Chapter 8:

Graph Mining

Jilles Vreeken

Revision 1, December 4th
typo's fixed: edge order

IRDM '15/16

1 Dec 2015

UNIVERSITÄT DES SAARLANDES

M2CI CLUSTER OF EXCELLENCE

mpi max planck institut informatik

IRDM Chapter 8, overview

1. The basics
2. Properties of Graphs
3. Frequent Subgraphs
4. Community Detection
5. Graph Clustering

You'll find this covered in:
Aggarwal, Ch. 17, 19
Zaki & Meira, Ch. 4, 11, 16

IRDM Chapter 8, today

1. The basics
2. Properties of Graphs
3. Frequent Subgraphs
4. Community Detection
5. Graph Clustering

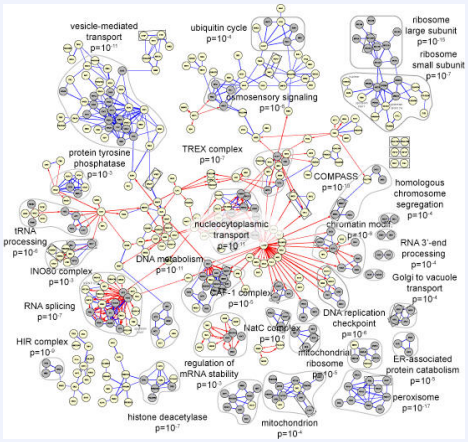
You'll find this covered in:
Aggarwal, Ch. 17, 19
Zaki & Meira, Ch. 4, 11, 16

Chapter 7.1:
The Basics

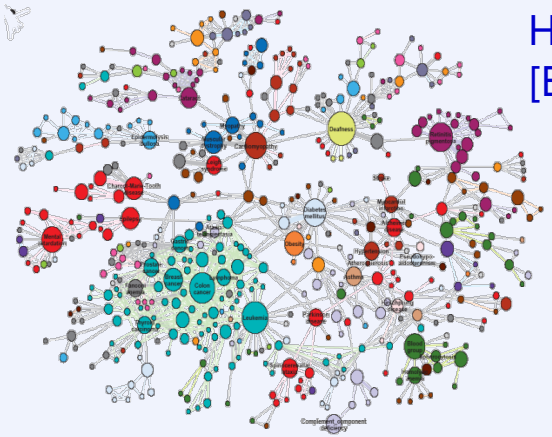
Aggarwal Ch. 17.1

Networks are everywhere!

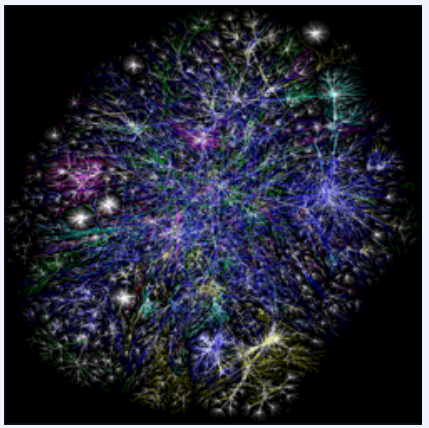
Facebook Network [2010]



Gene Regulatory Network [Decourty 2008]



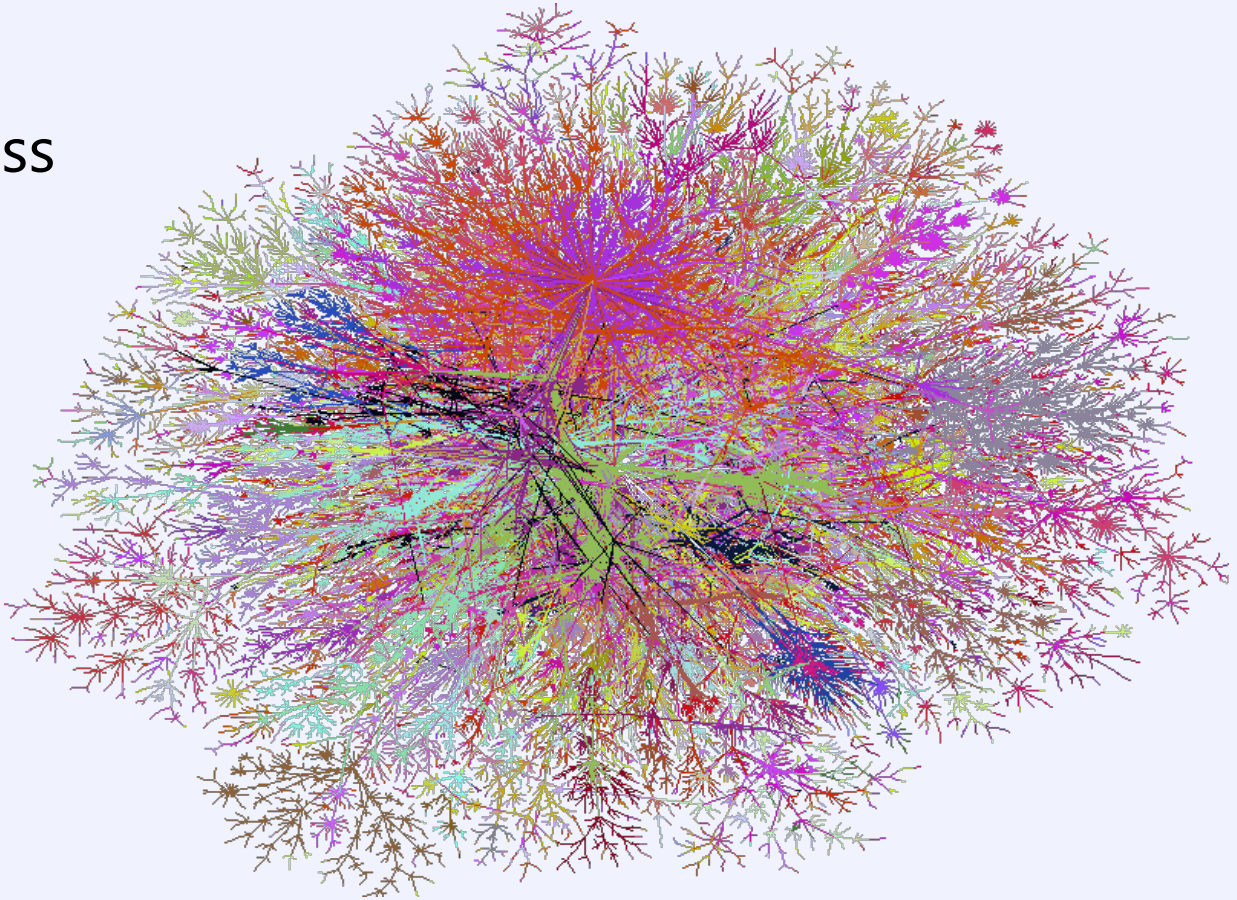
Human Disease Network [Barabasi 2007]



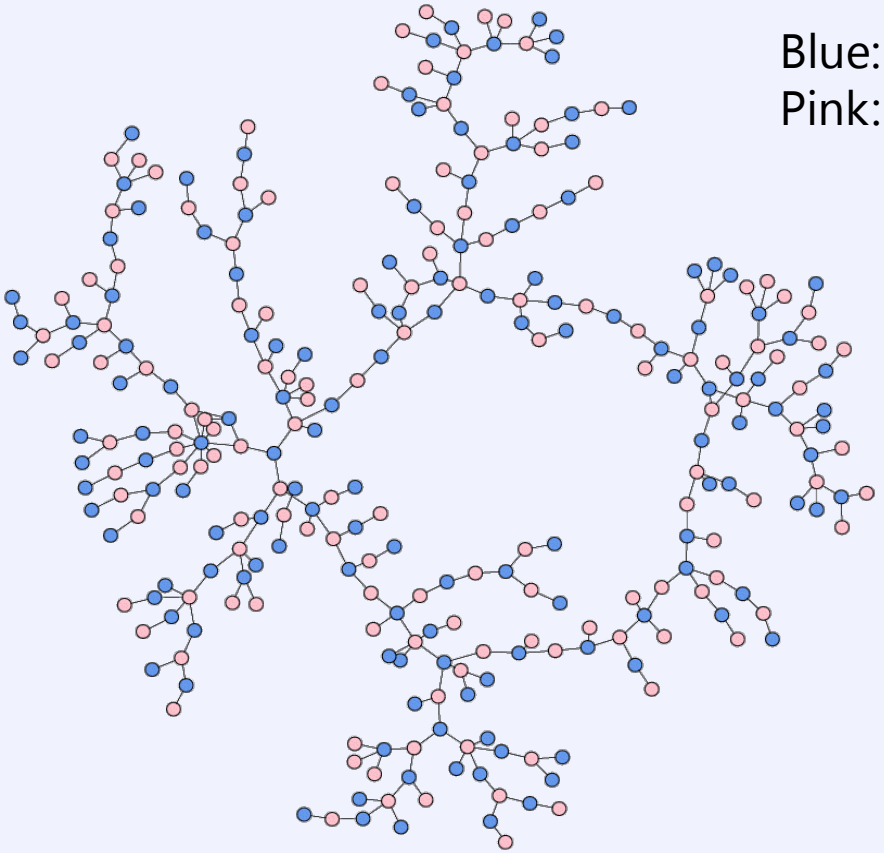
The Internet [2005]

The Internet

Skewed
Degrees
Robustness



High school dating network

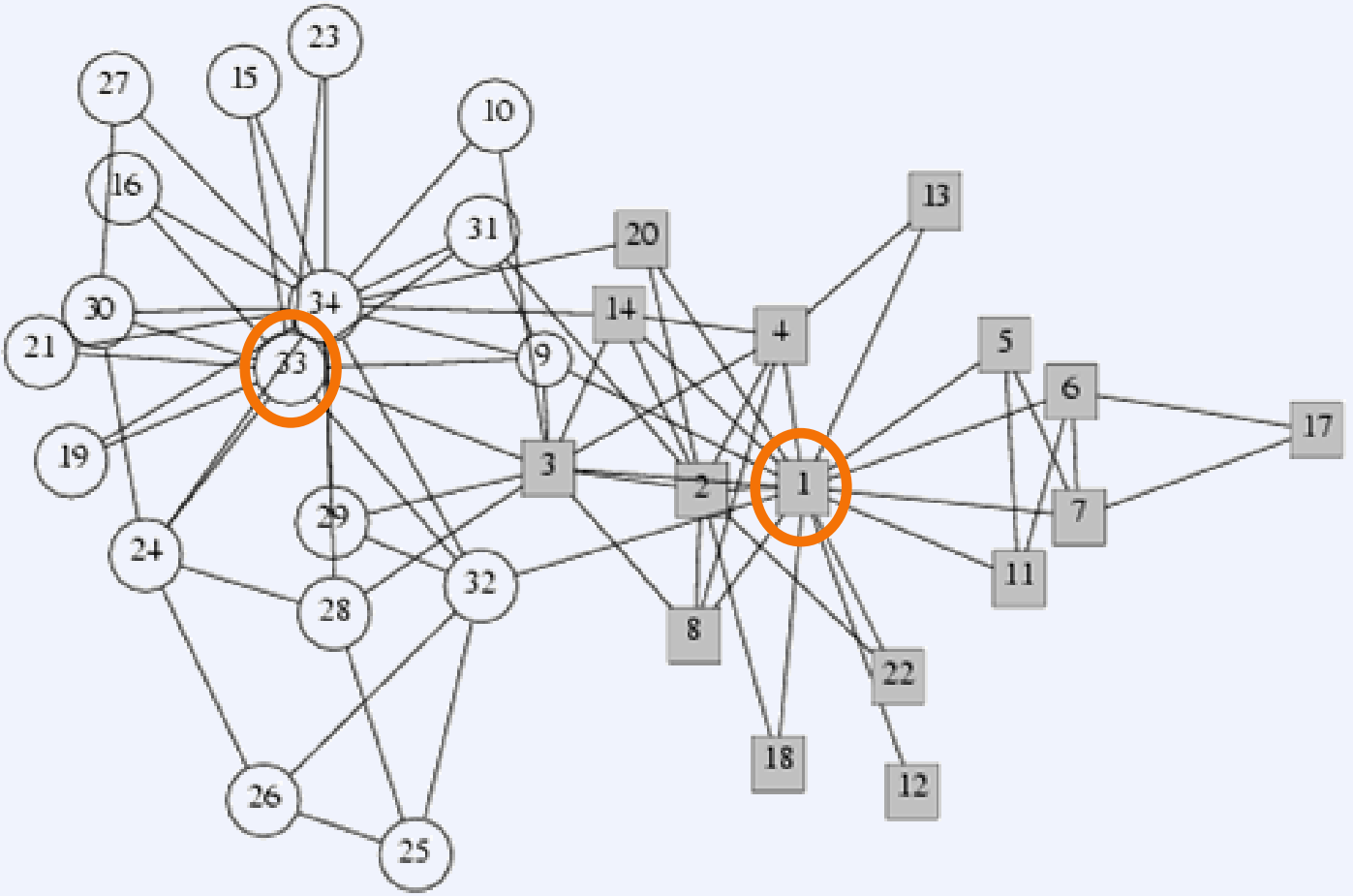


Blue: Male
Pink: Female

Interesting observations?

(Bearman et. al. Am. Jnl. of Sociology, 2004. Image: Mark Newman)

Karate club network



Friends

How many of you think that
your friends have **more** friends than you?

A recent Facebook study

- examined all of FB's users: 721 million people with 69 billion friendships
  - about 10% of the world's population
- found that 93 percent of the time a user's friend count was **less than the average friend count** of his or her friends,
- users had an average of 190 friends,
  while their friends averaged 635 friends of their own

Reasons?

You are a loner?

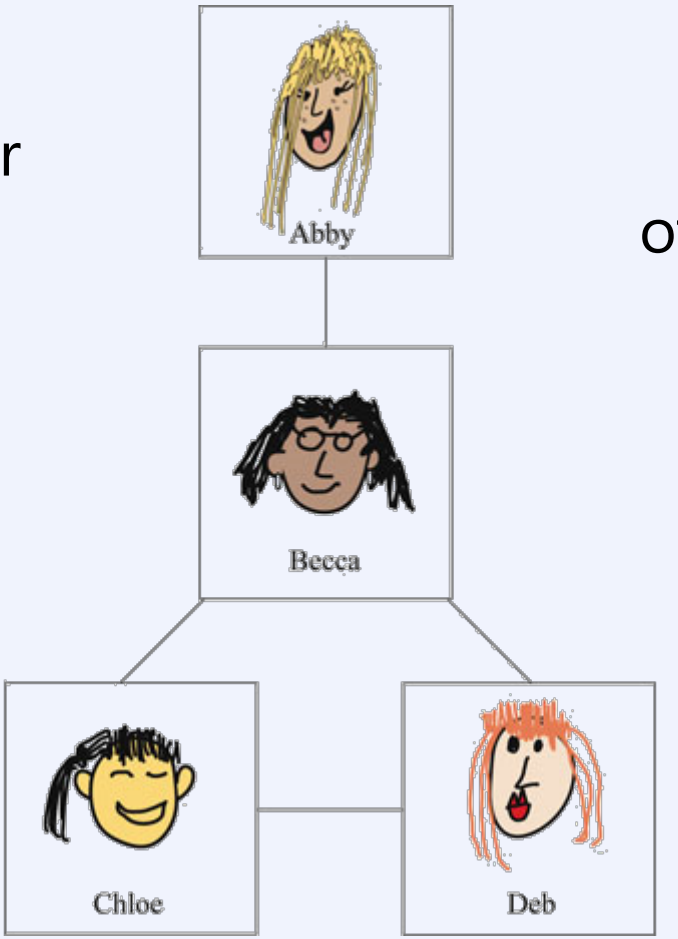
Your friends are extraverts?

There are more extraverts than introverts in the world?

Example

Average number of **friends**?

$$= \frac{1 + 3 + 2 + 2}{4}$$
$$= 2$$



Average number of **friends of friends**?

$$= (3 + 1 + 2 + 2 + 3 + 2 + 3 + 2)/8$$

$$= ((1 \times 1) + (3 \times 3) + (2 \times 2) + (2 \times 2))/8$$

$$= \mathbf{2.25}$$

(Strogatz, NYT 2012)

Always true (almost)!

Proof?

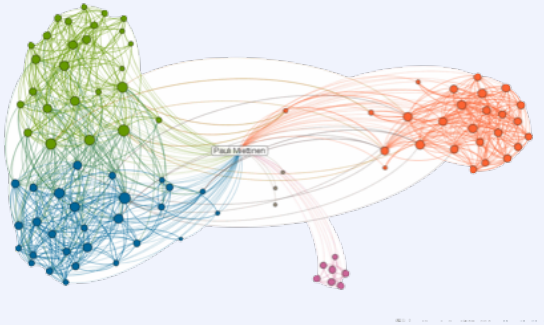
$$E[X] = \sum x_i / N$$

$$Var[X] = E[(X - E[X])^2]$$
$$= E[X^2] - E[X]^2$$

$$\frac{E[X^2]}{E[X]} = E[X] + \frac{Var[X]}{E[X]}$$

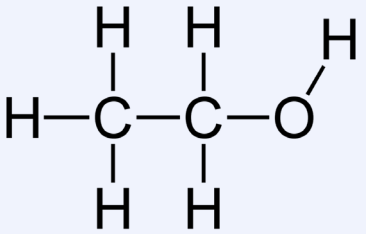
Essentially, it's true if there is **any** spread in the number of friends (i.e. whenever there's a non-zero variance).

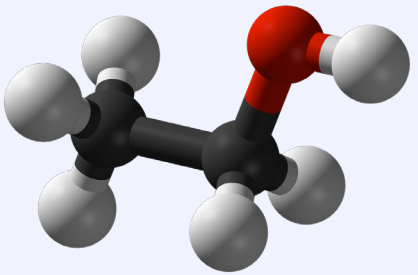
Why graphs?



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 them → a task for data mining!

What is a graph?

A **graph** G 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
  - for **undirected graphs** pairs are **unordered**, for **directed graphs** pairs are **ordered**

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 A with $(A)_{ij} = 1$ if $(v_i, v_j) \in E$

Eccentricity, radius, and 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 excluding v_i itself

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

- 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:

$$C(G) = n^{-1} \sum_i C(v_i)$$

What do to with a graph?

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) graph(s)
- areas with higher inter-connectivity than intra-connectivity
- …

Graph mining is perhaps the most popular topic in contemporary data mining research

- though not necessary called as such…

Chapter 7.2:
Properties of Graphs

Aggarwal Ch. 17.1, 19.2; Zaki & Meira Ch 4

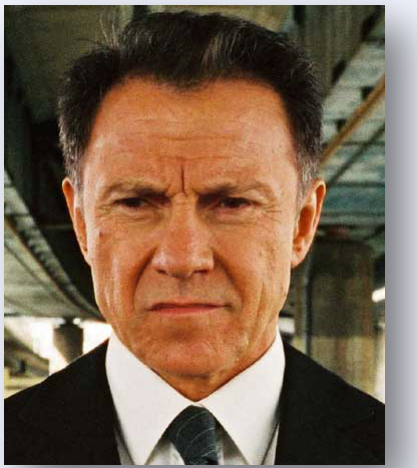
Centrality

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…
- this makes Kevin Bacon a centre of the co-acting graph

Kevin, however, is 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 $c : V \to \mathbb{R}$ inducing 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, $c(v_i) = \frac{1}{e(v_i)}$

- the least 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

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

- A is the adjacency matrix of the directed graph G
- p is n-dimensional vector giving the prestige of the vertices
- $p = A^T p$
- starting from an initial prestige vector p_0, we get

$$p_k = A^T p_{k-1} = A^T(A^T p_{k-2}) = (A^T)^2 p_{k-2} = (A^T)^3 p_{k-3} = \cdots = (A^T)^k p_0$$

Vector p converges to the dominant eigenvector of A^T

- under some assumptions

(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

It's a small world after all

A graph G is said to exhibit a **small-world property** if its average path length scales logarithmically,

$$\mu_L \propto \log n$$

- six degrees of Kevin Bacon is based on this property
- similarly so for Erdős numbers
  - how far a mathematician is from Hungarian combinatorist Paul Erdős
  - 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

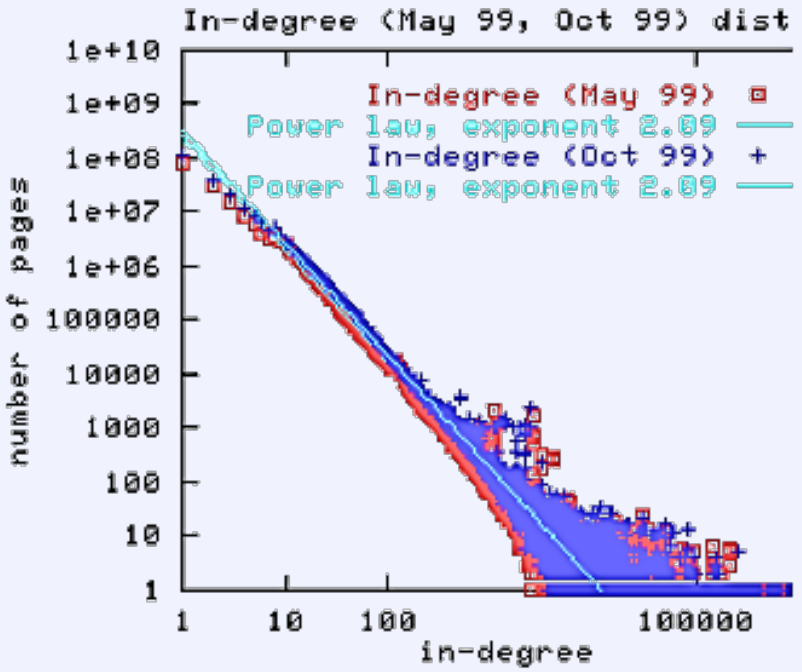
- how many vertices with degree 1, how many with degree 2, etc.
- $f(k)$ is the number of edges with degree k

A graph G is said to exhibit a **scale-free property** if $f(k) \propto k^{-\gamma}$

- follows a so-called power-law distribution
- majority of vertices have low degree, few with very high degree
- scale-free: $f(ck) = \alpha(ck)^{-\gamma} = (\alpha c^{-\gamma})k^{-\gamma} \propto k^{-\gamma}$

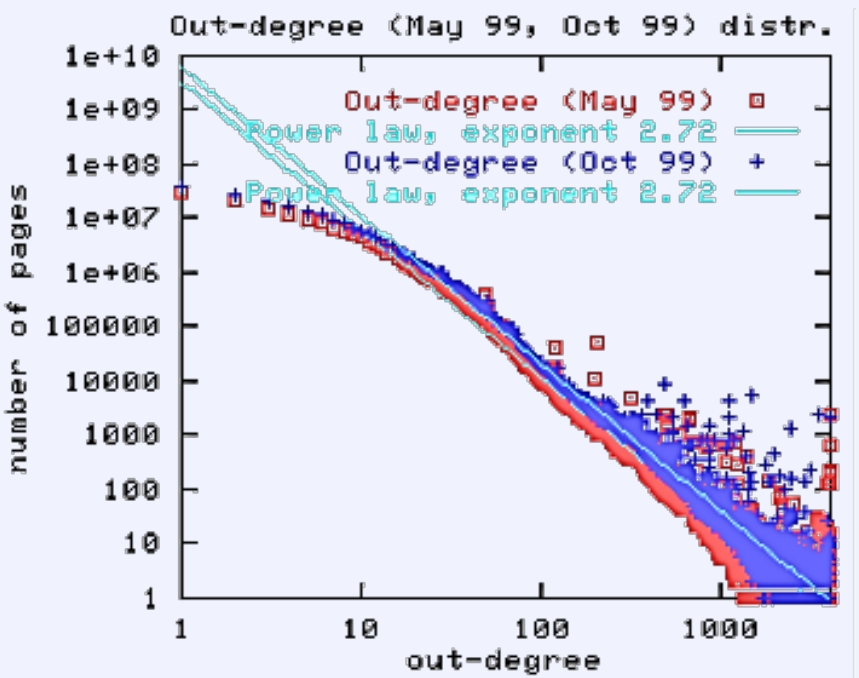
Example: WWW links

In-degree



$s = 2.09$

Out-degree



$s = 2.72$

(Broder et al.,'00)

Clustering effect

A graph exhibits the **clustering effect** if the distribution of average clustering coefficient (per degree) follows a 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 7.3:
Frequent Subgraph Mining

Aggarwal Ch. 17.2, 17.4; Zaki & Meira Ch 11

Subgraphs

Graph (V', E') is a **subgraph** of graph (V, E) iff

- $V' \subseteq V$
- $E' \subseteq E$

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 $\phi: V \rightarrow V'$ such that

- $(u, v) \in E$ if and only if $\big(\phi(u), \phi(v)\big) \in E'$
- $L(v) = L(\phi(v))$ for all $v \in V$
- $L(u, v) = L\big(\phi(u), \phi(v)\big)$ 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

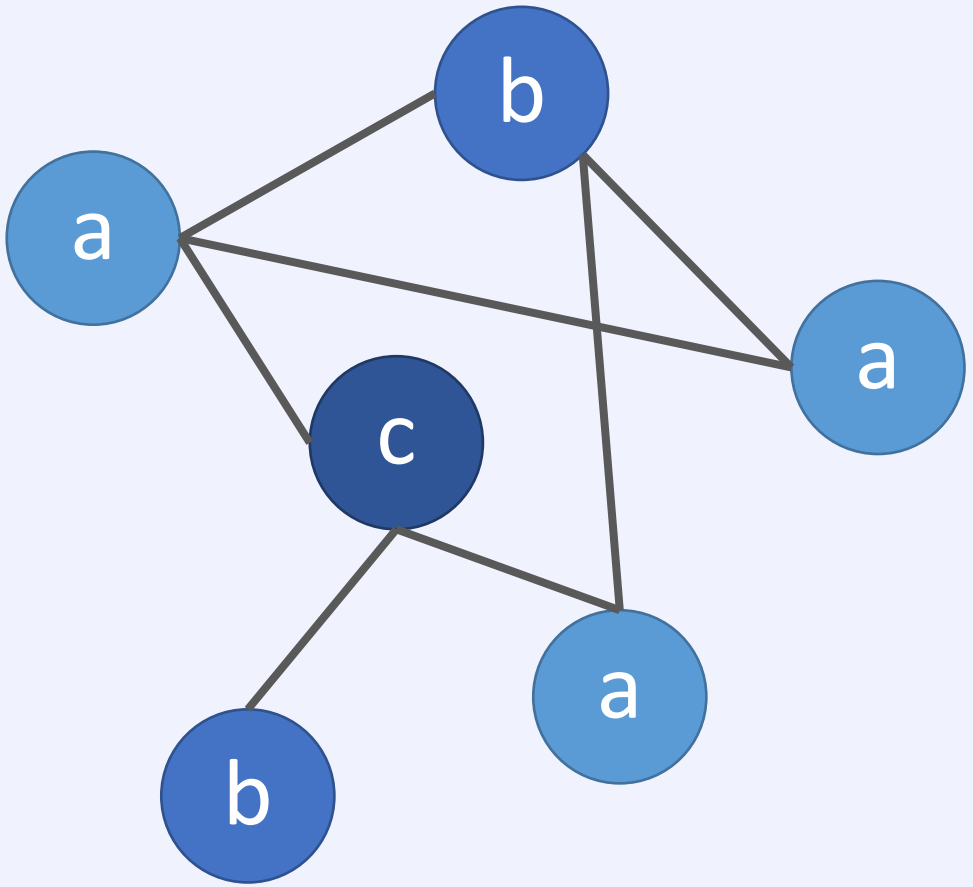
Isomorphism defines an equivalence class

- id: $V \rightarrow V$, $id(v) = v$ 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 $\phi \circ \chi$

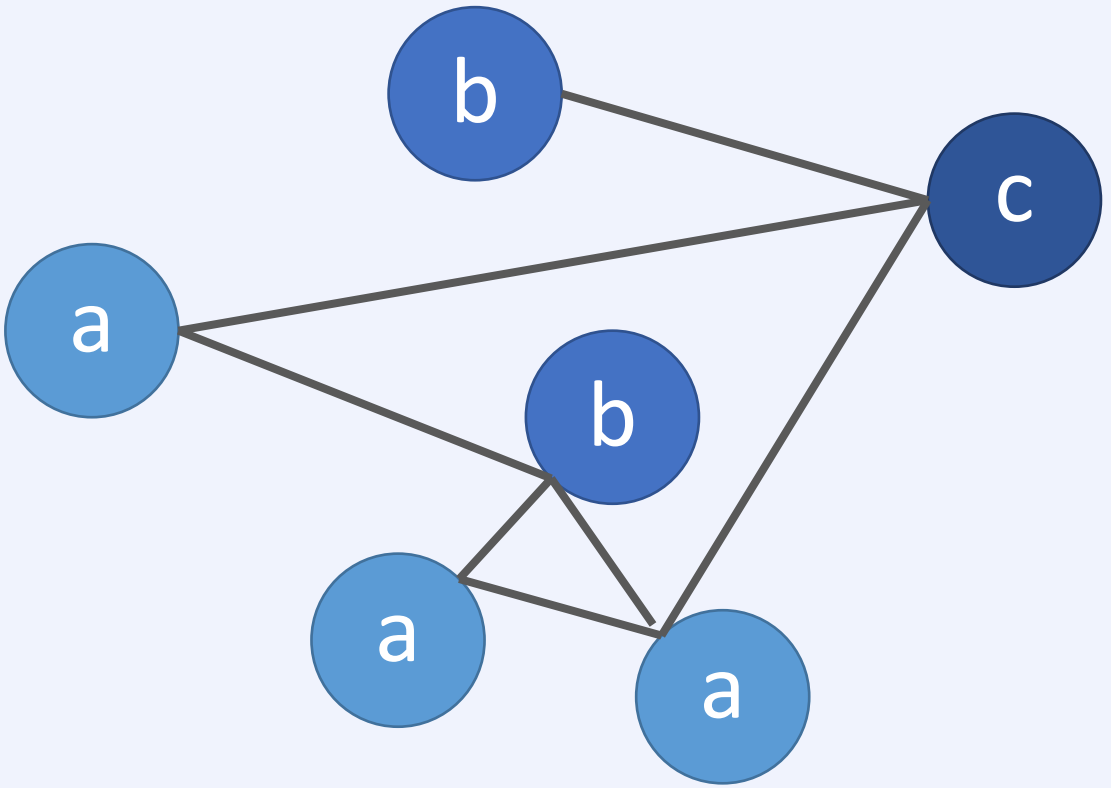
A **canonization** of a graph G, $canon(G)$ produces another graph C such that if H is a graph that is isomorphic to G, $canon(G) = canon(H)$

- two graphs are isomorphic if and only if their canonical versions are the same

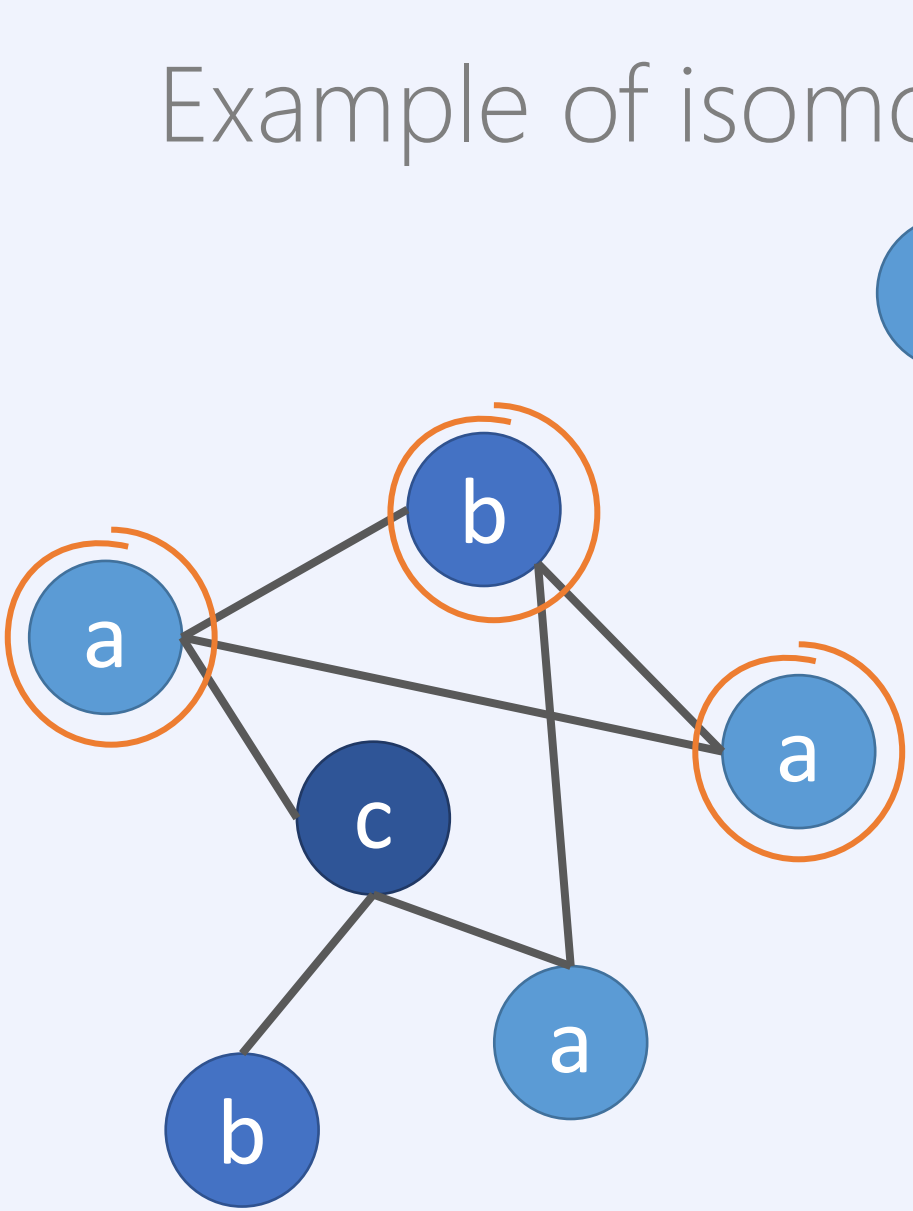
Example of isomorphic graphs



Example of isomorphic graphs



Example of isomorphic graphs



Frequent subgraph mining

Given a set $\boldsymbol{D}$ of n graphs and a minimum support σ, find all connected graphs that are subgraph isomorphic to at least σ graphs in $\boldsymbol{D}$

- enormously complex problem

For graphs that have m vertices there are

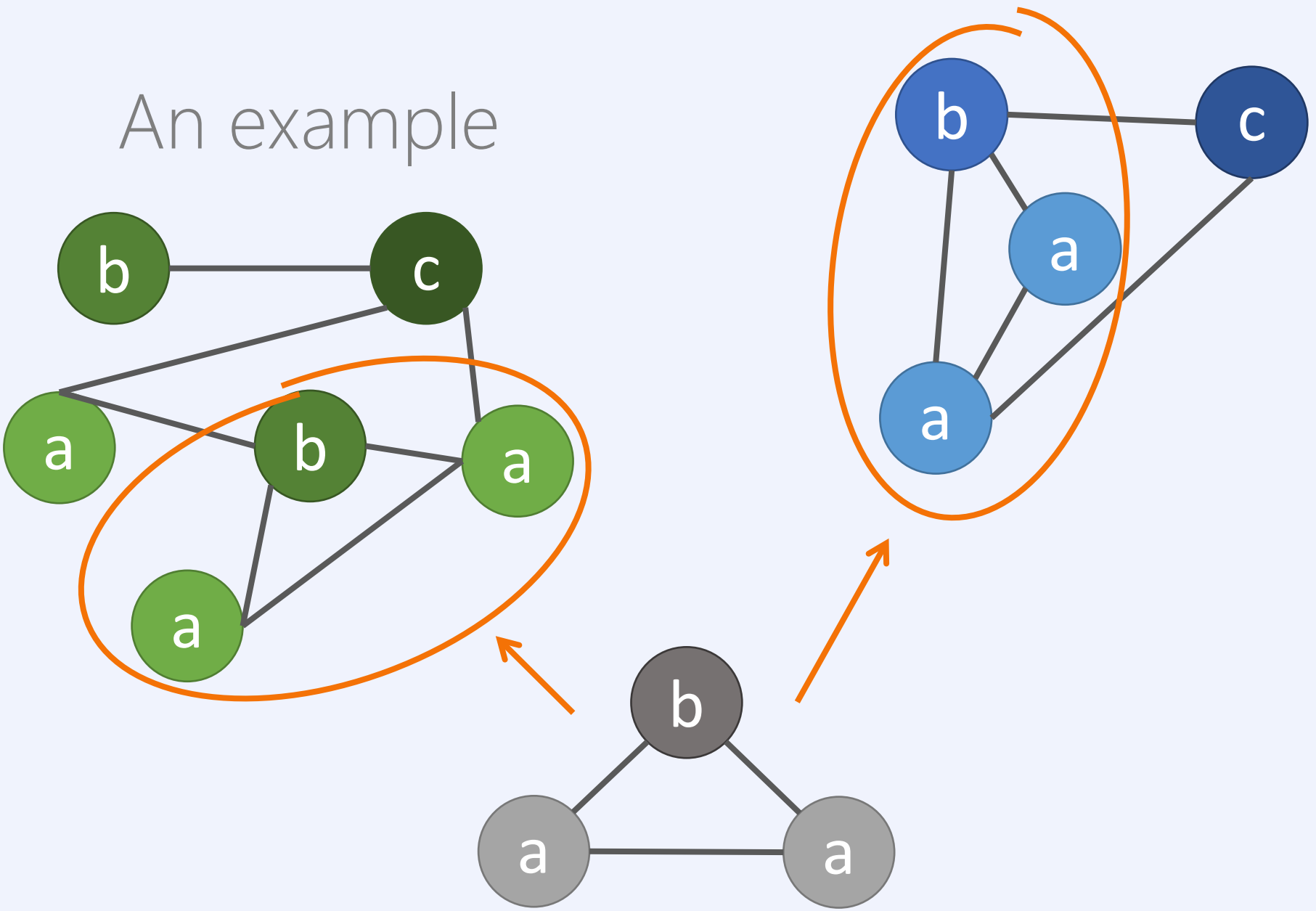
- $2^{O(m^2)}$ subgraphs (not all are connected)

If we have s labels for vertices and edges we have

- $O\left((2s)^{O(m^2)}\right)$ labelings of the different graphs

Counting support means solving **multiple** NP-hard problems

An example



Mining frequent subgraph patterns

Like for itemsets, the sub-graph definition of support is monotone

- we can employ level-wise search!

We can modify

- APRIORI to get to AGM (Inokuchi, Washio, Motoda, 2000)
- ECLAT to get FFSM (Huan, Wang, Prins, 2003)
- FP-GROWTH to get GSPAN (Pei et al., 2001)

GraphApriori

Algorithm GRAPHAPRIORI(graph db $\boldsymbol{D}$, minsup σ)
begin
  $k \leftarrow 1$;
  $\mathcal{F}_k \leftarrow \{\text{all frequent singleton graphs}\}$
  while $\mathcal{F}_k$ is not empty **do**
    Generate $\mathcal{C}_{k+1}$ by joining pairs of graphs in $\mathcal{F}_k$ that have in common a subgraph of size $(k-1)$
    Prune subgraphs from $\mathcal{C}_{k+1}$ that violate downward closure
    Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \boldsymbol{D})$ and retaining subgraphs from $\mathcal{C}_{k+1}$ with support at least σ
    $k \leftarrow k+1$
  end
  return $\bigcup_{i=1}^{k} \mathcal{F}_i$
end

(Inokuchi et al. 2000; Kuramochi & Karypis 2001)

Graph/

Algorithm GR … $\boldsymbol{D}$, m…

begin

$k \leftarrow 1$;

$\mathcal{F}_k \leftarrow \{$all fre… ns$\}$

while $\mathcal{F}_k$ is …

Generate $\mathcal{C}$… of gr…
that have… aph c…

Prune subg… t viol… e

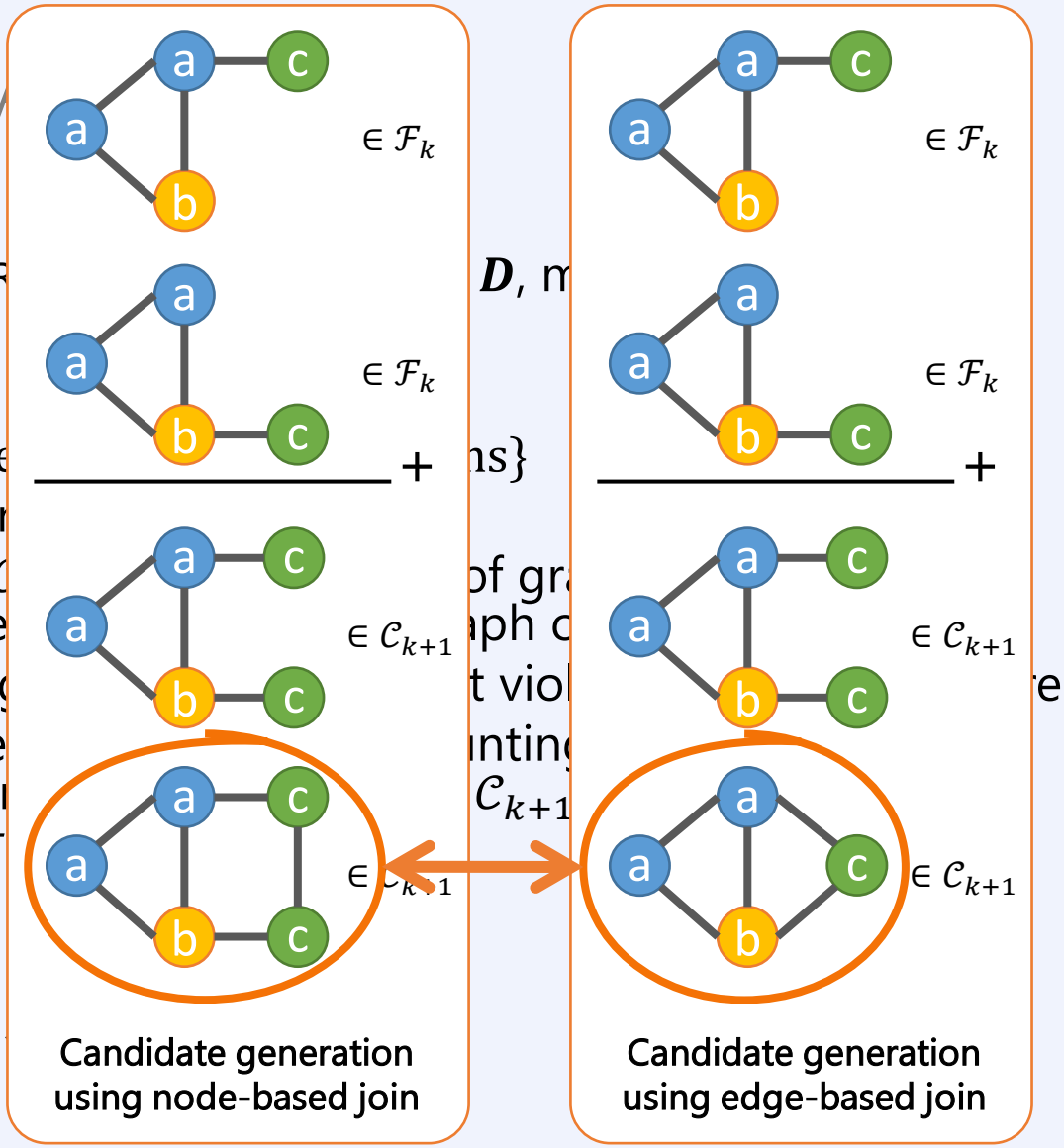
Determine… untin…
and retai… $\mathcal{C}_{k+1}$
at least σ…

$k \leftarrow k + 1$

end

return $\bigcup_{i=1}^{k}$ …

end



(Inokuchi et al. 2000; Kuramochi & Karypis 2001)

Canonical codes

We can improve the running time of frequent subgraph mining by either

- speeding up the computation of support
  - lots of efforts in faster isomorphism checking but only little progress
- creating fewer candidates that we need to check
  - level-wise algorithms generate huge numbers of candidates, all of which must be checked for isomorphism with others

The gSPAN algorithm is the frequent subgraph mining equivalent of FP-growth; it uses a depth-first approach

(Zaki & Meira chapter 11 ; Yan & Han 2002)

Depth-First Spanning tree

A depth-first spanning (DFS) tree of a graph G

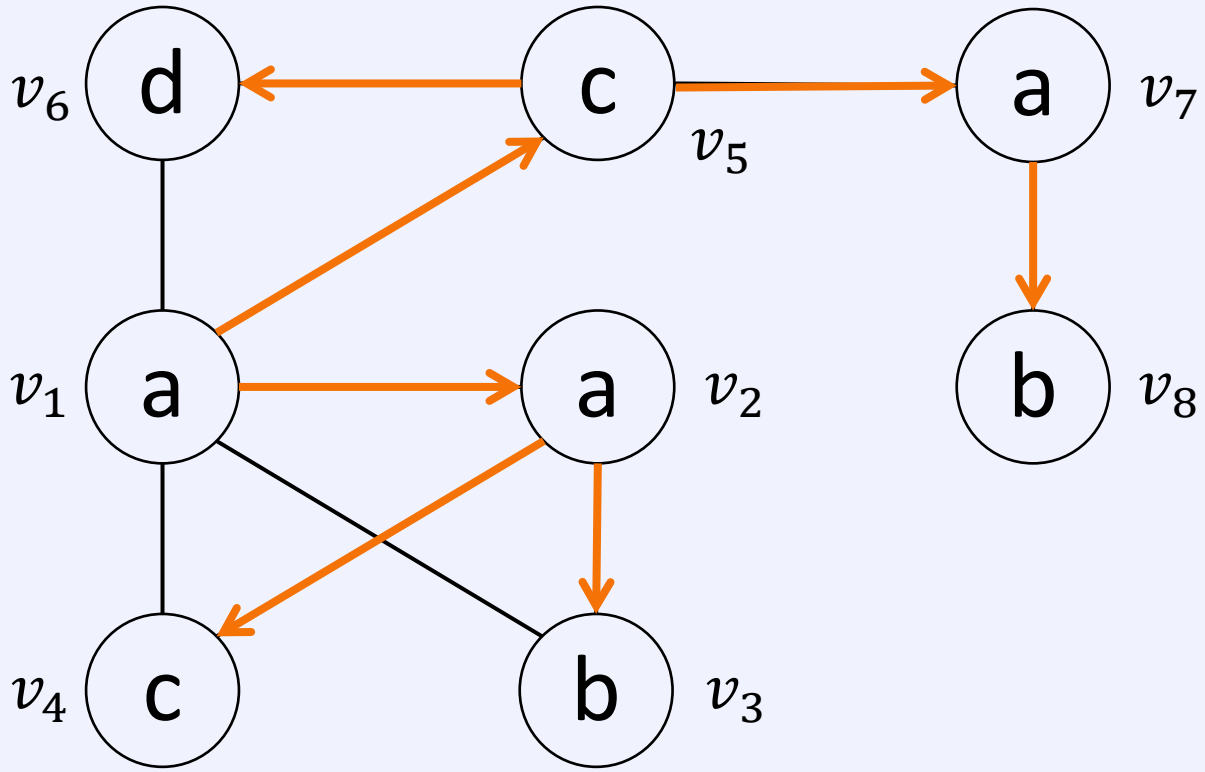
- is a connected tree
- contains all the vertices of G
- is built in depth-first order
  - selection between the siblings is e.g. based on the vertex index

Edges of the DFS tree are **forward edges**

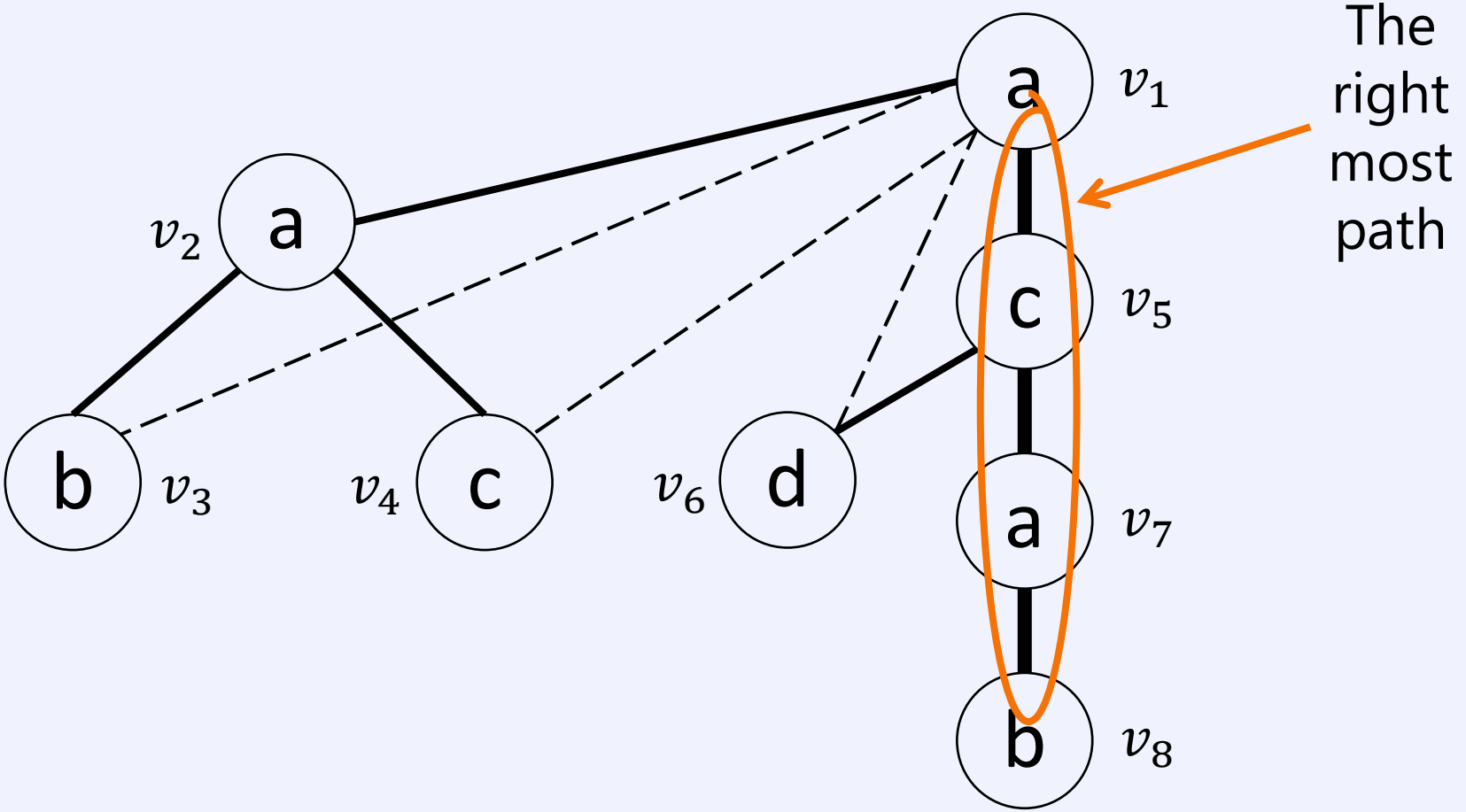
Edges not in the DFS tree are **backward edges**

A **rightmost path** in the DFS tree is the path that travels from the root to the **rightmost** vertex by always taking the **rightmost** child (last added)

An example – DFS traversal



An example – the DFS tree



Candidates from the DFS tree

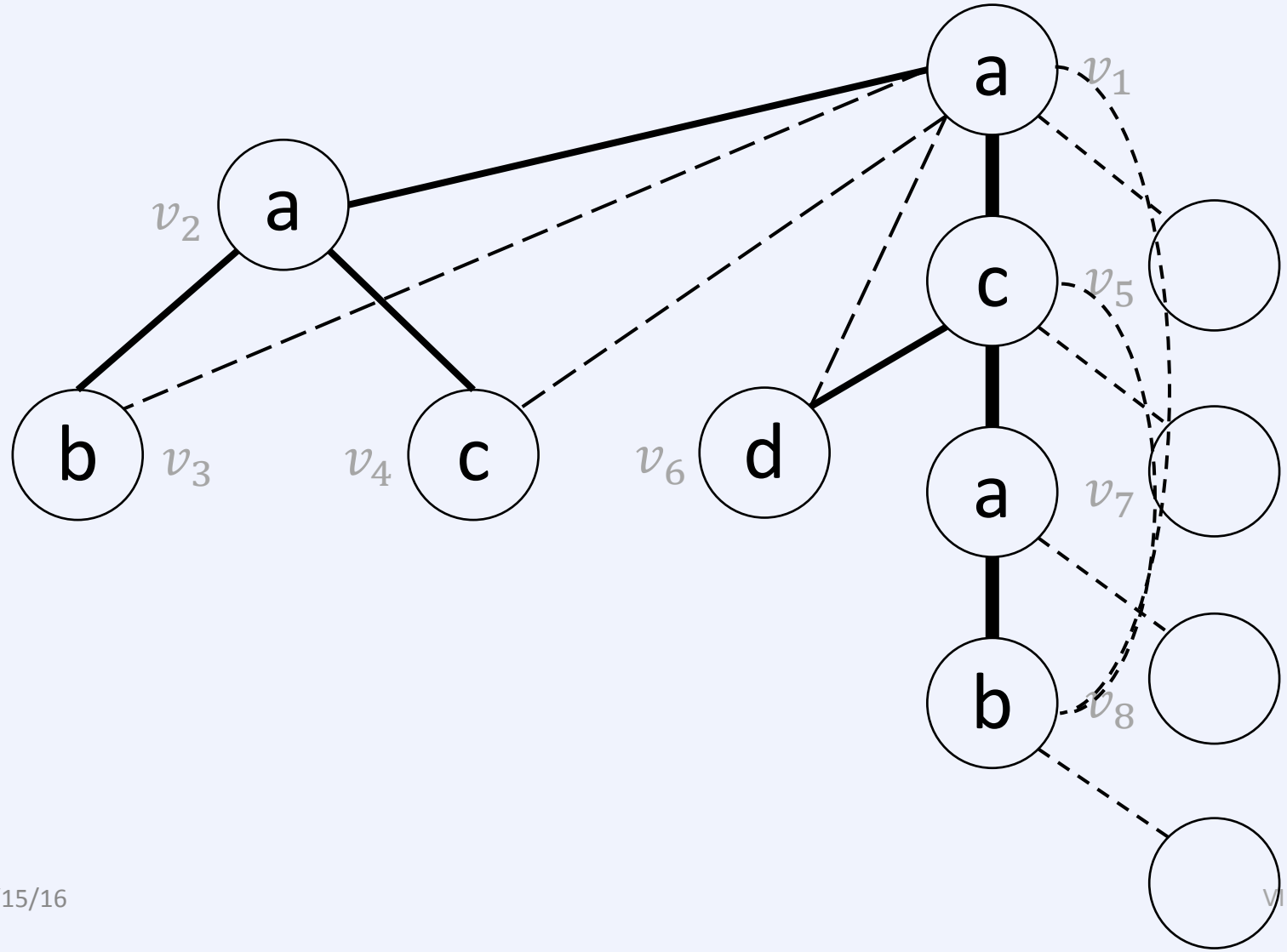
Given graph G, we extend it only from the vertices in the rightmost path

- we can add a **backward** edge 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 – the DFS tree



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 lexicographical

Ordering the edges

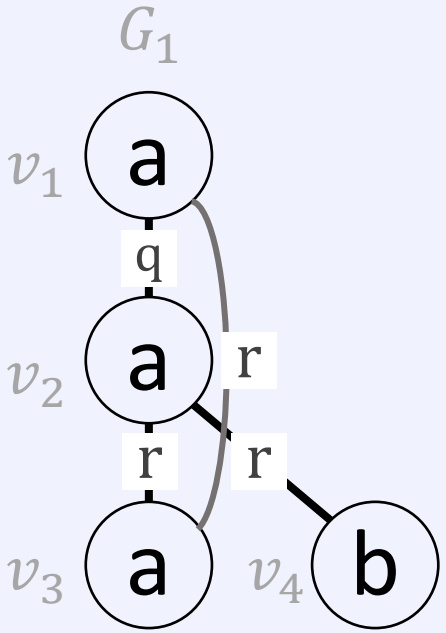
Let $e_{ij} = \langle v_i, v_j \rangle$ and $e_{xy} = \langle v_x, v_y \rangle$

$e_{ij} <_e e_{xy}$ if

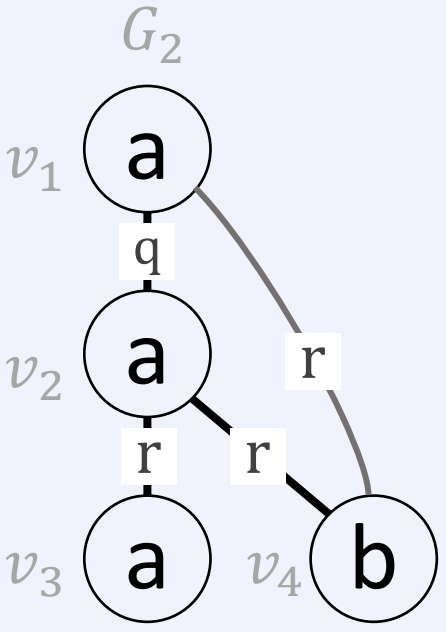
- if e_{ij} and e_{xy} are both forward edges, then
  - $j < y$; or
  - $j = y$ and $i > x$
- if e_{ij} and e_{xy} are both backward edges, then
  - $i < x$; or
  - $i = x$ and $j < y$
- if e_{ij} is forward and e_{xy} is backward, then $j \leq x$
- if e_{ij} is backward and e_{xy} is forward, then $i < y$

(typo fixed, edge order now in sync with Zaki & Meira)

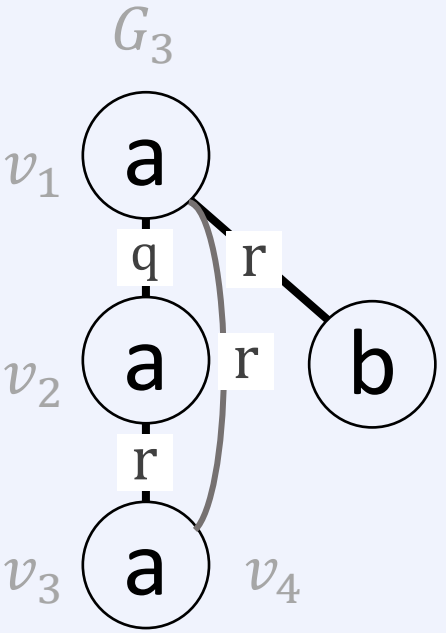
Example



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

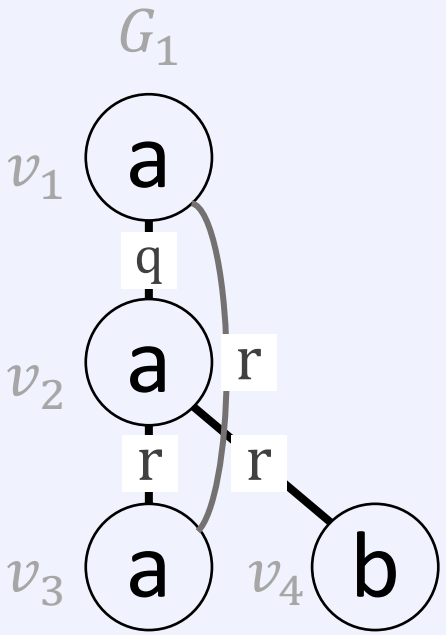


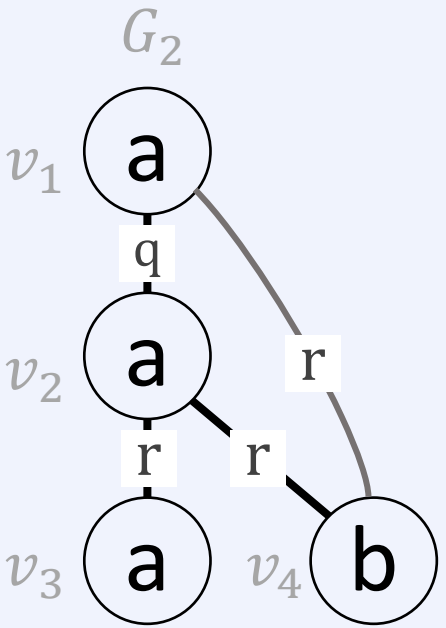
$t_{21} = \langle v_1, v_2, a, a, q \rangle$
$t_{22} = \langle v_2, v_3, a, b, r \rangle$
$t_{23} = \langle v_2, v_4, a, a, r \rangle$
$t_{24} = \langle v_4, v_1, a, a, r \rangle$

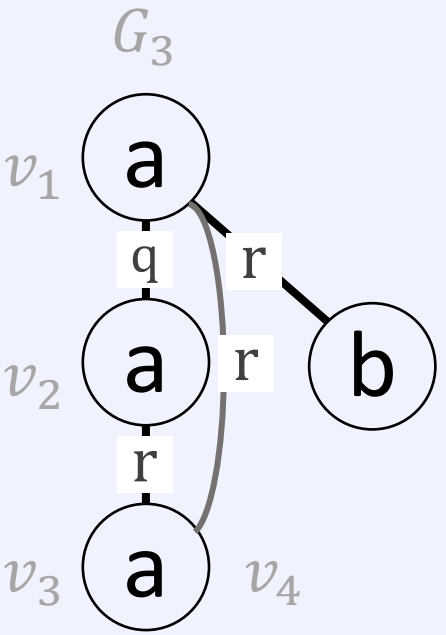


$t_{31} = \langle v_1, v_2, a, a, q \rangle$
$t_{32} = \langle v_2, v_3, a, a, r \rangle$
$t_{33} = \langle v_3, v_1, a, a, r \rangle$
$t_{34} = \langle v_1, v_4, a, b, r \rangle$

Example







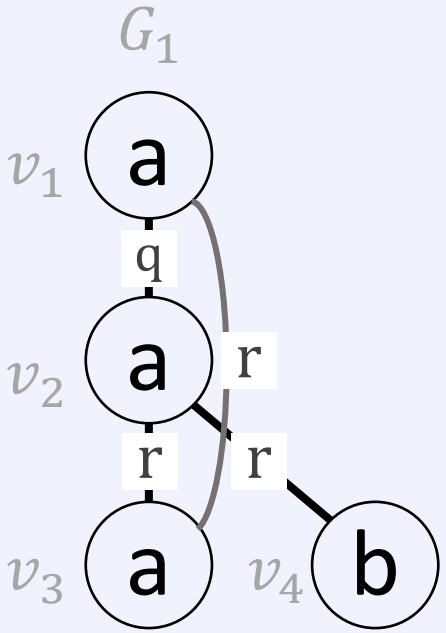
$t_{11} = \langle v_1, v_2, a, a, q \rangle$
$t_{12} = \langle v_2, v_3, a, a, r \rangle$
$t_{13} = \langle v_3, v_1, a, a, r \rangle$
$t_{14} = \langle v_2, v_4, a, b, r \rangle$

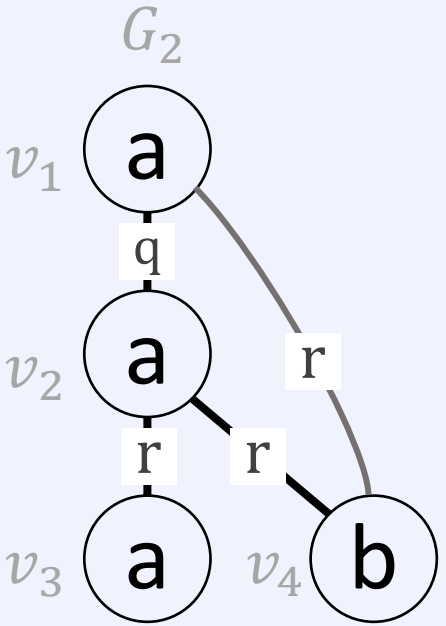
$t_{21} = \langle v_1, v_2, a, a, q \rangle$
$t_{22} = \langle v_2, v_3, a, b, r \rangle$
$t_{23} = \langle v_2, v_4, a, a, r \rangle$
$t_{24} = \langle v_4, v_1, a, a, r \rangle$

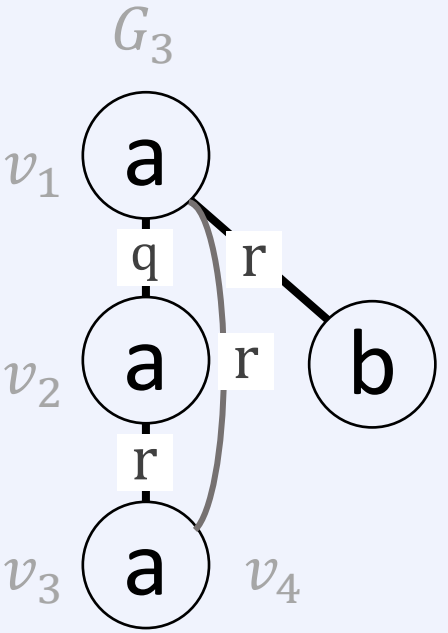
$t_{31} = \langle v_1, v_2, a, a, q \rangle$
$t_{32} = \langle v_2, v_3, a, a, r \rangle$
$t_{33} = \langle v_3, v_1, a, a, r \rangle$
$t_{34} = \langle v_1, v_4, a, b, r \rangle$

First rows are identical

Example







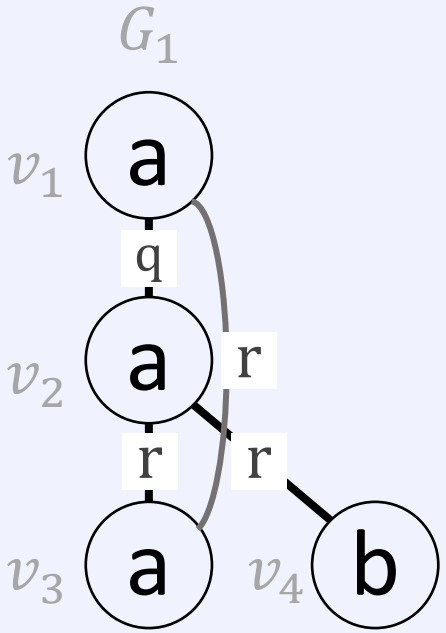
$t_{11} = \langle v_1, v_2, a, a, q \rangle$
$t_{12} = \langle v_2, v_3, a, a, r \rangle$
$t_{13} = \langle v_3, v_1, a, a, r \rangle$
$t_{14} = \langle v_2, v_4, a, b, r \rangle$

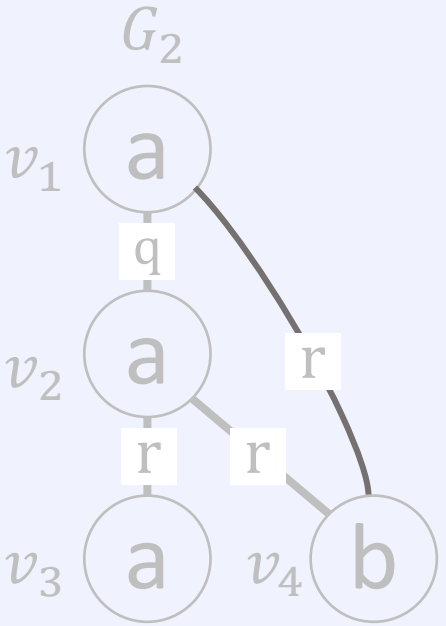
$t_{21} = \langle v_1, v_2, a, a, q \rangle$
$t_{22} = \langle v_2, v_3, a, b, r \rangle$
$t_{23} = \langle v_2, v_4, a, a, r \rangle$
$t_{24} = \langle v_4, v_1, a, a, r \rangle$

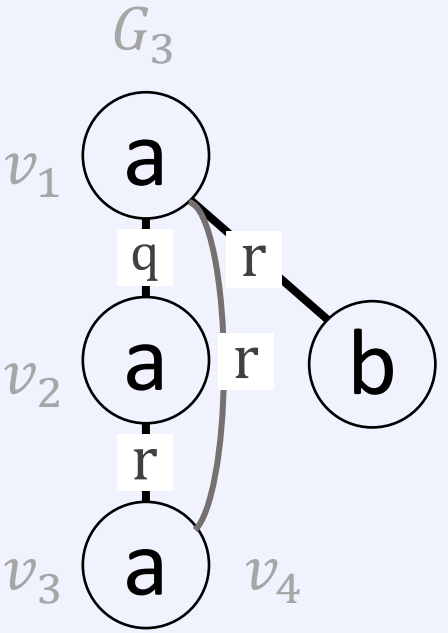
$t_{31} = \langle v_1, v_2, a, a, q \rangle$
$t_{32} = \langle v_2, v_3, a, a, r \rangle$
$t_{33} = \langle v_3, v_1, a, a, r \rangle$
$t_{34} = \langle v_1, v_4, a, b, r \rangle$

In second row, G_2 is bigger in label order

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 two rows are forward edges, and $4 = 4$ but $2 > 1 \rightarrow G_1$ is smallest

graph-based substructure pattern mining

The general idea

- use the DFS codes to create candidates
  - extend only canonical and frequent candidates

There can be very, very many extensions

- we need to see them all, and all of their isomorphisms, to count their supports

Constructing 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 if and 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 subgraphs of the graphs in the data

- but not all vertices in the code tree correspond to canonical codes

We (implicitly) traverse this tree

The gSPAN algorithm (sketch)

gSPAN(graph G, minsup σ)
  for each frequent 1-edge graph **do**
    call subgrm to grow all nodes in the tree rooted in this edge-graph
    remove this edge from the graph

SUBGRM(frequent subgraph X, minsup σ)
  if the code is not canonical **then** return
  add X to the set of frequent graphs
  create all super-graphs $Y \supset X$, extending X with one more edge
  compute frequencies of all Y's
  call SUBGRM for canonical representation of all frequent Y's

Computing frequencies

gSPAN merges extension generation and support computation

For each graph in the data base

- gSPAN computes all isomorphisms of the current candidate
  - can mean solving NP-complete problems…
- for all isomorphisms, 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

Checking 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 extension 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 do not have a canonical code

- if after k steps we arrive at the extensions code, it is canonical

Easier problems

Much of the complexity of subgraph mining lies in (checking for) isomorphisms

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

Conclusions

Graphs are everywhere
- many interesting problems
- real graphs often exhibit power-law-like behaviour

Graphs generalise many data settings
- makes it possible to create general algorithms

Many problems in graphs are very difficult
- subgraph isomorphism

Frequent subgraph mining
- involves multiple NP-hard problems

Thank you!

Graphs are everywhere

- many interesting problems
- real graphs often exhibit power-law-like behaviour

Graphs generalise many data settings

- makes it possible to create general algorithms

Many problems in graphs are very difficult

- subgraph isomorphism

Frequent subgraph mining

- involves multiple NP-hard problems