Chapter X: Graph Mining

Information Retrieval & Data Mining
Universität des Saarlandes, Saarbrücken
Winter Semester 2013/14

Chapter X: Graph Mining

1. Introduction to Graph Mining
2. Centrality and Other Graph Properties
3. Frequent Subgraph Mining
    - 3.1. Graphs and Isomorphism
    - 3.2. Canonical Codes
    - 3.3. gSpan
4. Graph Clustering
    - 4.1. Clustering as Graph Cutting
    - 4.2. Spectral Clustering
    - 4.3. Markov Clustering

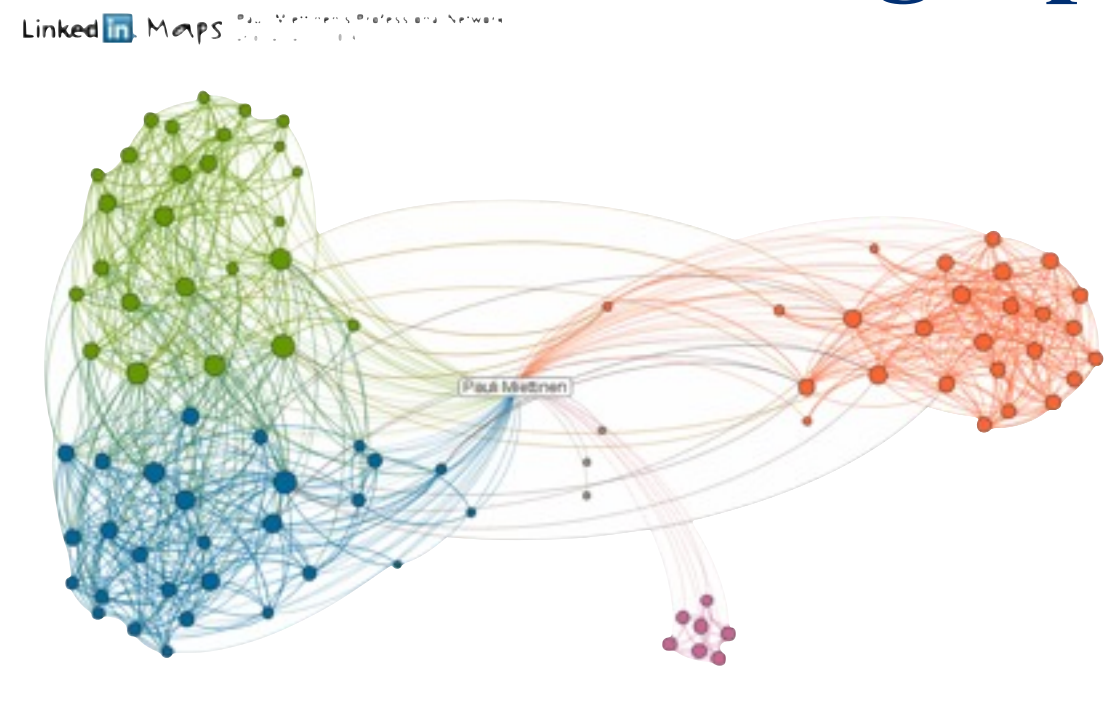
ZM Ch. 4, 11, 16

Chapter X.1: Introduction

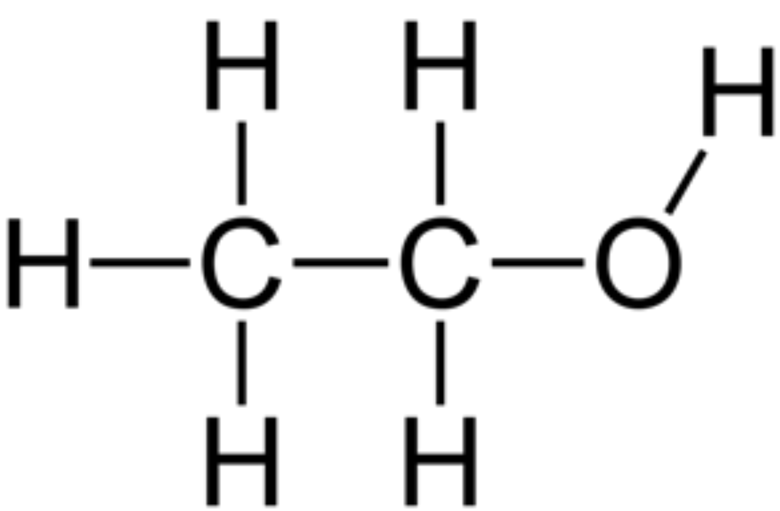
1. Why Graphs?
2. What are Graphs?
3. What to do with Graphs?

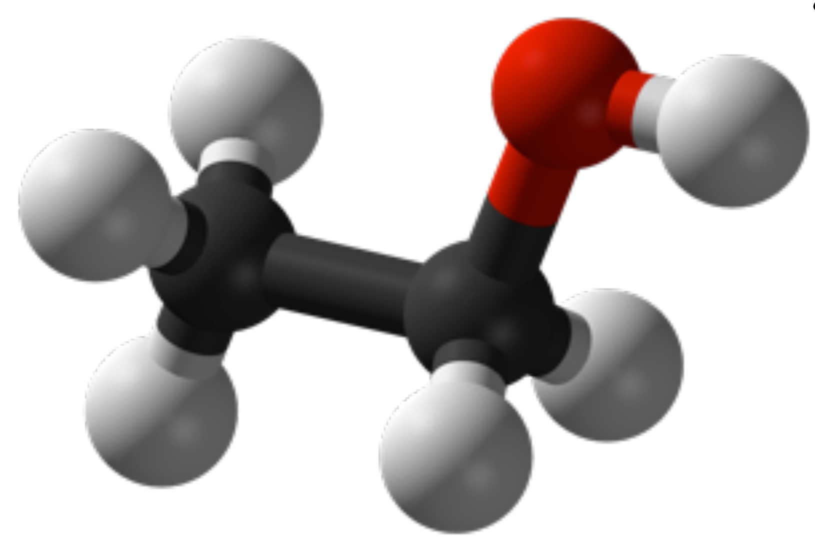
Why Graphs?

- Many real-world data sets are in the forms of **graphs**
  - Social networks
  - Hyperlinks
  - Protein–protein interaction
  - XML parse trees
  - …



- Many of these graphs are enormous
  - Humans cannot understand ⇒ task for data mining!





What are Graphs?

- A **graph** is a pair $(V, E \subseteq V^2)$
  - Elements in V are **vertices** or **nodes** of the graph
  - Pairs (v, u) in E are **edges** or **arcs** of the graph
    - Pairs can be either **ordered** or **unordered** for **directed graphs** or **undirected graphs**, respectively
- The graphs can be **labelled**
  - Vertices can have labeling $L(v)$
  - Edges can have labeling $L(v, u)$
- A **tree** is a **rooted**, **connected**, and **acyclic** graph
- Graphs can be represented using **adjacency matrices**
  - $|V|\times|V|$ matrix $\boldsymbol{A}$ with $(\boldsymbol{A})_{ij} = 1$ if $(v_i, v_j) \in E$

Eccentricity, Radius & Diameter

- The **distance** $d(v_i, v_j)$ between two vertices is the (weighted) length of the shortest path between them
- The **eccentricity** of a vertex v_i, $e(v_i)$, is its maximum distance to any other vertex, $\max_j\{d(v_i, v_j)\}$
- The **radius** of a connected graph, $r(G)$, is the minimum eccentricity of any vertex, $\min_i\{e(v_i)\}$
- The **diameter** of a connected graph, $d(G)$, is the maximum eccentricity of any vertex, $\max_i\{e(v_i)\} = \max_{i,j}\{d(v_i, v_j)\}$
  - The *effective diameter* of a graph is smallest number that is larger than the eccentricity of a large fraction of the vertices in the graph

Clustering Coefficient

- The **clustering coefficient** of vertex v_i, $C(v_i)$, tells how clique-like the neighbourhood of v_i is
  - Let n_i be the number of neighbours of v_i and m_i the number of edges *between* the neighbours of v_i (v_i excluded)

$$C(v_i) = m_i \Big/ \binom{n_i}{2} = \frac{2m_i}{n_i(n_i - 1)}$$

  - Well-defined only for v_i with at least two neighbours
    - For others, let $C(v_i) = 0$
- The clustering coefficient of the graph is the average clustering coefficient of the vertices: $C(G) = n^{-1}\Sigma_i C(v_i)$

What to do with Graphs?

- There are many interesting data one can mine from graphs and sets of graphs
  - Cliques of friends from social networks
  - Hubs and authorities from link graphs
  - Who is the centre of the Hollywood
  - Subgraphs that appear frequently in a set of graphs
  - Areas with higher inter-connectivity than intra-connectivity
  - …

This week

- Graph mining is perhaps the most popular topic in contemporary data mining research
  - Though not necessary called as such…

Chapter X.2: Centrality and Other Graph Properties

1. Centrality
2. Graph Properties

ZM Ch. 4

Centrality

- Six degrees of Kevin Bacon
  - "Every actor is related to Kevin Bacon by no more than 6 hops"
  - Kevin Bacon has acted with many, that have acted with many others, that have acted with many others…
- That makes Kevin Bacon a *centre* of the co-acting graph
  - Although he's not the centre: the average distance to him is 2.998 but to Harvey Keitel it is only 2.848

http://oracleofbacon.org

Degree and Eccentricity Centrality

- **Centrality** is a function $c: V \to \mathbb{R}$ that induces a total order in V
  - The higher the centrality of a vertex, the more important it is
- In **degree centrality** $c(v_i) = d(v_i)$, the degree of the vertex
- In **eccentricity centrality** the least eccentric vertex is the most central one, $c(v_i) = 1/e(v_i)$
  - The lest eccentric vertex is *central*
  - The most eccentric vertex is *peripheral*

Closeness Centrality

- In **closeness centrality** the vertex with least distance to *all other* vertices is the centre

$$c(v_i) = \left(\sum_j d(v_i, v_j) \right)^{-1}$$

- In eccentricity centrality we aim to minimize the maximum distance
- In closeness centrality we aim to minimize the average distance
  - This is the distance used to measure the centre of Hollywood

Betweenness Centrality

- The **betweenness centrality** measures the number of shortest paths that travel through v_i
  - Measures the "monitoring" role of the vertex
  - "All roads lead to Rome"
- Let η_{jk} be the number of shortest paths between v_j and v_k and let $\eta_{jk}(v_i)$ be the number of those that include v_i
  - Let $\gamma_{jk}(v_i) = \eta_{jk}(v_i)/\eta_{jk}$
  - Betweenness centrality is defined as

$$c(v_i) = \sum_{j \neq i} \sum_{\substack{k \neq i \\ k > j}} \gamma_{jk}$$

Prestige

- In **prestige**, the vertex is more central if it has many incoming edges from other vertices of high prestige
  - $\boldsymbol{A}$ is the adjacency matrix of the directed graph G
  - $\boldsymbol{p}$ is n-dimensional vector giving the prestige of the vertices
  - $\boldsymbol{p} = \boldsymbol{A}^T\boldsymbol{p}$
  - Starting from an initial prestige vector $\boldsymbol{p}_0$, we get
    $\boldsymbol{p}_k = \boldsymbol{A}^T\boldsymbol{p}_{k-1} = \boldsymbol{A}^T(\boldsymbol{A}^T\boldsymbol{p}_{k-2}) = (\boldsymbol{A}^T)^2\boldsymbol{p}_{k-2} = (\boldsymbol{A}^T)^3\boldsymbol{p}_{k-3} = \ldots$
    $= (\boldsymbol{A}^T)^k\boldsymbol{p}_0$
- Vector $\boldsymbol{p}$ converges to the dominant eigenvector of $\boldsymbol{A}^T$
  - Under some assumptions
- N.B. PageRank is based on (normalized) prestige

Graph Properties

- Several real-world graphs exhibit certain characteristics
  - Studying what these are and explaining why they appear is an important area of network research
- As data miners, we need to understand the consequences of these characteristics
  - Finding a result that can be explained merely by one of these characteristics is not interesting
- We also want to *model* graphs with these characteristics

Small-World Property

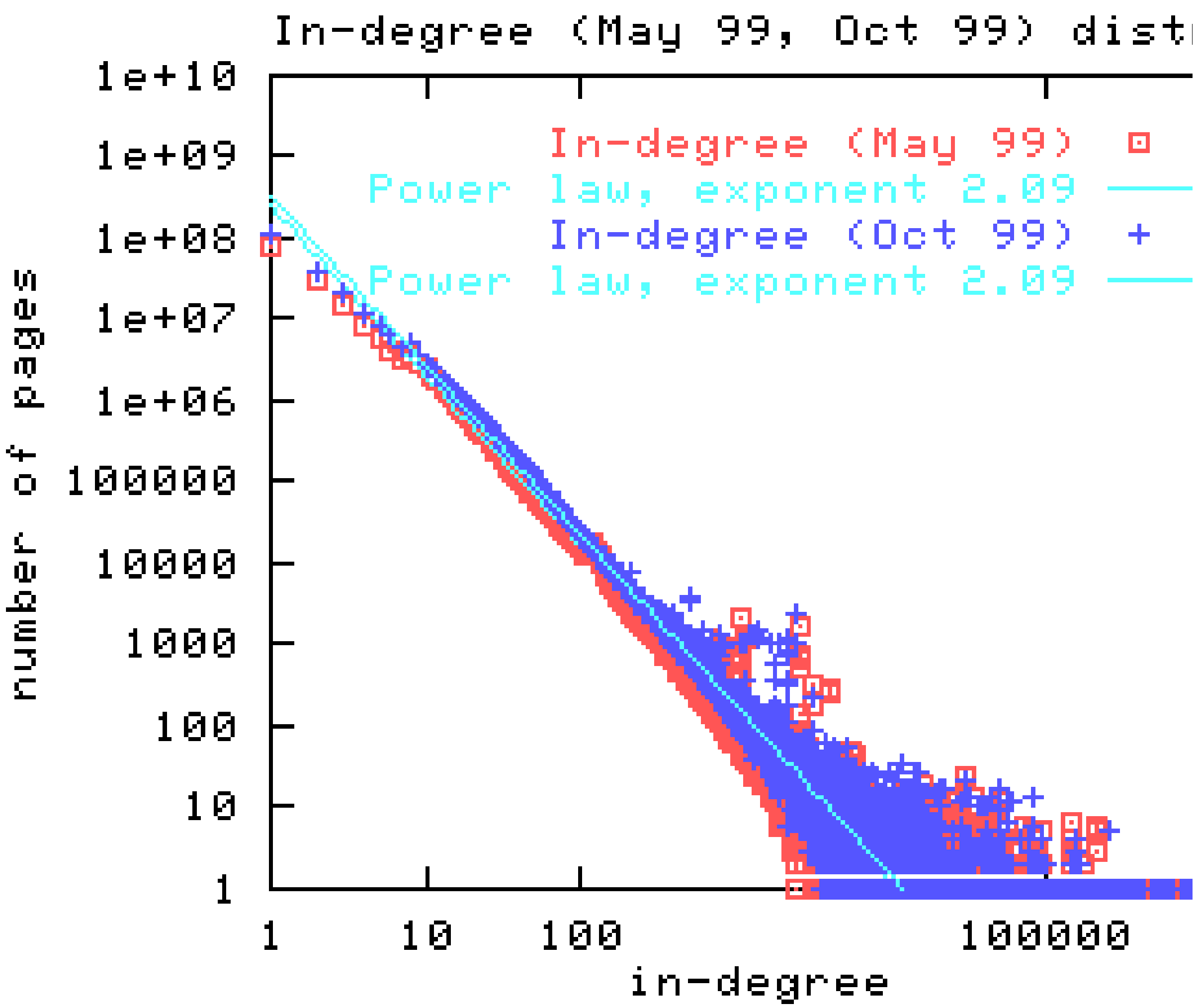
- A graph G is said to exhibit a **small-world property** if its average path length scales logarithmically, $\mu_L \propto \log n$
  - The six degrees of Kevin Bacon is based on this property
  - Also the Erdős number
    - How far a mathematician is from Hungarian combinatorist Paul Erdős
    - A radius of a large, connected mathematical co-authorship network (268K authors) is 12 and diameter 23

Scale-Free Property

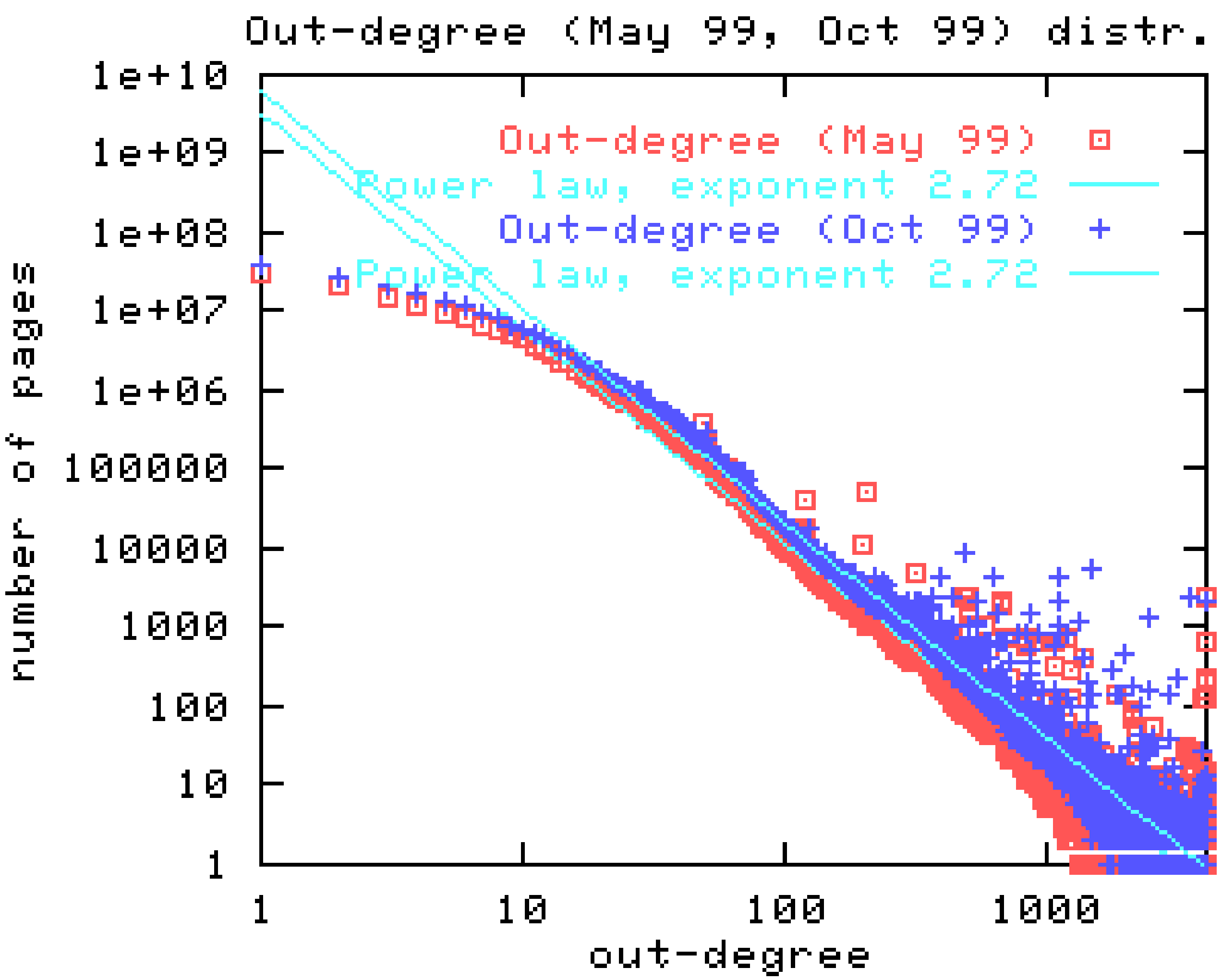
- The **degree distribution** of a graph is the distribution of its vertex degrees
  - How many vertices with degree 1, how many with degree 2, etc.
  - $f(k)$ is the number of edges with degree k
- A graph is said to exhibit **scale-free property** if $f(k) \propto k^{-\gamma}$
  - So-called power-law distribution
  - Majority of vertices have small degrees, few have very high degrees
  - Scale-free: $f(ck) = \alpha(ck)^{-\gamma} = (\alpha c^{-\gamma})k^{-\gamma} \propto k^{-\gamma}$

Example: WWW Links

In-degree



Out-degree



Broder et al. *Graph structure in the web*. WWW'00

$s = 2.09$ $s = 2.72$

Clustering Effect

- A graph exhibits **clustering effect** if the distribution of average clustering coefficient (per degree) follow the power law
  - If $C(k)$ is the average clustering coefficient of all vertices of degree k, then $C(k) \propto k^{-\gamma}$
- The vertices with small degrees are part of highly clustered areas (high clustering coefficient) while "hub vertices" have smaller clustering coefficients

Chapter X.3: Frequent Subgraph Mining

1. Graphs and Isomorphism
    1.1. Definitions
    1.2. Support of a subgraph
2. Canonical Codes
3. gSPAN Algorithm
4. Easier Problems

ZM Ch. 11

Graphs and Isomorphism

- Graph (V', E') is the **subgraph** of graph (V, E) if
  - $V' \subseteq V$
  - $E' \subseteq E$
- Note that subgraphs don't have to be connected
  - Today we consider only **connected subgraphs**
- To check whether a graph is a subgraph of other is trivial
  - But in most real-world applications there are no direct subgraphs
  - Two graphs might be similar even if their vertex sets are disjoint

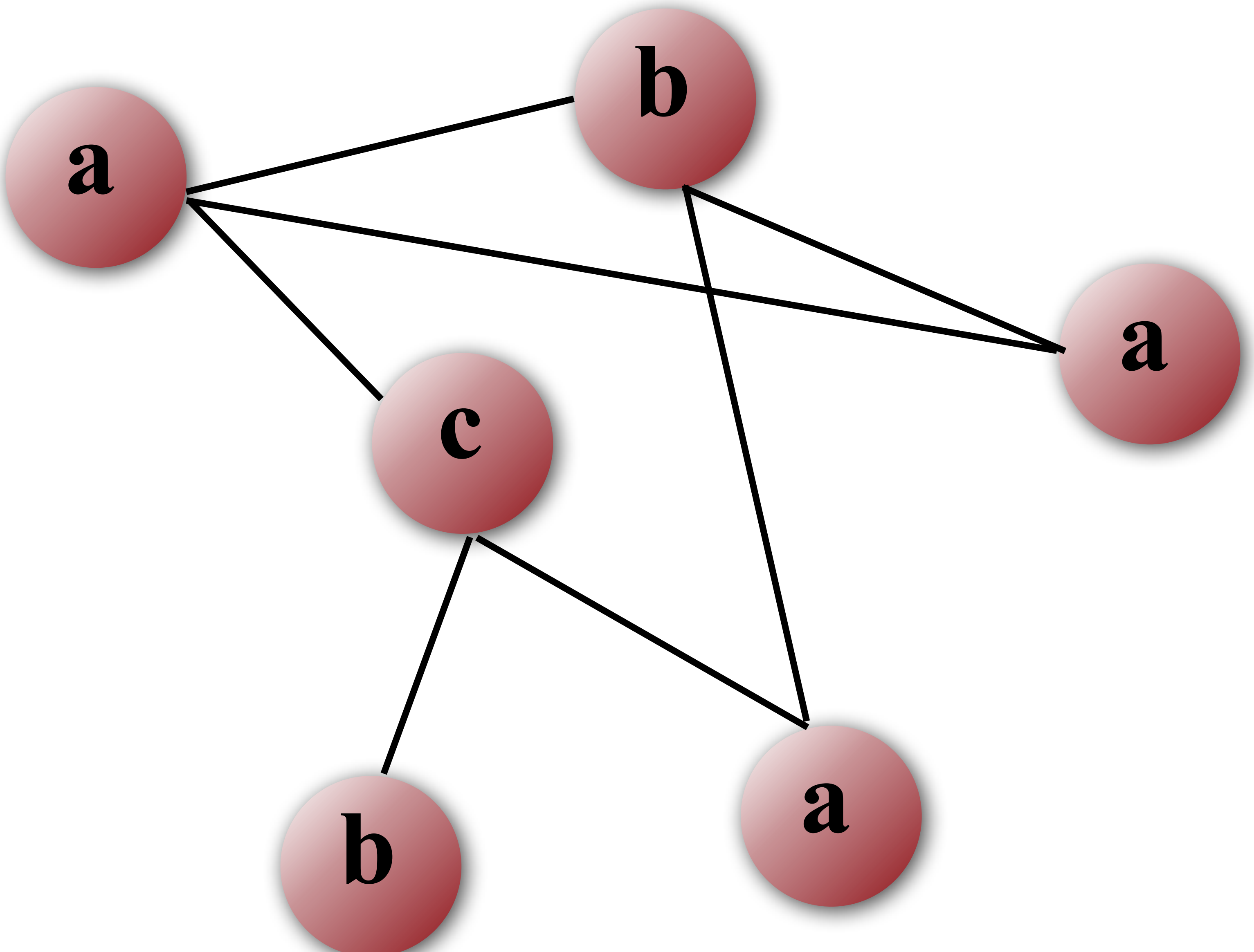
Graph Isomorphism

- Graphs $G = (V, E)$ and $G' = (V', E')$ are **isomorphic** if there exists a bijective function $\varphi: V \rightarrow V'$ such that
  - $(u, v) \in E$ if and only if $(\varphi(u), \varphi(v)) \in E'$
  - $L(v) = L(\varphi(v))$ for all $v \in V$
  - $L(u, v) = L(\varphi(u), \varphi(v))$ for all $(u, v) \in E$
- Graph G' is **subgraph isomorphic** to G if there exists a subgraph of G which is isomorphic to G'
- No polynomial-time algorithm is known for determining if G and G' are isomorphic
- Determining if G' is subgraph isomorphic to G is NP-hard

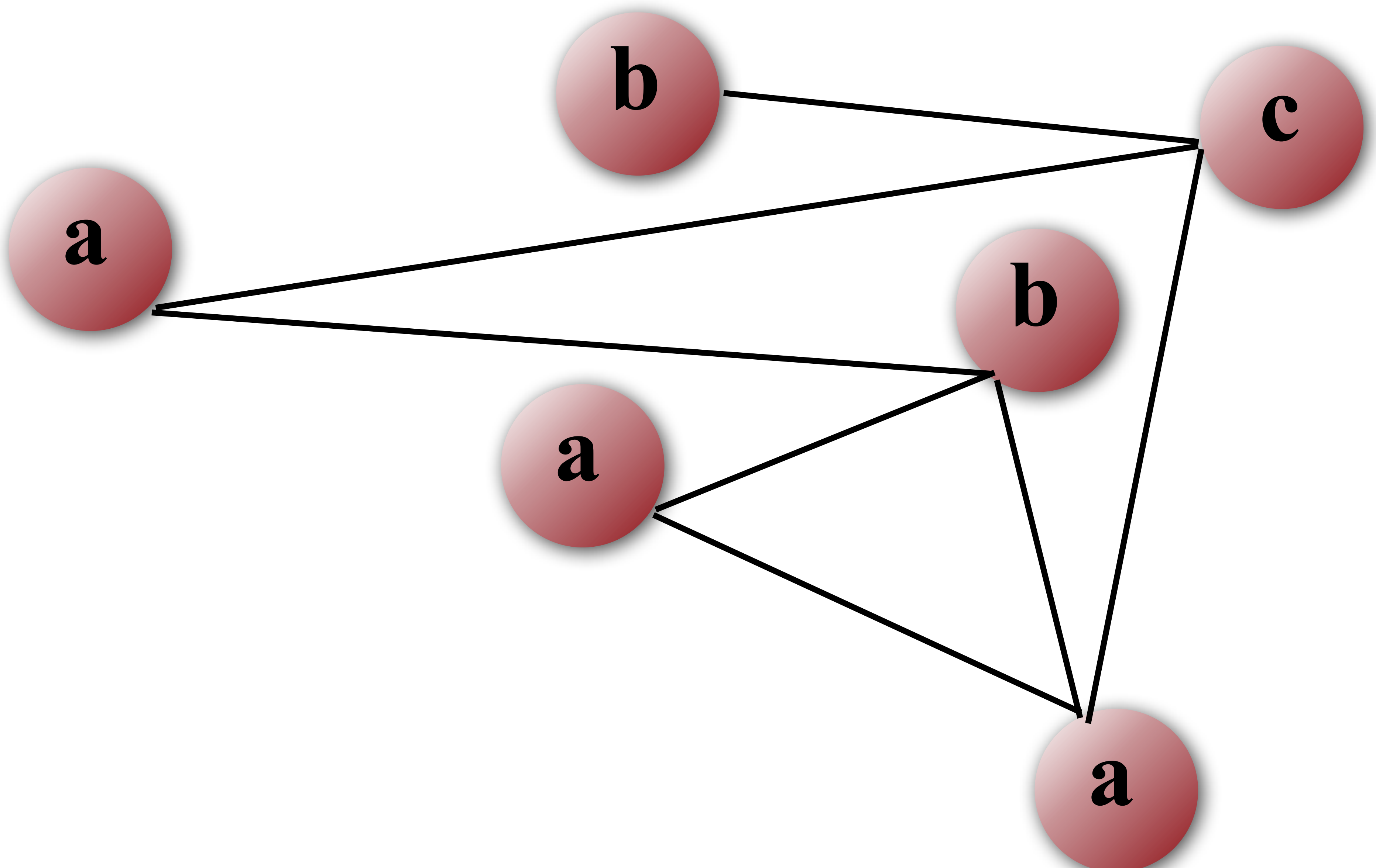
Equivalence and Canonical Graphs

- Isomorphism defines an equivalence class
  - id: $V \to V$, id(v) = v shows G is isomorphic to itself
  - If G is isomorphic to G' via φ, then G' is isomorphic to G via φ^{-1}
  - If G is isomorphic to H via φ and H to I via χ, then G is isomorphic to I via $\varphi \circ \chi$
- A **canonization** of a graph G, $canon(G)$ produces another graph C such that if H is a graph that is isomorphic to G, $canon(G) = canon(H)$
  - Two graphs are isomorphic if and only if their canonical versions are the same

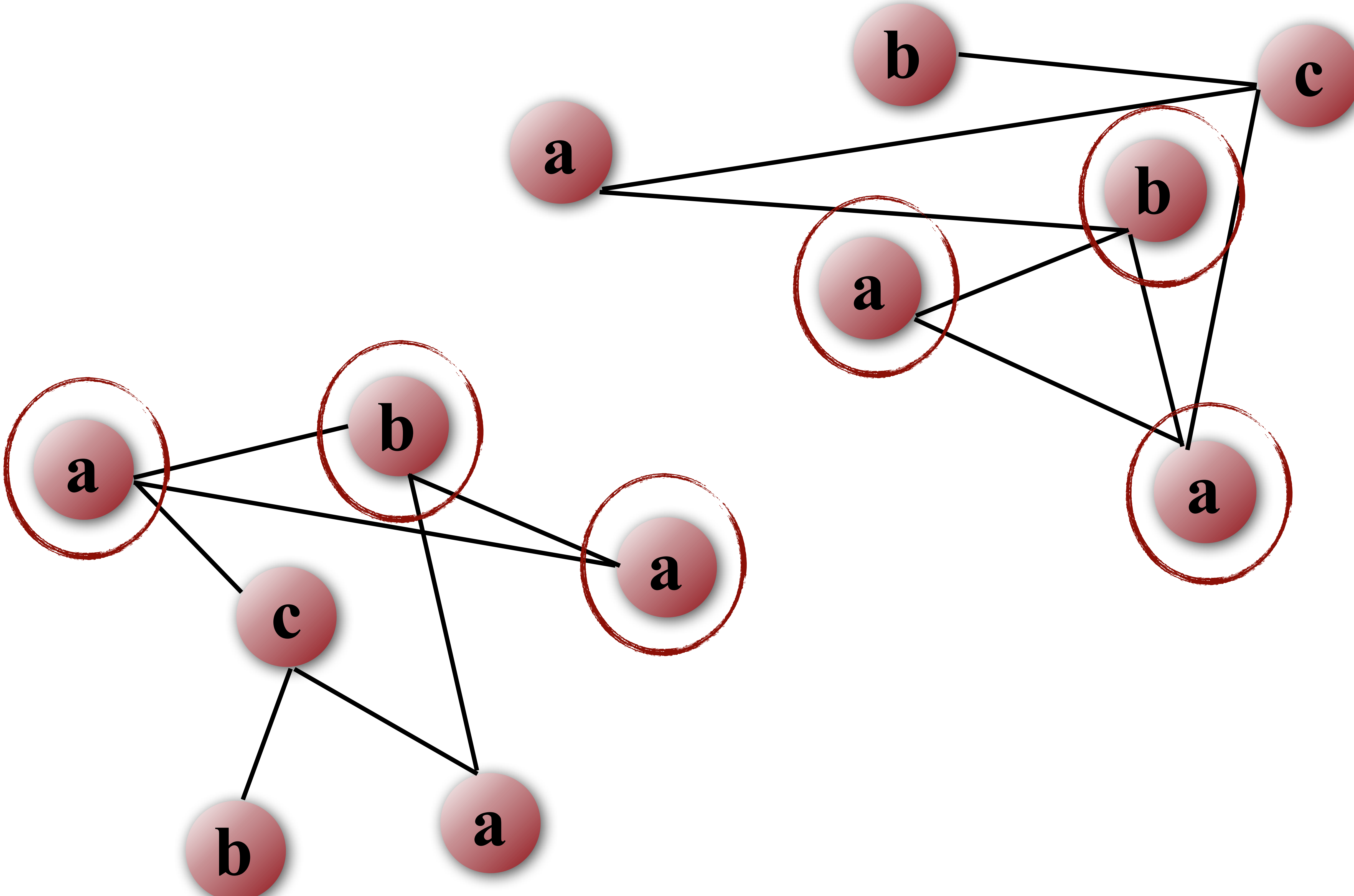
An Example of Isomorphic Graphs



An Example of Isomorphic Graphs



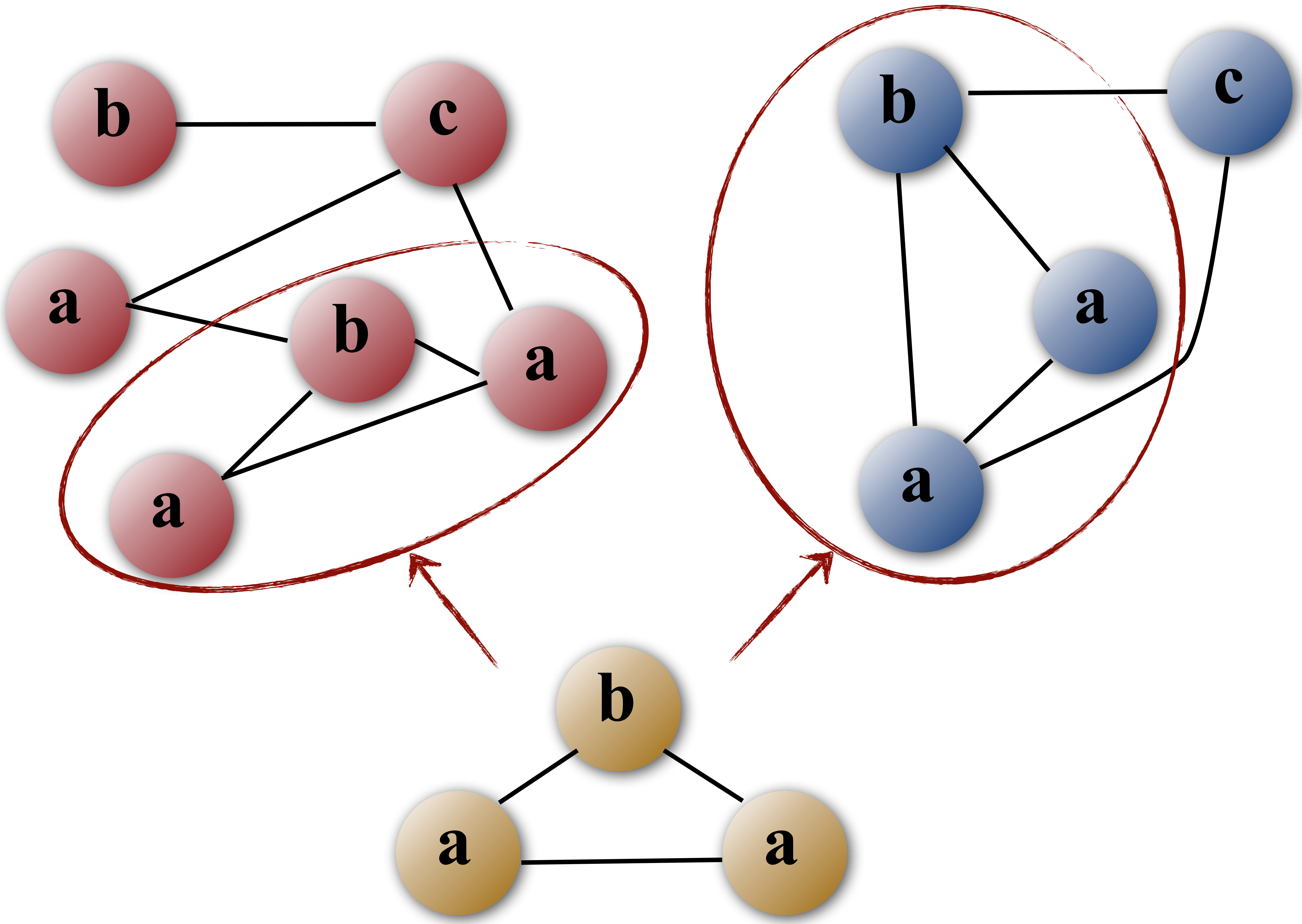
An Example of Isomorphic Graphs



Frequent Subgraph Mining

- Given a set D of n graphs and a minimum support parameter *minsup*, find all connected graphs that are subgraph isomorphic to at least *minsup* graphs in D
  - Enormously complex problem
  - For graphs that have m vertices there are
    - $2^{O(m^2)}$ subgraphs (not all are connected)
  - If we have s labels for vertices and edges we have
    - $O\left((2s)^{O(m^2)}\right)$ labelings of the different graphs
  - Counting the support means solving multiple NP-hard problems

An Example



Canonical Codes

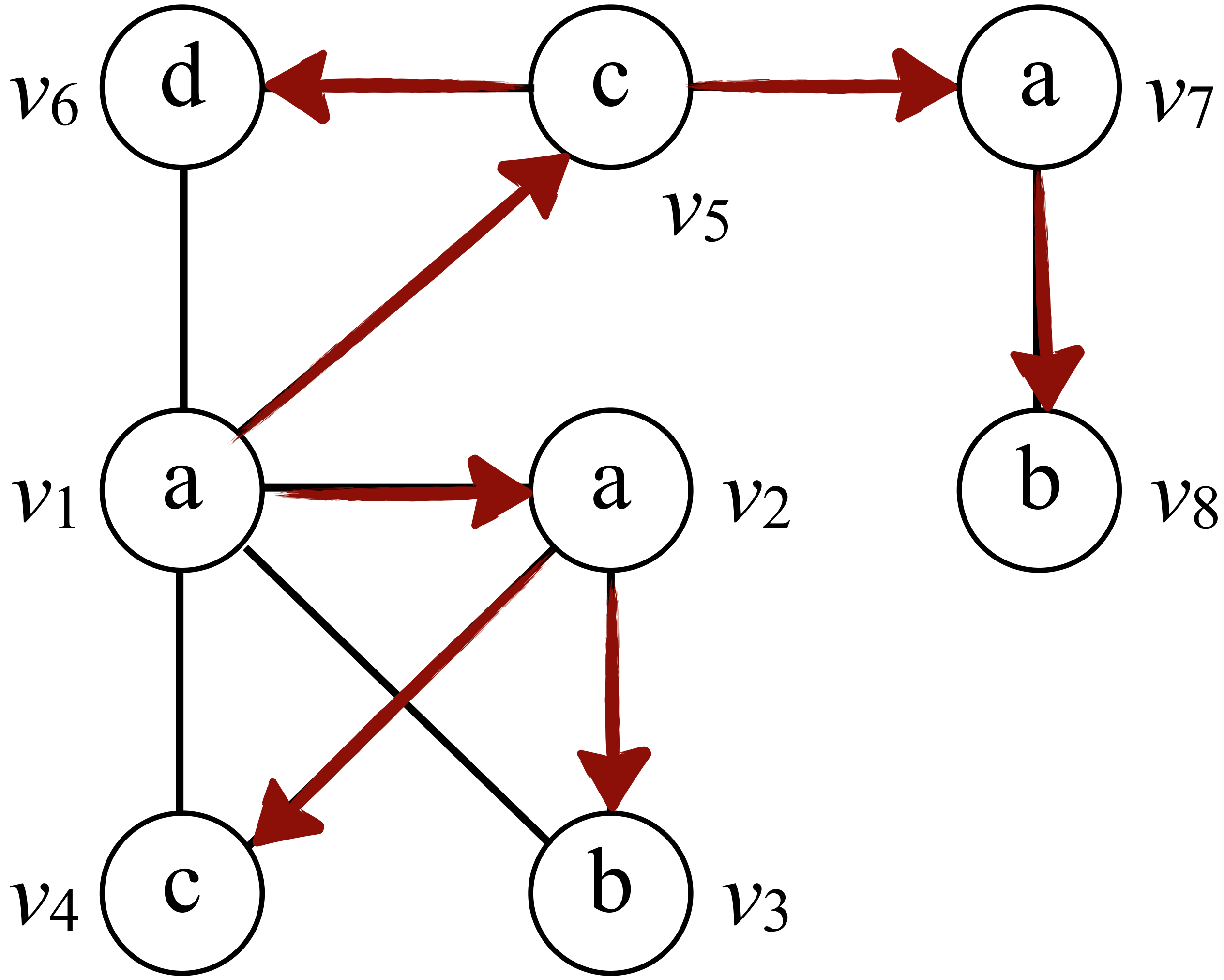
- We can improve the running time of frequent subgraph mining by either
  - Making the frequency check faster
    - Lots of efforts in faster isomorphism checking but only little progress
  - Creating less candidates that need to be checked
    - Level-wise algorithms (like AGM) generate huge numbers of candidates
    - Each must be checked with for isomorphism with others
- The gSpan (graph-based Substructure pattern mining) algorithm replaces the level-wise approach with a depth-first approach

Yan & Han 2002; Z&M Ch. 11

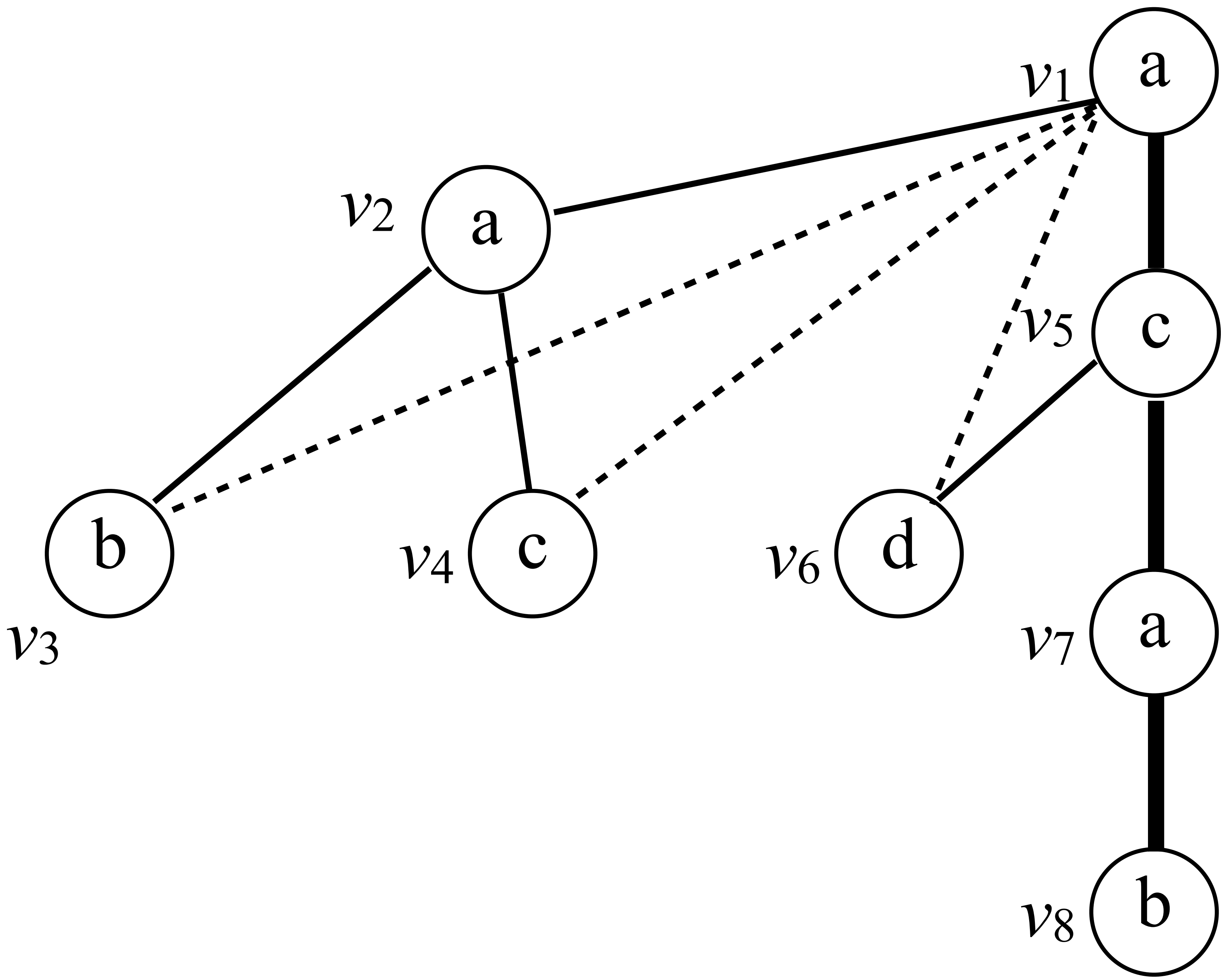
Depth-First Spanning Tree

- A depth-first spanning (DFS) tree of a graph *G*
  - Is a connected tree
  - Contains all the vertices of *G*
  - Is build in depth-first order
    - Selection between the siblings is e.g. based on the vertex index
- Edges of the DFS tree are *forward edges*
- Edges not in the DFS tree are *backward edges*
- A *rightmost path* in the DFS tree is the path travels from the root to the *rightmost vertex* by always taking the rightmost child (last-added)

An Example



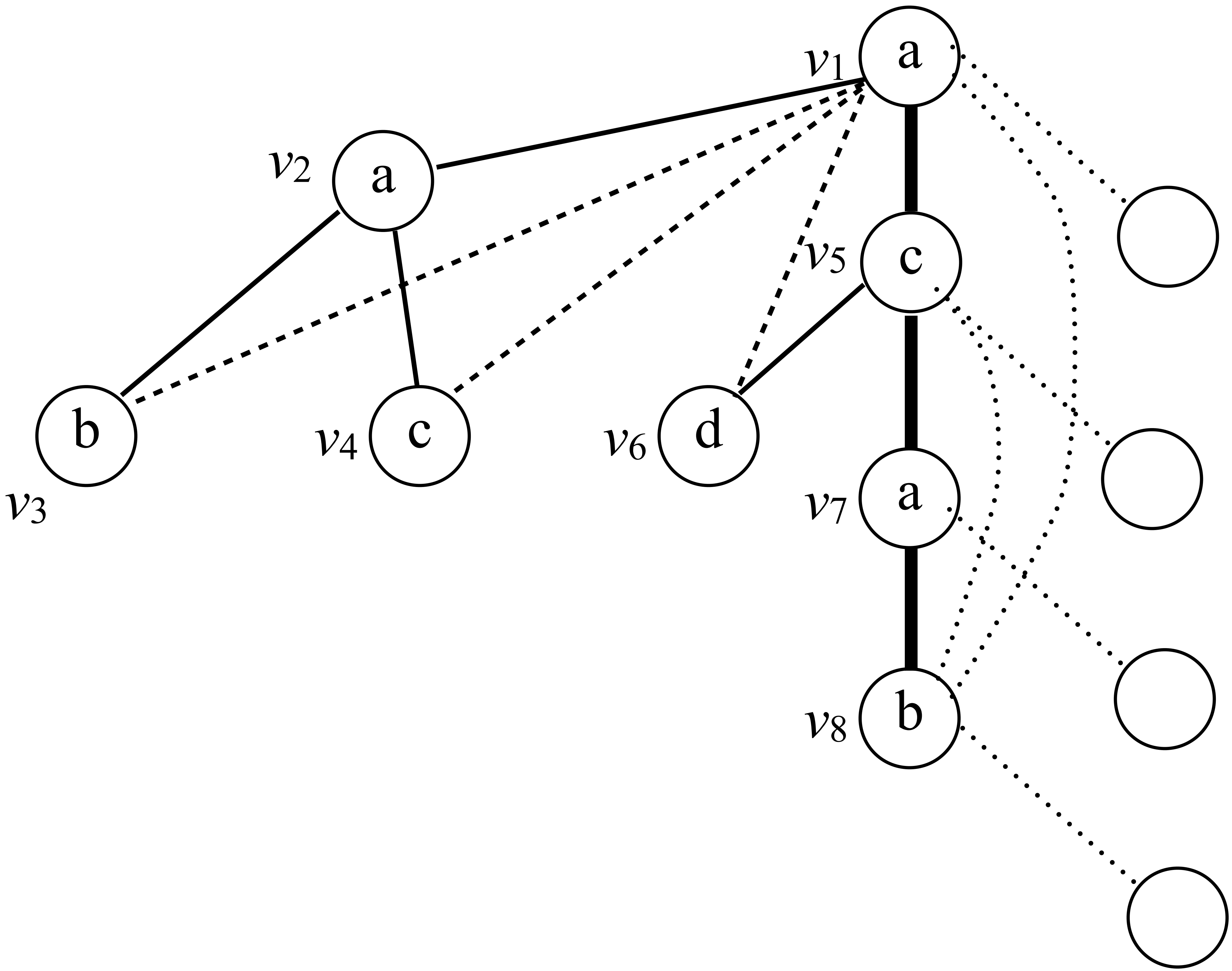
The DFS Tree



Generating Candidates from DFS Tree

- Given graph G, we extend it *only* from the vertices in the rightmost path
  - We can add backwards edges from the rightmost vertex to some other vertex in the rightmost path
  - We can add a forward edge from any vertex in the rightmost path
    - This increases the number of vertices by 1
- The order of generating the candidates is
  - First backward extensions
    - First to root, then to root's child, …
  - Then forward extensions
    - First from the leaf, then from leaf's father, …

An Example



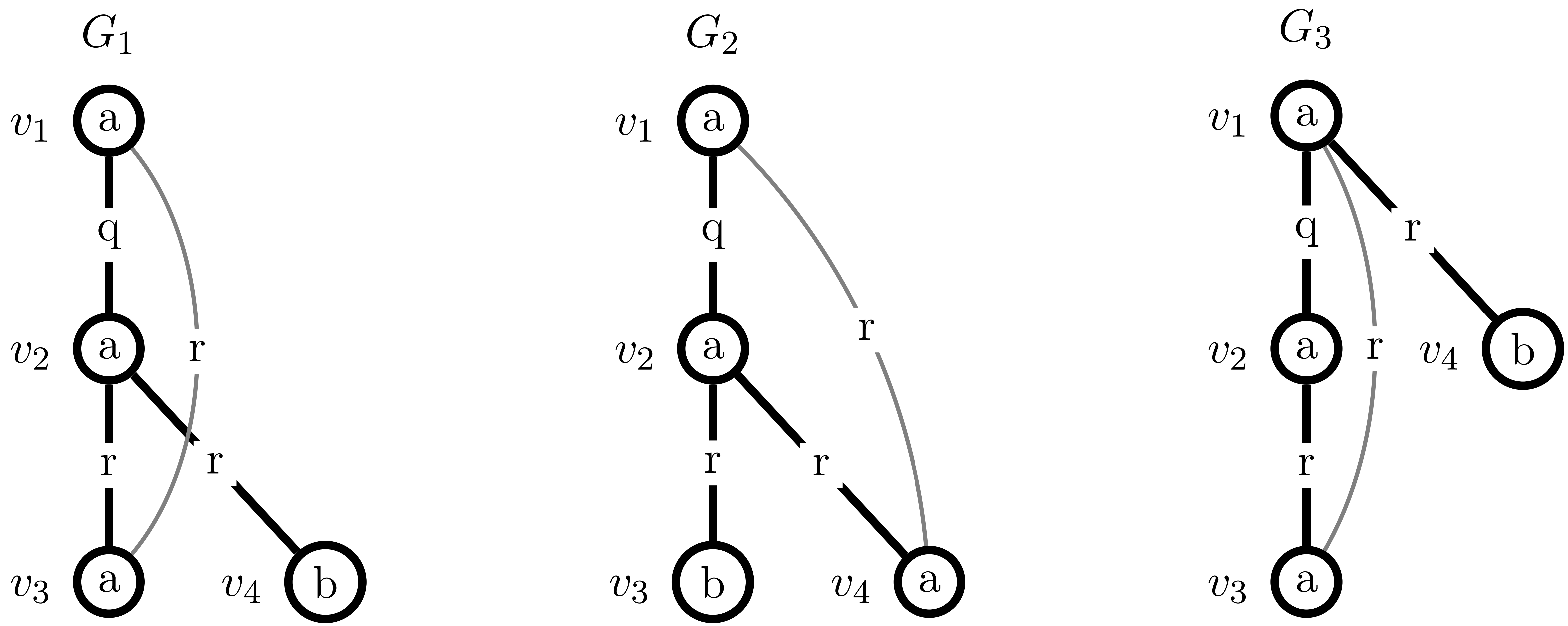
DFS Codes and their Orders

- A **DFS code** is a sequence of tuples of type $\langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j)\rangle$
  - Tuples are given in DFS order
    - Backwards edges are listed before forward edges
    - Vertices are numbered in DFS order
- A DFS code is **canonical** if it is the smallest of the codes in the ordering
  - $\langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j)\rangle < \langle v_x, v_y, L(v_x), L(v_y), L(v_x, v_y)\rangle$ if
    - $\langle v_i, v_j\rangle <_e \langle v_x, v_y\rangle$; or
    - $\langle v_i, v_j\rangle = \langle v_x, v_y\rangle$ and $\langle L(v_i), L(v_j), L(v_i, v_j)\rangle <_l \langle L(v_x), L(v_y), L(v_x, v_y)\rangle$
  - The ordering of the label tuples is the lexicographical ordering

Ordering the Edges

- Let $e_{ij} = \langle v_i, v_j \rangle$ and $e_{xy} = \langle v_x, v_y \rangle$
- $e_{ij} <_e e_{xy}$ if
  - If e_{ij} and e_{xy} are forward edges, then
    - $j < y$; or
    - $j = y$ and $i > x$
  - If e_{ij} and e_{xy} are backward edges, then
    - $i < x$; or
    - $i = x$ and $j < y$
  - If e_{ij} is forward and e_{xy} is backward, then $i < y$
  - If e_{ij} is backward and e_{xy} is forward, then $j \leq x$

Example



$t_{11} = \langle v_1, v_2, a, a, q\rangle$
$t_{12} = \langle v_2, v_3, a, a, r\rangle$
$t_{13} = \langle v_3, v_1, a, a, r\rangle$
$t_{14} = \langle v_2, v_4, a, b, r\rangle$

$t_{21} = \langle v_1, v_2, a, a, q\rangle$
$t_{22} = \langle v_2, v_3, a, b, r\rangle$
$t_{23} = \langle v_2, v_4, a, a, r\rangle$
$t_{24} = \langle v_4, v_1, a, a, r\rangle$

$t_{31} = \langle v_1, v_2, a, a, q\rangle$
$t_{32} = \langle v_2, v_3, a, a, r\rangle$
$t_{33} = \langle v_3, v_1, a, a, r\rangle$
$t_{34} = \langle v_1, v_4, a, b, r\rangle$

Last rows are … G_1 is smallest

The gSPAN Algorithm

- The general idea:
  - Use the DFS codes to create candidates
    - Extend only canonical and frequent candidates
- There can be very, very many extensions
  - And we need to see them all, and all of their isomorphisms, to count the support

Building the Candidates

- The candidates are build in a **DFS code tree**
  - A DFS code **a** is an **ancestor** of DFS code **b** if **a** is a proper prefix of **b**
  - The siblings in the tree follow the DFS code order
- A graph can be frequent only if all of the graphs representing its ancestors in the DFS tree are frequent
- The DFS tree contains all the canonical codes for all the subgraphs of the graphs in the data
  - But not all of the vertices in the code tree correspond to canonical codes
- We will (implicitly) traverse this tree

The Algorithm

- gSpan:
  - **for each** frequent 1-edge graphs
    - **call** subgrm to grow all nodes in the code tree rooted in this 1-edge graph
    - remove this edge from the graph
- subgrm
  - **if** the code is not canonical, return
  - Add this graph to the set of frequent graphs
  - Create each super-graph with one more edge and compute its frequency
  - **call** subgrm with each frequent super-graph's canonical representation

How to compute the frequency?

- gSPAN merges extension generation and support computation
- For each graph in the data base
  - gSPAN computes all the isomorphisms of the current candidate
    - Can mean solving NP-complete problems…
  - For all isomorphisms, gSPAN computes all backward and forward extensions
    - These extensions are stored together with the graph they appear in
- The support of each extension is the number of times we've stored it

How to check the canonicity?

- Given a DFS code of an extension, we need to check if the code is canonical
- This can be done by re-creating the code
  - At every step, choose the smallest of the right-most path extensions of the current code *in the graph corresponding to the extension*
- If at any step we get a code that is smaller than the suffix of the extension's code, we can't have a canonical code
  - If after *k* steps we arrive to the extensions code, the code was canonical

Easier Problems

- Much of the complexity of subgraph mining lies in the isomorphism
- But for some types of graphs isomorphism is easy
  - Different types of trees
    - Ordered and unordered
    - Rooted and unrooted
  - Graphs where every node has a distinct label