Subgraph Matching

COMP9312_24T2

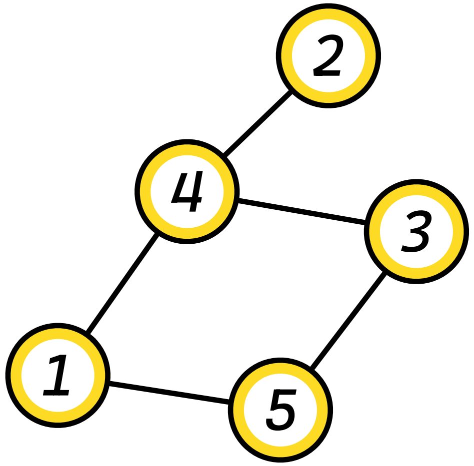
UNSW SYDNEY

Outline

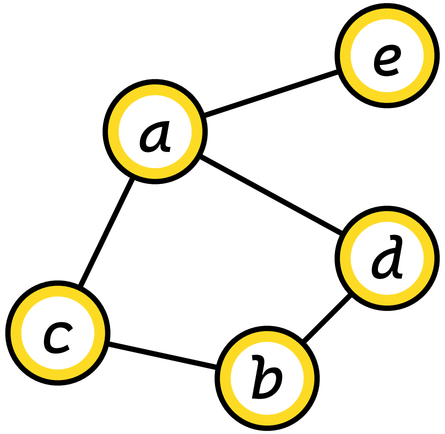
- Basic Concepts
- Triangle Counting
- General Subgraph Matching

Graph Homomorphism

Two graphs G and H are homomorphic, if there exists a function $f\colon V_G \rightarrow V_H$ between vertices of the graph such that if $\{a, b\}$ is an edge in G then $\{f(a), f(b)\}$ is an edge in H.



G

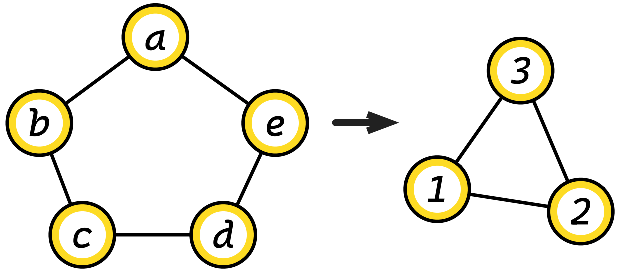


H

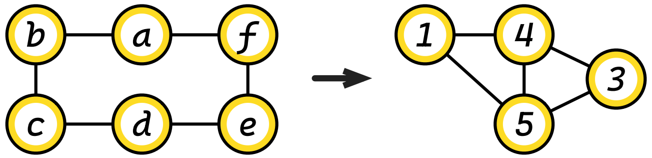
An ideal case...

Graph Homomorphism (cont)

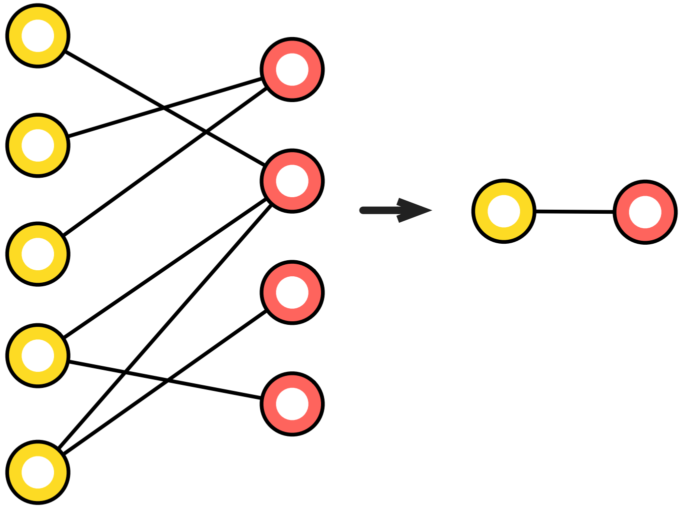
Some other cases …




```
f(a) = 1 f(b) = 2
f(c) = 1 f(d) = 2
      f(e) = 3
```



Try to find the mapping function

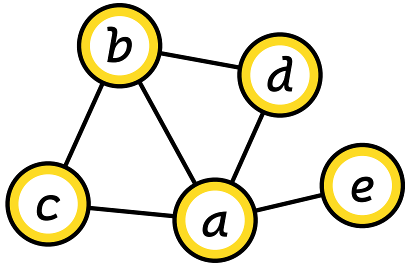


All bipartite graphs can be mapped into an edge

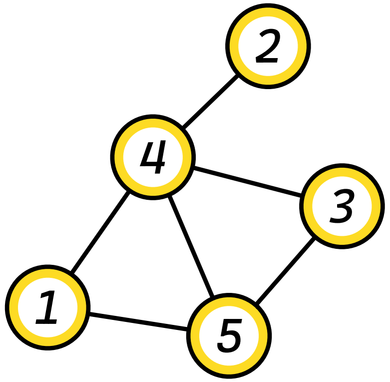
Graph Isomorphism

Two graphs G and H are isomorphic, if there exists a **bijection** $f\colon V_G \to V_H$ between vertices of the graph such that if $\{a,b\}$ is an edge in G then $\{f(a),f(b)\}$ is an edge in H.

G



H



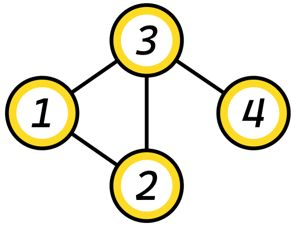
f(a) = 4		f(a) = 4
f(b) = 5		f(b) = 5
f(c) = 1	or	f(c) = 3
f(d) = 3		f(d) = 1
f(e) = 2		f(e) = 2

There is a one-to-one vertex correspondence in Graph Isomorphism.

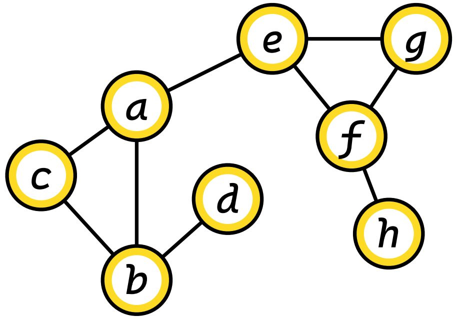
Subgraph Matching

Given a query graph Q and a data graph G, compute (count/enumerate) all subgraphs of G that are isomorphic to Q.

All matching instances of $f(1), f(2), f(3), f(4)$:



Query graph



Data graph

Subgraph Matching

Given a query graph Q and a data graph G, compute (count/enumerate) all subgraphs of G that are isomorphic to Q.

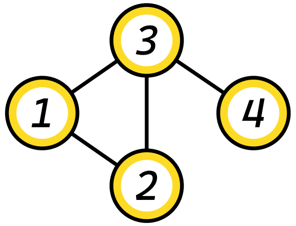
All matching instances of ***f(1),f(2),f(3),f(4)***:

<b,c,a,e> <c,b,a,e>

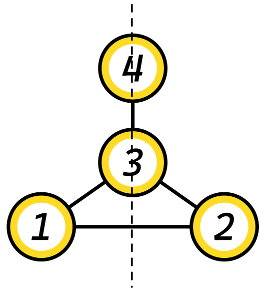
<a,c,b,d> <c,a,b,d>

<g,f,e,a> <f,g,e,a>

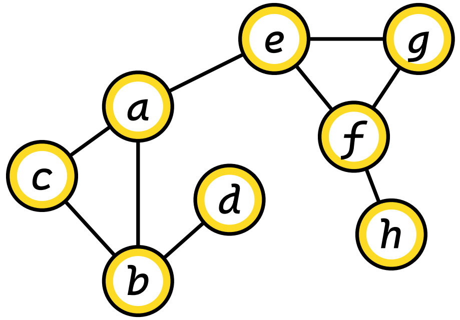
<e,g,f,h> <g,e,f,h>



Query graph



The query graph is symmetric



Data graph

Application

- Bioinformatics of protein-protein interaction networks
- Chemistry (similarity between chemical compounds)
- Node representation
- Malware detection
- System analysis
- …

Subgraph Matching

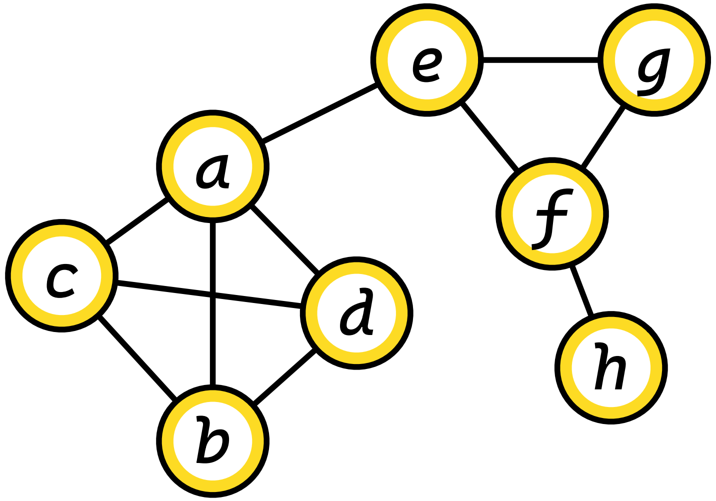
- Exact counting/enumeration
  - Specific patterns (triangle, k-clique, butterfly, …)
  - General subgraphs

- Approximate counting (estimation)
  - Probability theory
  - Graph Neural Networks

Triangle Counting

Triangle Counting

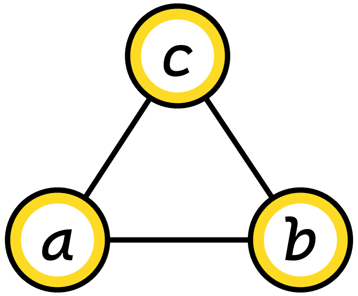
Count all triangles in a graph.



Five triangles exist in this example.

Triangle Counting

Edge-iterator: Given an edge (u, v), any triangle that includes the edge must contain a third vertex w that has connections to both of u and v. Thus, we can obtain any triangles containing edge (u, v) based on the intersection of N(u) and N(v). For each edge, the edge-iterator returns the set of triangles associated with that edge, and when repeated on all edges, the set of all triangle solutions is made available.



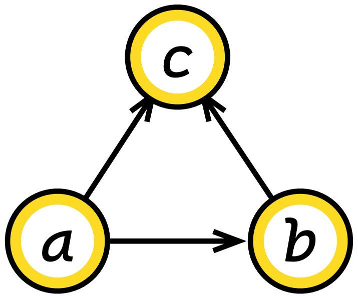
Duplication: If we use the above method to calculate the triangles, then we will count a triangle repeatedly.

Latapy, M. (2008). Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, *407*(1-3), 458-473.

Triangle Counting

Priority: The priority of vertices can be determined by the degree of each vertex. The smaller the degree; the smaller the priority. If the degrees of two vertices are the same, it can be determined by the alphabetical order or numerical order of the vertices.

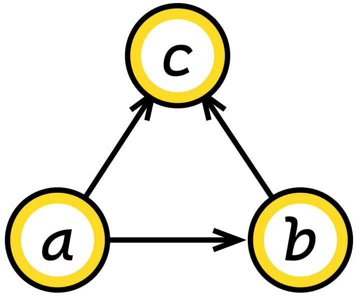
Orientation technique (without duplication): Each undirected edge is mapped to a directed edge where the direction (i.e., orientation) is decided by the priority of its endpoints in the vertex-ordering (i.e., **u->v if u has a higher priority than v**). **We refer to vertex u as a pivot vertex if u has two out-going edges.** We can association a triangle in the undirected graph with only one pivot vertex to ensure one and only one instance of this triangle in the output, which significantly improves the performance.



Latapy, M. (2008). Main-memory triangle computations for very large (sparse (power-law)) graphs. Theoretical computer science, 407(1-3), 458-473.

Triangle Counting

Compact Forward (CF) Algorithm: We denote the set of outgoing-neighbors of vertex u in G as $N^+(u)$, and the out-degree as $deg^+(u) = |N^+(u)|$. In line 1, undirected graph G is transformed into a directed graph G via the orientation technique. (Line 2 onward follows the edge-iterator framework.) In Line 3, triangles are enumerated by iterating through the outgoing neighborhoods rather than the full neighborhood. In Line 4, a merge-based intersection identifies the common out-going neighbors of u and v, denoted by T. A set of triangles (u, v, w) is then output for every vertex w ∈ T.



Algorithm 1: CF(G)

Input : G : an undirected graph
Output : All triangles in G

1 $\boldsymbol{G} \leftarrow$ Orientation graph of G based on degree-order;
2 **for** each vertex $u \in \boldsymbol{G}$ **do**
3   **for** each out-going neighbor v **do**
4         $T \leftarrow N^+(u) \cap N^+(v)$;
5   **for** each vertex $w \in T$ **do**
6   Output the triangle (u, v, w);

Triangle Counting: Naïve Join

The adjacency list of each vertex is unsorted: The time complexity of CF algorithm is: $O(\sum_{(u,v)\in E} deg^{+}(u) * deg^{+}(v))$.

Proof: We need to check whether there are common neighbors in the adjacency list of u and v. For each neighbor of u, we need to check $deg^{+}(v)$ times in the adjacency list of v, u has $deg^{+}(u)$ neighbors, so the time complexity of finding common neighbor of two vertices (line 4 in the pseudo code) is $O(deg^{+}(u) * deg^{+}(v))$. Thus, the total time complexity of CF algorithm is: $O(\sum_{(u,v)\in E} deg^{+}(u) * deg^{+}(v))$.

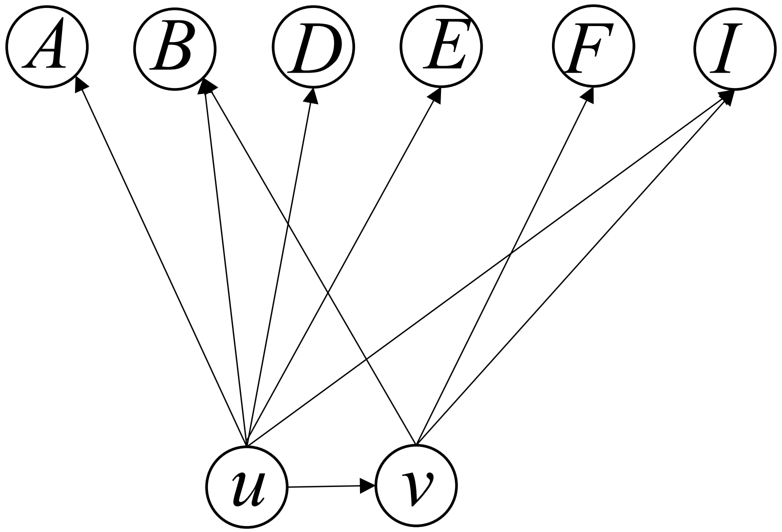
Adjacency list of u:

$N^{+}(u)$	B	A	D	I	E

Adjacency list of v:

$N^{+}(v)$	B	F	I

For the sake of brevity, here we only give the out-neighbors of u and v and the degree of vertices other than u, v is higher than the degree of u, v.



Triangle Counting

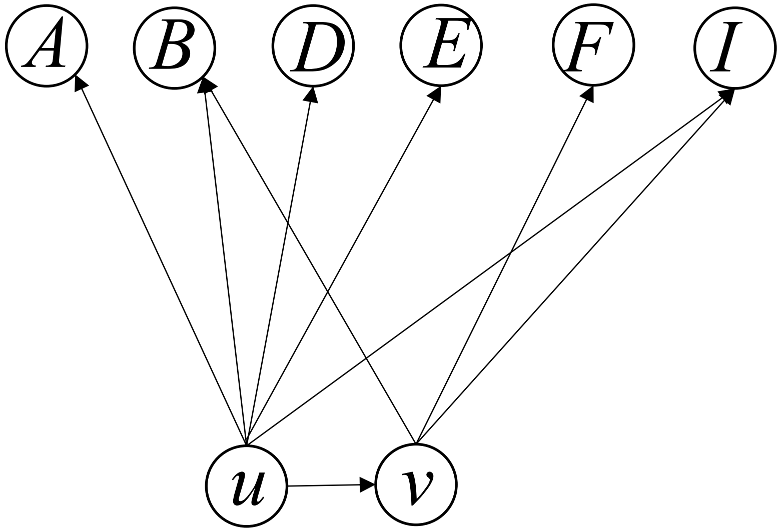
If the adjacency list of each vertex is sorted, the merge-based intersection operation at Line 4 takes $O(deg^+(u) + deg^+(v))$. The time complexity of CF algorithm is: $O(\sum_{(u,v)\in E} deg^+(u) + deg^+(v))$.

Ordered adjacency list of u:

$N^+(u)$	**A**	**B**	**D**	**E**	**I**

Ordered adjacency list (with order) of v:

$N^+(v)$	**B**	**F**	**I**



Triangle Counting

Suppose a hash table has been built for each vertex based on the out-going neighbors in the oriented graph. At Line 4 of Algorithm CF, we may choose the vertex with larger number of neighbors as the hash table for intersection operation with $O(\min(deg^+(u),\ deg^+(v)))$ look-up cost.

A hash table has been built for each vertex: The time complexity of CF algorithm is:
$O(\sum_{(u,v)\in E} \min(deg^+(u),\ deg^+(v)))$ = $O(\sum_{(u,v)\in E} \min(deg(u),\ deg(v)))$ = $O(\alpha\ \cdot m)$ = $O(m^{1.5})$.

Graph arboricity

https://en.wikipedia.org/wiki/Arboricity

Triangle Counting

Adjacency list of u: $N^{+}(u)$	A	B	D	E	I
Adjacency list of v: $N^{+}(v)$	B	F	I		

Choosing the vertex with larger number of neighbors as the hash table for intersection operation.

0	→ E
1	→ A
2	→ B
3	
4	→ D → I
5	
6	
…	

Here is an example of u's hash table. We use division hashing to make this hash table. The value of the vertex is replaced by the corresponding number (A: 1, B: 2, D: 4, E: 5, I: 9) and the hash function used is X % 5.

Because the query time complexity of hash table is O(1), we only need to query $deg^{+}(v)$ times.

Final result: there are two common neighbors B and I.

Triangle Counting

- Building a hash table for neighbors of each vertex takes too much space for big graphs.
- Utilize the degree order
- Scan the smaller set -> scan the out neighbor of v

A hash table is being built on the fly: The time complexity of CF algorithm is:

$O\left(\sum_{(u,v)\in E'} deg^{+}(v)\right) = O\left(\sum_{(u,v)\in E'} deg(v)\right) = O\left(\sum_{(u,v)\in E} \min(\deg(u), deg\,(v))\right)$

$= O(\alpha \cdot m) = O(m^{1.5})$.

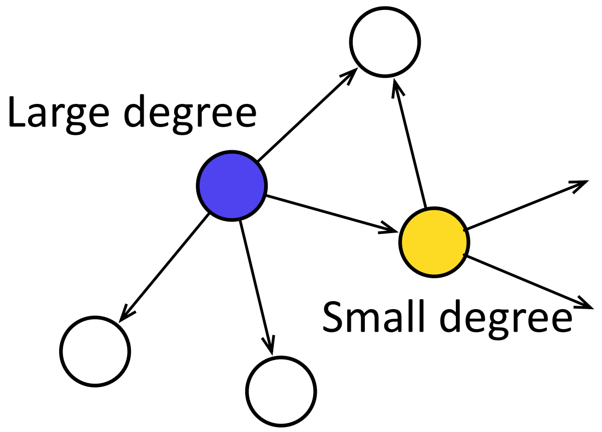
E': the set of all directed edges

Coding practice~

Further Optimization?

Optional

[2020 DASFAA] AOT: Pushing the Efficiency Boundary of Main-memory Triangle Listing



Optimization in previous slide:
Record the neighbor of large degree
Scan the neighbor of small degree

Observation: are we scanning all neighbors?
Only scan out neighbors -> compare out-degree instead of degree

$$O\left(\sum_{(u,v)\in E} \min(\deg(u), deg\ (v))\right) \Rightarrow O\left(\sum_{(u,v)\in E} \min(deg^{+}(u), deg^{+}\ (v))\right)$$

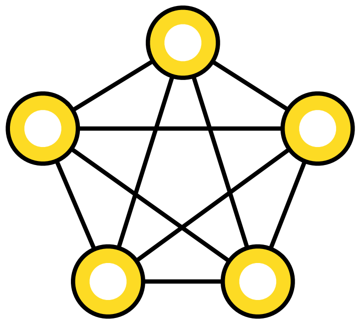
See details in: https://cgi.cse.unsw.edu.au/~cs9312/24T2/DASFAA_2020.pdf

K-Clique Enumeration

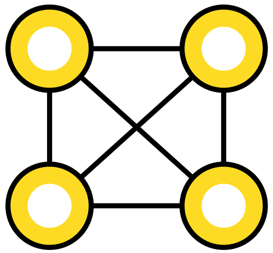
A clique is a graph in which every pair of vertices are connected.

A k-clique is a clique with k vertices.

Can you design an algorithm to enumerate all k-cliques?



5-clique



4-clique

General Subgraph Matching

Subgraph Matching

Optional

Avoid redundancy for symmetric query graphs by enforcing $f(1)<f(2)$

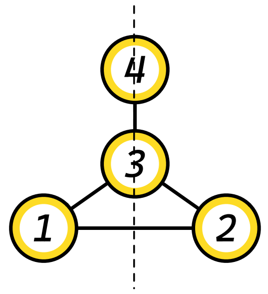
All matching instances of $f(1),f(2),f(3),f(4)$:

<a,c,b,d> ~~<c,a,b,d>~~

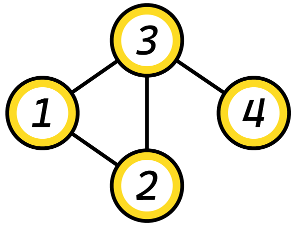
<b,c,a,e> ~~<c,b,a,e>~~

<f,g,e,a> ~~<g,f,e,a>~~

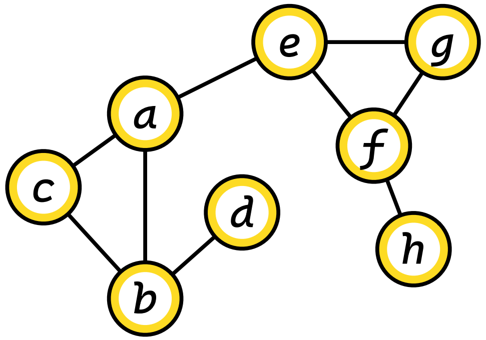
<e,g,f,h> ~~<g,e,f,h>~~



1 and 2 are equivalent in this case



Query graph

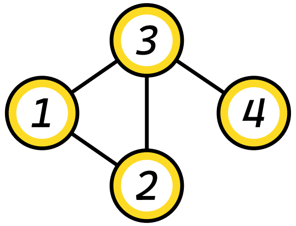


Data graph

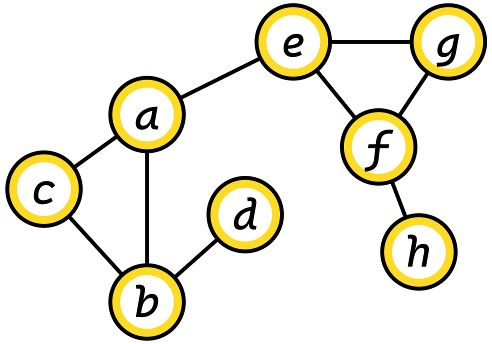
Subgraph Matching

Optional

1. Set up a matching order
2. Match following the order
3. Apply pruning rules



Query graph

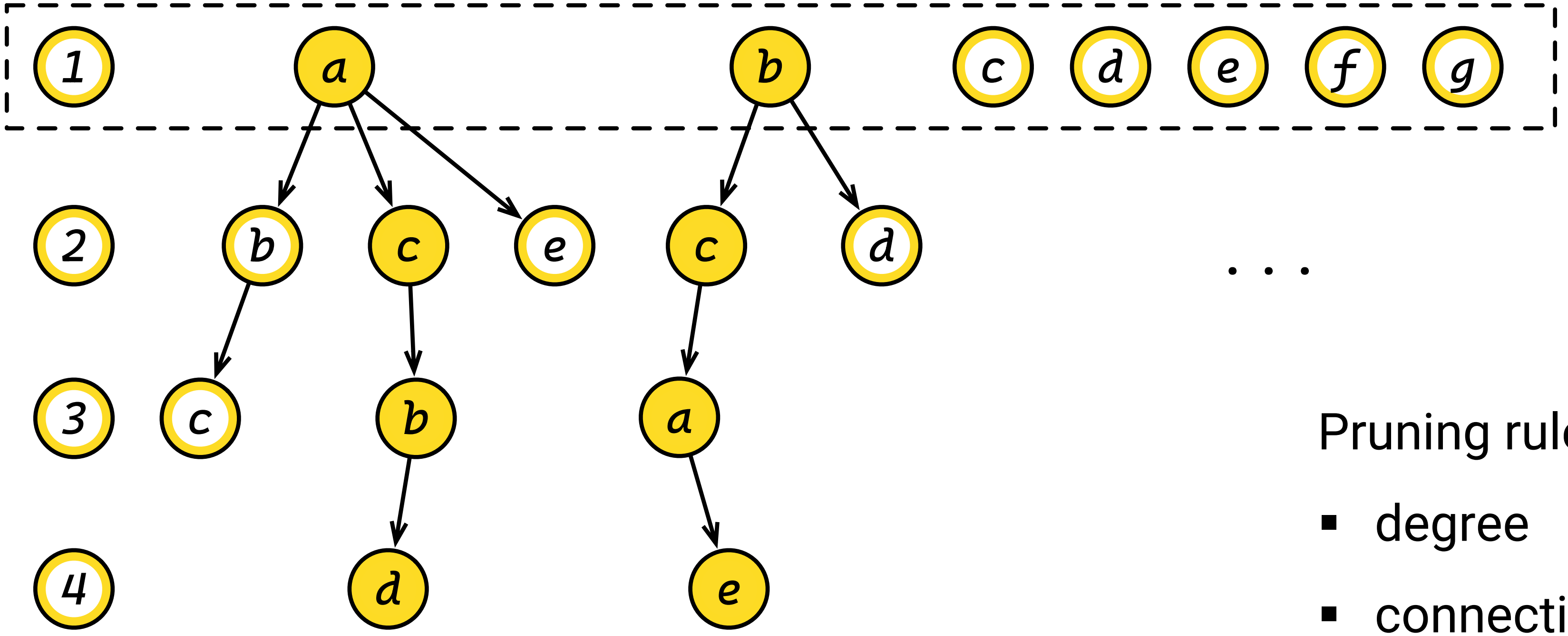


Data graph

Subgraph Matching

Optional

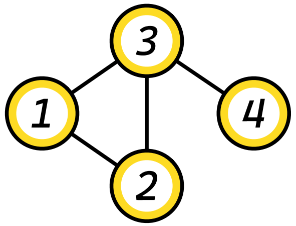
Matching order <1,2,3,4>



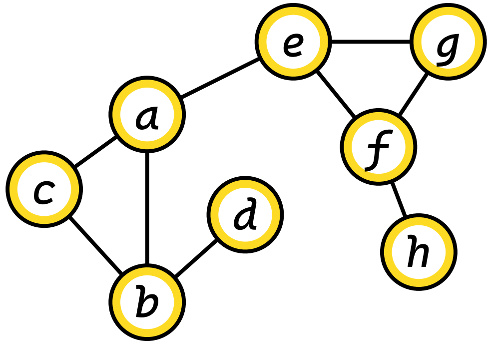
Matching tree

Pruning rules:

- degree
- connectivity



Query graph

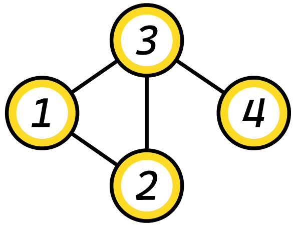


Data graph

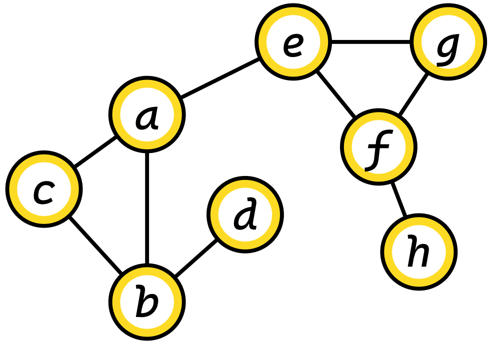
Further Optimization

Optional

- Matching plan
  - order plan (vertex-based)
  - join plan (vertex-based)
- Efficient common neighbor computation



Query graph



Data graph