the first database graph.

A

1	2	3
b	1	1
0	a	1
0	0	a

B

1	3	2
b	1	1
0	a	0
0	1	a

C

2	1	3
a	0	1
1	b	1
0	0	a

D

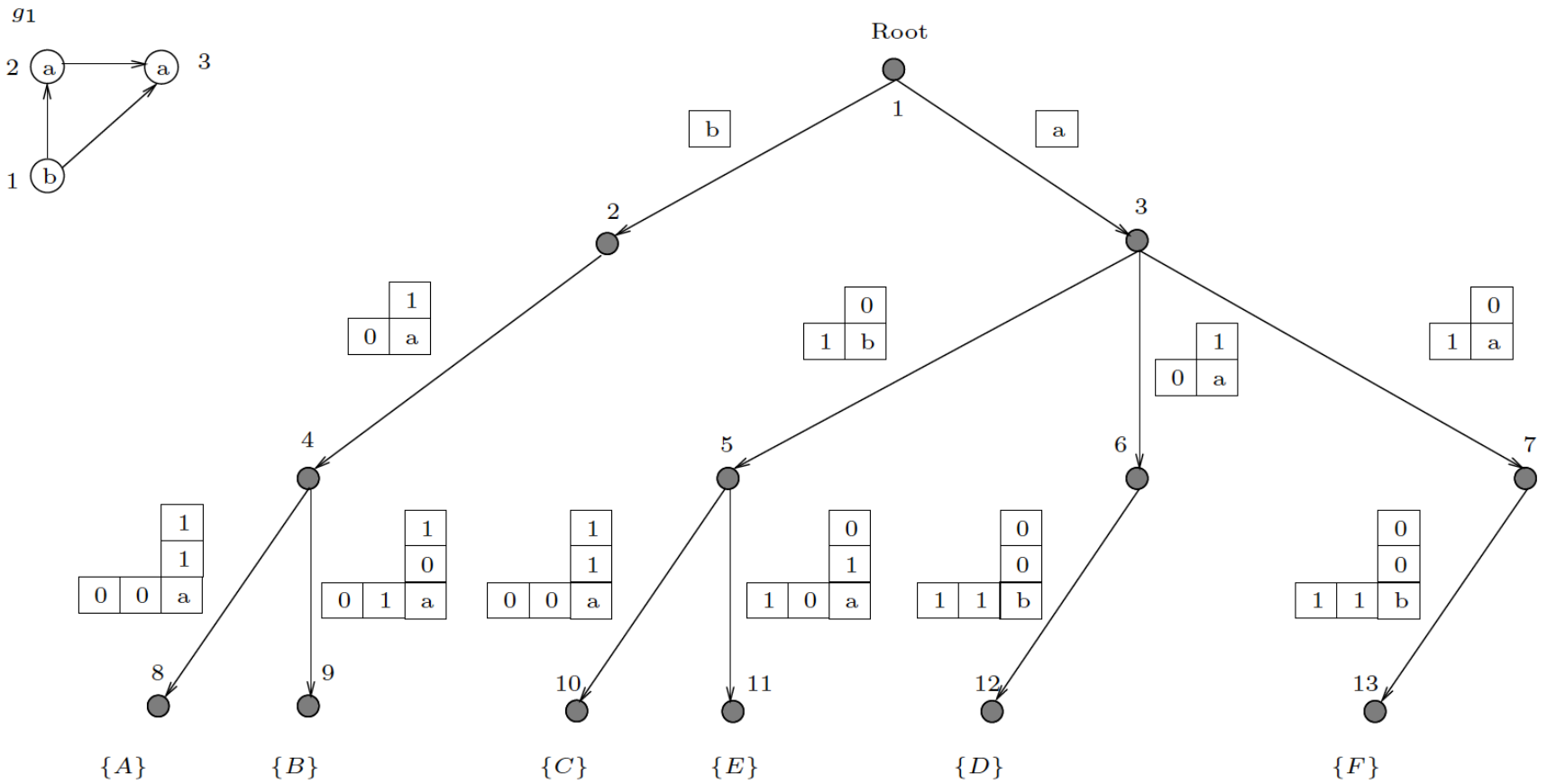
2	3	1
a	1	0
0	a	0
1	1	b

E

3	1	2
a	0	0
1	b	1
1	0	a

F

3	2	1
a	0	0
1	a	0
1	1	b



Example

- Integrate permutations of the other database graphs.

A'

1	2	3
a	1	0
0	a	1
1	0	a

B'

1	3	2
a	0	1
1	a	0
0	1	a

C'

2	1	3
a	0	1
1	a	0
0	1	a

D'

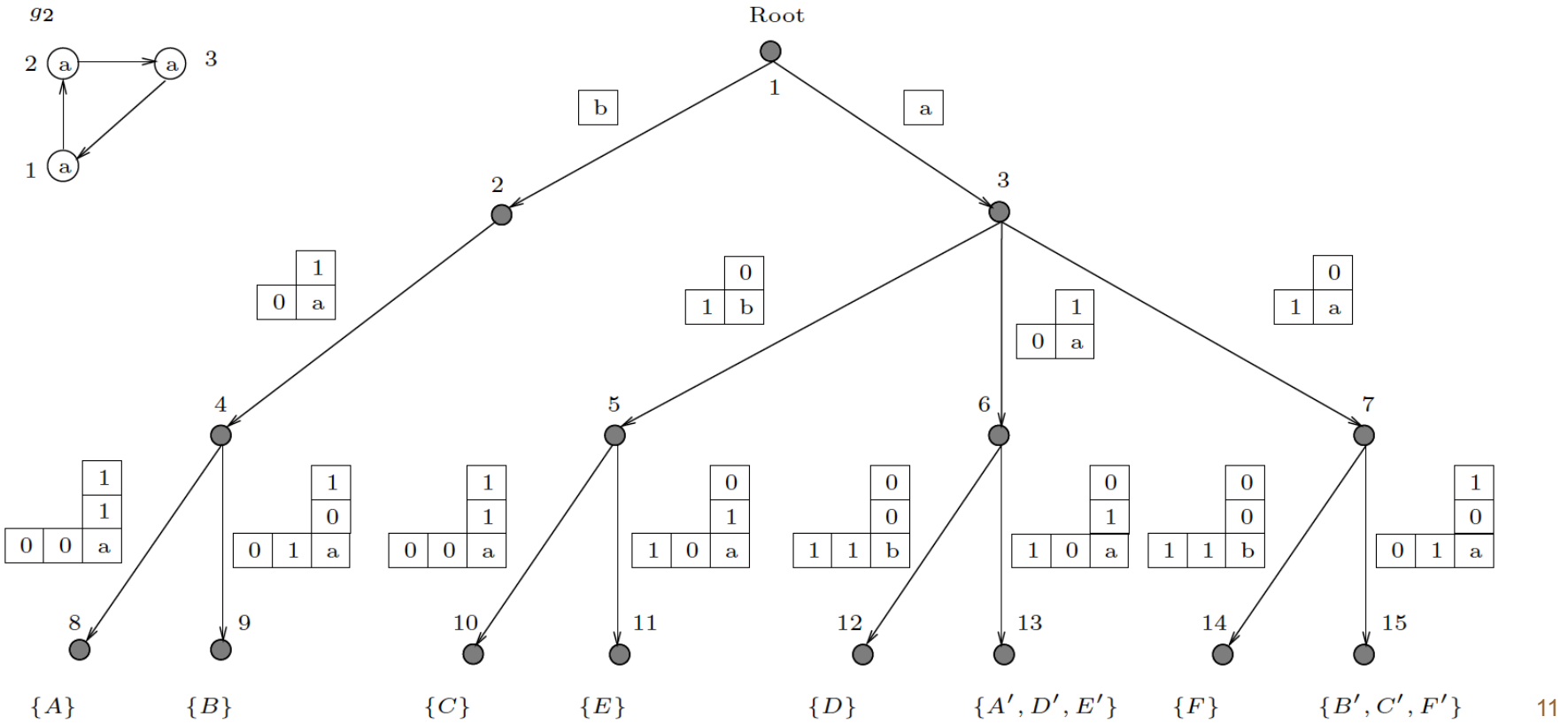
2	3	1
a	1	0
0	a	1
1	0	a

E'

3	1	2
a	1	0
0	a	1
1	0	a

F'

3	2	1
a	0	1
1	a	0
0	1	a



Decision Tree Complexity

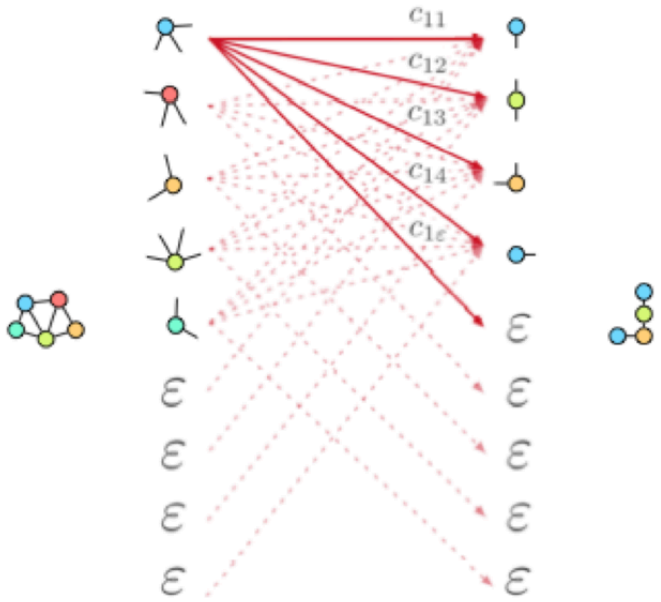
- Tree matching (on-line): the adjacency matrix of the input graph is matched top-to-bottom with the tree:
  - Input graph is completely matched and a leaf node is reached: the input graph is isomorph to the graph of this leaf node.
  - Input graph is completely matched and a non-leaf node is reached: the input graph is a subgraph of all successor graphs of this node.
  - Input graph cannot be matched completely: there is no (sub)graph isomorphism between the input graph and the database graphs.
- Complexity:
  - L - number of graphs in the database
    n - number of nodes in the input graph
    m - number of nodes in the largest database graph
  - Time: $O(n^2)$ - independent of m and L
  - Space: $O(Lm^m)$ - limitation of the procedure

Inexact Matching

Approximation of Graph Edit Distance

- For matching large graphs and/or a large number of graphs, approximate solutions for graph edit distance have been proposed.
- In the following, an established method based on bipartite graph matching (BP) is presented.
- It reduces edit distance to an assignment problem between local substructures, which can be solved efficiently in cubic time.
- A valid but not necessarily optimal edit path is obtained, hence the BP approximation is an upper bound of the true edit distance.

$$BP(g_1, g_2) \geq d(g_1, g_2)$$



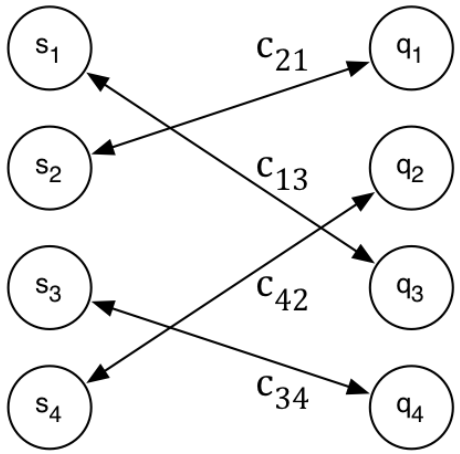
Assignment Problem

- Given two disjoint sets of equal size S = {s_1,…,s_n} and Q = {q_1,…,q_n} and a square n x n cost matrix C = c_{ij}, where c_{ij} is the cost of assigning the ith element of S to the jth element of Q. Then the *assignment problem*, also known as linear sum assignment problem (LSAP), is to find the minimum cost permutation

$$(\phi_1,\ldots,\phi_n) = \underset{(\phi_1,\ldots,\phi_n)\in S_n}{\operatorname{argmin}} \sum_{i=1}^{n} c_{i\phi_i}$$

  of the integers (1,…,n) where S_n is the set of all n! possible permutations.

- Can be solved in $O(n^3)$, for example based on the Hungarian algorithm.



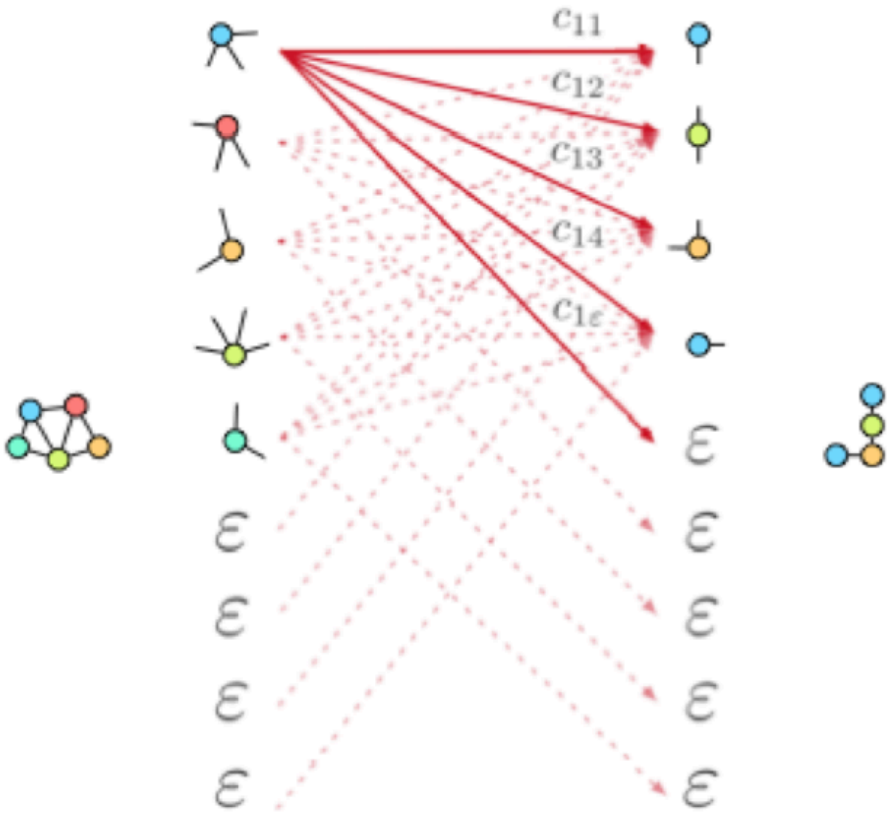
$(\phi_1,\phi_2,\phi_3,\phi_4) = (3,1,4,2)$

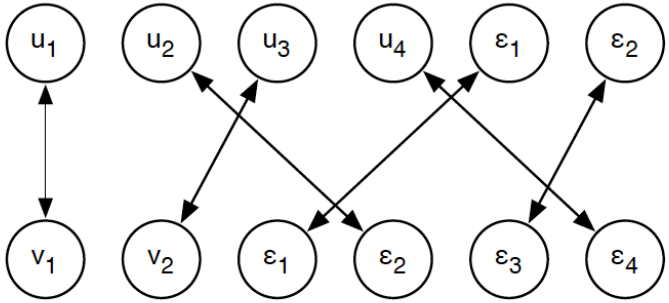
Stating GED as an LSAP

- Three major issues:
  1. LSAPs are stated for sets with equal cardinality. However, graphs do not have the same size in general.
  2. LSAPs are stated for substitutions only. However, graph edit distance also allows deletions and insertions.
  3. Graphs are not only sets of nodes but also contain edges that represent structural relationships.
- The first two issues are perfectly resolvable by means of BP approximation, the third issue only partially.
- Idea: match local substructures consisting of nodes and their adjacent edges. Add epsilon nodes for deletion and insertion, respectively.

Bipartite Graph Matching (BP)

- The matching graph is *bipartite*, that is a graph with two distinct sets of nodes, where all edges connect the two sets.





Cost Matrix

- The (n+m) x (n+m) cost matrix has four parts:
  1. Upper left: substitutions (nodes plus adjacent edges)
  2. Upper right: deletions (nodes plus adjacent edges)
  3. Lower left: insertions (nodes plus adjacent edges)
  4. Lower right: dummy assignments ($\epsilon \rightarrow \epsilon$)

$$\mathbf{C} = \left[\begin{array}{cccc|cccc} c_{11} & c_{12} & \cdots & c_{1m} & c_{1\varepsilon} & \infty & \cdots & \infty \\ c_{21} & c_{22} & \cdots & c_{2m} & \infty & c_{2\varepsilon} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\ c_{n1} & c_{n2} & \cdots & c_{nm} & \infty & \cdots & \infty & c_{n\varepsilon} \\ \hline c_{\varepsilon 1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\varepsilon 2} & \ddots & \vdots & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\varepsilon m} & 0 & \cdots & 0 & 0 \end{array}\right]$$

Cost Matrix Entries

- Deletion costs include the deletion of node $u_i \in V_1$ as well as the deletion of all edges P_i adjacent to u_i:

$$c_{i\varepsilon} = c(u_i \rightarrow \varepsilon) + \sum_{p \in P_i} c(p \rightarrow \varepsilon)$$

- Insertion costs include the insertion of node $v_j \in V_2$ as well as the insertion of all edges Q_j adjacent to v_j:

$$c_{\varepsilon j} = c(\varepsilon \rightarrow v_j) + \sum_{q \in Q_j} c(\varepsilon \rightarrow q)$$

- Substitution costs include the node substitution ($u_i \rightarrow v_j$) as well as an estimation of the edge assignment cost $C(P_i \rightarrow Q_j)$:

$$c_{ij} = c(u_i \rightarrow v_j) + C(P_i \rightarrow Q_j)$$

Edge Assignment Cost

- The edge assignment cost $C(P_i \rightarrow Q_j)$ can only be estimated because *a priori* there is no information about node mappings, from which edge edit costs could be derived.
- An optimistic estimation is computed solving another assignment problem with an $(|P_i| + |Q_j|) \times (|P_i| + |Q_j|)$ cost matrix $D = d_{ij}$:
  - $d_{i\varepsilon} = c(p_i \rightarrow \varepsilon)$
  - $d_{\varepsilon j} = c(\varepsilon \rightarrow q_j)$
  - $d_{ij} = c(p_i \rightarrow q_j)$
- That is, $(n \cdot m+1)$ assignment problems have to be solved altogether:
  - First, $n \cdot m$ assignment problems for the edges to compute C.
  - Then, the final assignment problem based on C.
- Typically, the number of nodes is much higher than the maximum edge degree. In this case, the computational bottleneck is the final assignment problem based on C, which is solved in $O((n+m)^3)$.

BP Algorithm

- In summary, the BP approximation is obtained in three steps:

1: **procedure** BP(g_1, g_2)
2: Build cost matrix $\mathbf{C} = (c_{ij})$ according to the input graphs g_1 and g_2
3: Find optimal assignment $\psi = \{u_1 \rightarrow v_{\varphi_1}, u_2 \rightarrow v_{\varphi_2}, \ldots, u_{m+n} \rightarrow v_{\varphi_{m+n}}\}$ on $\mathbf{C}$
4: Complete edit path according to ψ and compute $d_{\langle\psi\rangle}(g_1, g_2)$
5: **return** $d_{\langle\psi\rangle}$

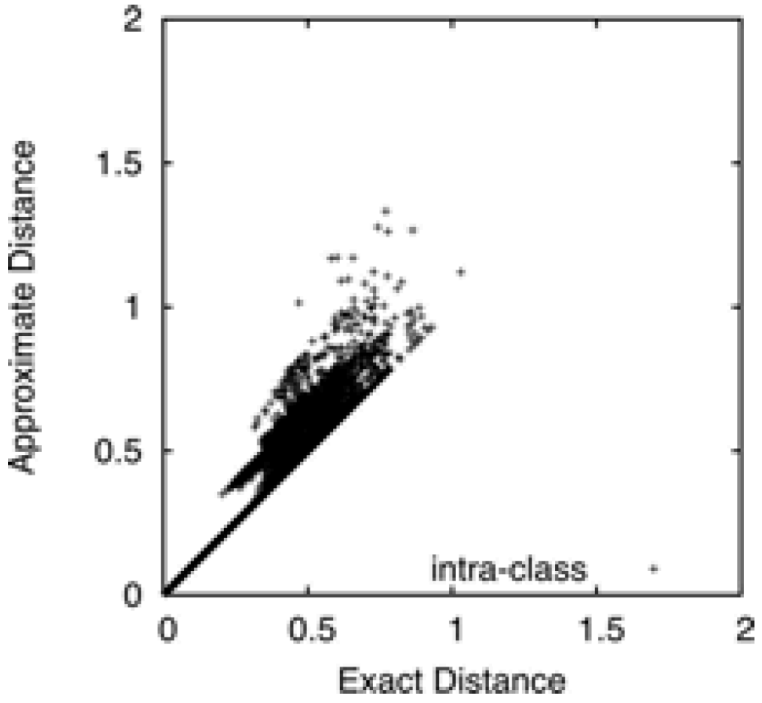
- After solving the assignment problem on C, each node of g_1 is uniquely mapped to a node in g_2 or deleted. *Vice versa*, each node in g_2 is uniquely mapped or inserted. In the third an final step of the BP algorithm (Line 4), the edit cost of this edit path is computed.
- Because the graph structure is taken into account only locally, not globally, the found edit path is not necessarily optimal. Therefore,

$$BP(g_1, g_2) \geq d(g_1, g_2)$$

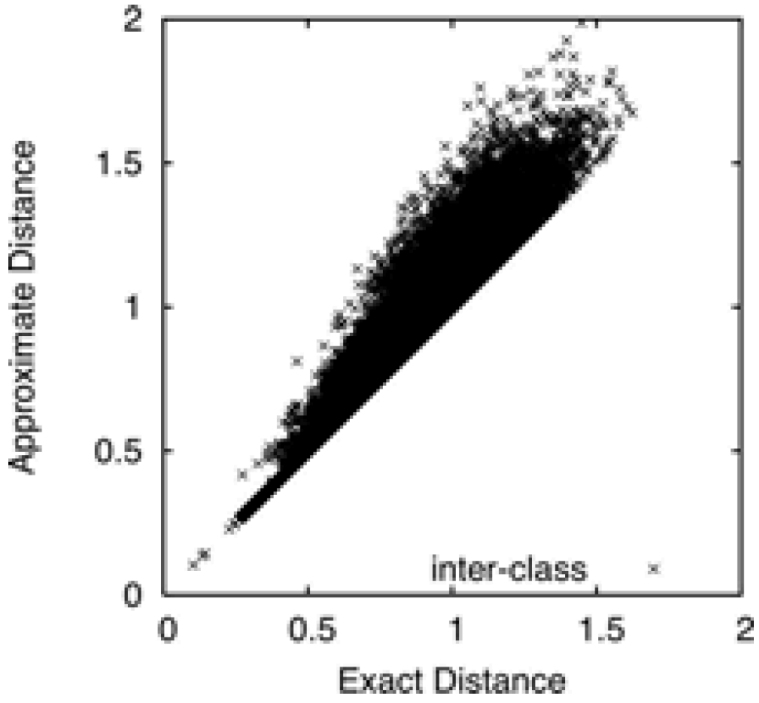
- Because of its cubic time complexity, the BP edit distance algorithm is applicable to large graphs with over hundred nodes.

Example: Approximation

- Example: matching letter graphs (with less than 10 nodes).
- Especially for small edit distances, the BP approximation is very accurate. These small distances are particularly important for classification tasks.



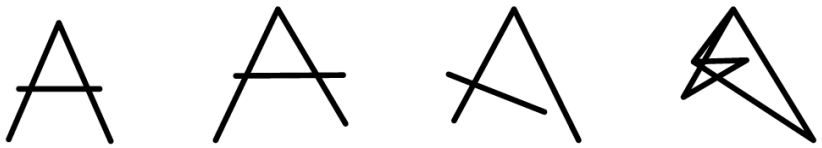
(a) Intra-class distances



(b) Inter-class distances

Example: Classification Accuracy

- Matching letter graphs with three levels of distortion, approximate edit distance (a) vs exact edit distance (e).
- Four meta-parameters have been optimized on the validation set (costs for deletion/insertion of nodes and edges, weight α between node and edge edit operations, and K for KNN).



Data Set	τ_{node}	τ_{edge}	α	k	Accuracy	
Letter low	0.3	0.5	0.75	3	99.7	(e)
	0.3	0.1	0.25	5	99.7	(a)
Letter medium	0.7	0.7	0.5	5	95.3	(e)
	0.7	1.9	0.75	5	95.2	(a)
Letter high	0.9	2.3	0.75	5	90.9	(e)
	0.9	1.7	0.75	5	90.9	(a)