CSC 522 - Automated Learning and Data Analysis

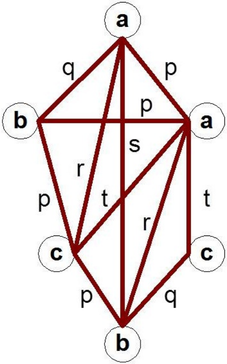
Graph Mining

Srinath Ravindran

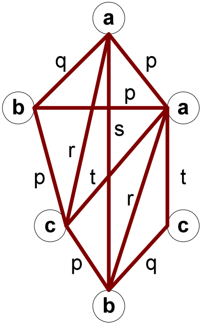
Department of Computer Science
North Carolina State University

Graphs - Brief Introduction

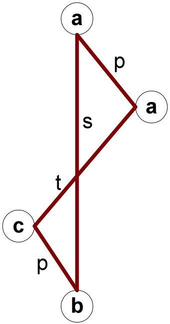
- A graph is an ordered pair $G = (V, E)$ comprising a set V of vertices or nodes together with a set E of edges or lines that connect the vertices.
- The size of a graph is the number of vertices $= |V|$



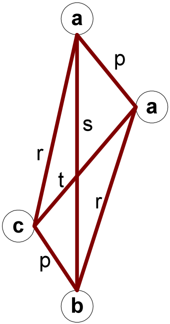
Graph Definitions



(a) Labeled Graph



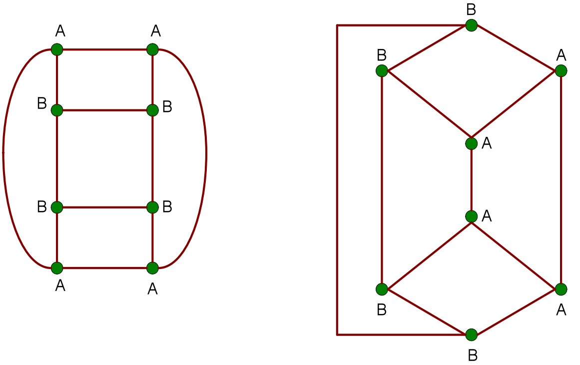
(b) Subgraph

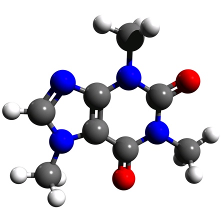


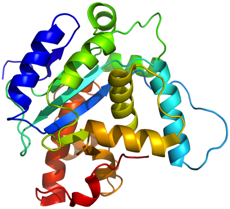
(c) Induced Subgraph

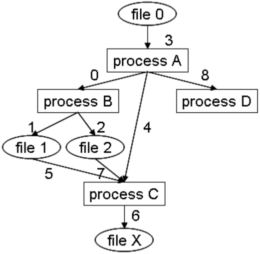
Graph Isomorphism

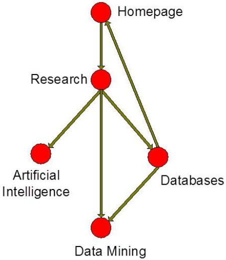
- A graph is isomorphic if it is topologically equivalent to another graph











PS. You may be interested in this paper:
The NFL Coaching Network: Analysis of the Social Network Among Professional Football Coaches, Fast and David Jensen, AAAI 2006

`http://sports.espn.go.com/nfl/columns/story?columnist=wickersham_seth&id=4781314`

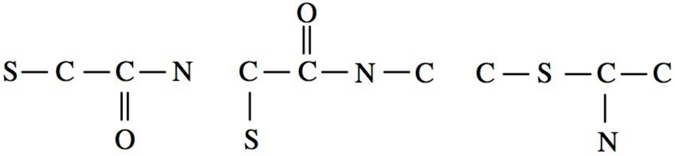
So, what can we do with these graphs?

- Mining Frequent Subgraphs
- Graph Classification
- Graph Clustering

Frequent Subgraph Mining

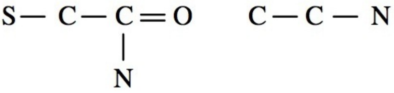
- A (sub)graph is frequent if its support (occurrence frequency) in a given dataset is no less than a minimum support threshold
- Applications
  1. Mining biomolecular/ Chemical structures - to identify the most common cores in active compounds
  2. Program control flow analysis
  3. Mining XML structures or Web communities
  4. Building blocks for graph classification, clustering, compression, comparison, and correlation analysis

Example



(g_1) (g_2) (g_3)

Sample graph dataset.



Frequency:2 Frequency:3

Source: Mining Graph Data, Diane Cook and Larry Holder, Wiley, 2007.

What are the Challenges?

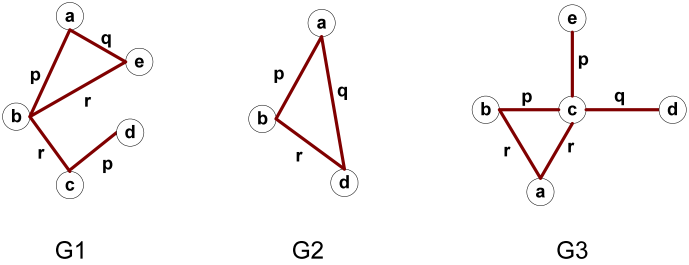
What are the Challenges?

- How do we represent the graphs?
- How do we generate candidates of size $(k+1)$ given a structure of size k?
- What is support and how do we count support?
- Assumption: frequent subgraphs must be connected
- Oh wait! aren't most of these computationally complex?

What are the Challenges?

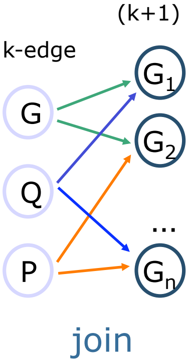
- How do we represent the graphs?
- How do we generate candidates of size $(k+1)$ given a structure of size k?
- What is support and how do we count support?
- Assumption: frequent subgraphs must be connected
- Oh wait! aren't most of these computationally complex? An n-edge frequent graph may have 2^n subgraphs!

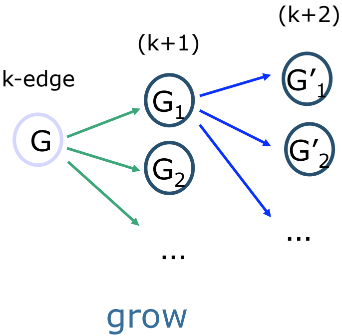
Representing Graphs as Transactions



	(a,b,p)	(a,b,q)	(a,b,r)	(b,c,p)	(b,c,q)	(b,c,r)	…	(d,e,r)
G1	1	0	0	0	0	1	…	0
G2	1	0	0	0	0	0	…	0
G3	0	0	1	1	0	0	…	0
G3	…	…	…	…	…	…	…	…

Generation of Candidate Patterns





Apriori-Based Approach VS. Pattern-Growth Approach

Apriori Based Approach

- Apriori-like approach: Use frequent k-subgraphs to generate frequent (k+1) subgraphs
- That's fine... but does Apriori Principle hold?

Apriori Based Approach

- Apriori-like approach: Use frequent k-subgraphs to generate frequent (k+1) subgraphs
- That's fine... but does Apriori Principle hold? YES !!!!!!!!!!!!!!!!!
  If a graph is frequent, all of its subgraphs are frequent
- Support: number of graphs that contain a particular subgraph

Apriori-like Algorithm

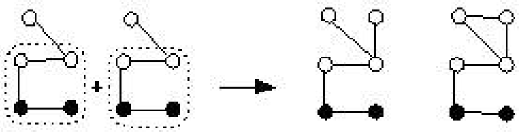
- Find frequent 1-subgraphs
- Repeat
  - Candidate generation
    - Use frequent (*k-1*)-subgraphs to generate candidate *k*-subgraph
  - Candidate pruning
    - Prune candidate subgraphs that contain infrequent (*k-1*)-subgraphs
  - Support counting
    - Count the support of each remaining candidate
  - Eliminate candidate *k*-subgraphs that are infrequent

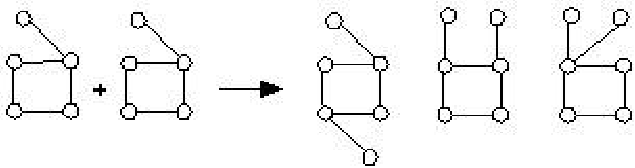
In practice, it is not as easy. There are many other issues

Candidate Generation

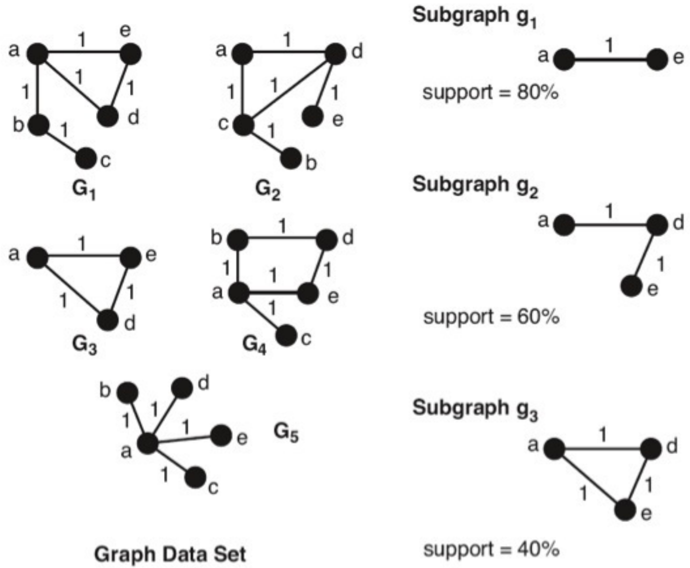
There are 2 ways to generate a candidate k-graph from two given $k-1$-subgraphs.

1. Vertex Growing: Iteratively expanding the graph by adding one vertex at a time
2. Edge Growing: Iteratively expanding the graph by adding one edge at a time

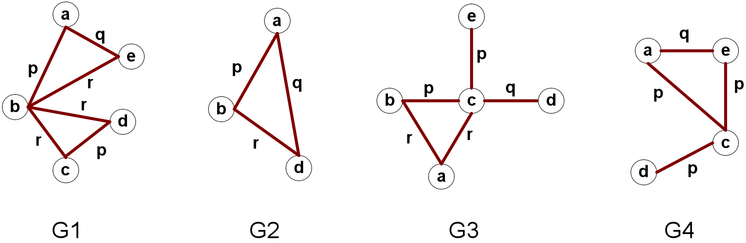




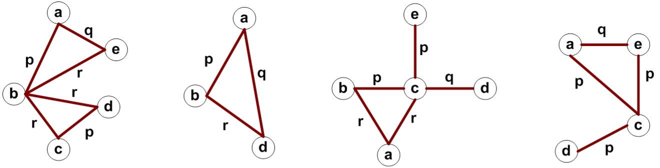
Support Counting



Example: Dataset



	(a,b,p)	(a,b,q)	(a,b,r)	(b,c,p)	(b,c,q)	(b,c,r)	…	(d,e,r)
G1	1	0	0	0	0	1	…	0
G2	1	0	0	0	0	0	…	0
G3	0	0	1	1	0	0	…	0
G4	0	0	0	0	0	0	…	0

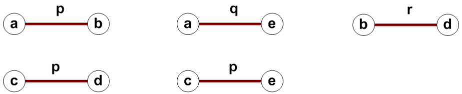


G1 G2 G3 G4

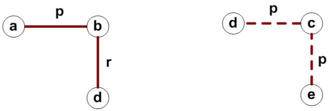
Minimum support count = 2

k=1
Frequent
Subgraphs

k=2
Frequent
Subgraphs



k=3
Candidate
Subgraphs



(Pruned candidate)

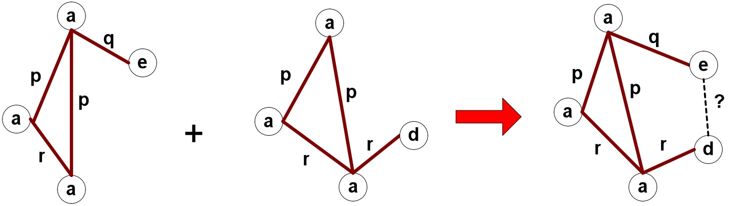
Issues

Apriori-based algorithms have two kinds of considerable overheads:

1. Joining two size-k frequent graphs to generate size-$(k + 1)$ graph candidates - This produces an exponential number of candidates.
2. Checking the frequency of these candidates separately - Some graphs are isomorphic

These overheads constitute the performance bottleneck of Apriori-based algorithms.

Multiplicity of Candidates (Vertex Growing)

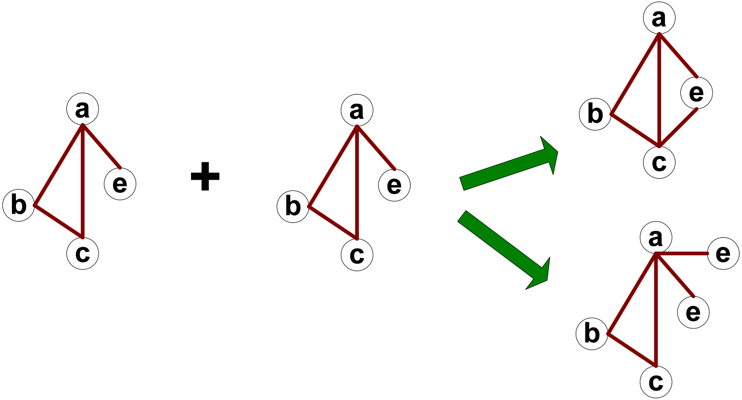


G1 G2 G3 = join(G1,G2)

$$M_{G1} = \begin{pmatrix} 0 & p & p & q \\ p & 0 & r & 0 \\ p & r & 0 & 0 \\ q & 0 & 0 & 0 \end{pmatrix} \qquad M_{G2} = \begin{pmatrix} 0 & p & p & 0 \\ p & 0 & r & 0 \\ p & r & 0 & r \\ 0 & 0 & r & 0 \end{pmatrix} \qquad M_{G3} = \begin{pmatrix} 0 & p & p & 0 & q \\ p & 0 & r & 0 & 0 \\ p & r & 0 & r & 0 \\ 0 & 0 & r & 0 & ? \\ q & 0 & 0 & ? & 0 \end{pmatrix}$$

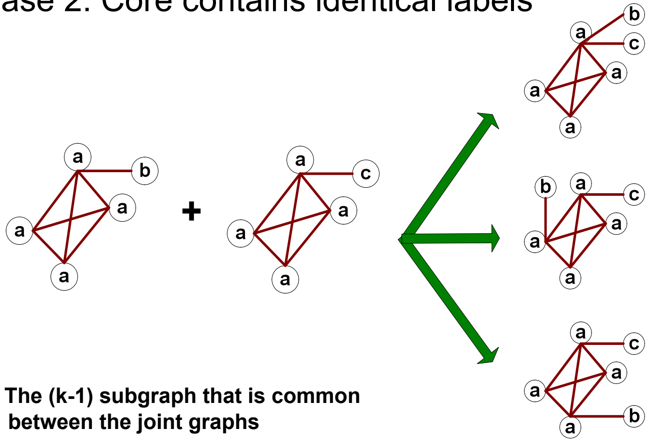
Multiplicity of Candidates (Edge growing)

- Case 1: identical vertex labels



Multiplicity of Candidates (Edge growing)

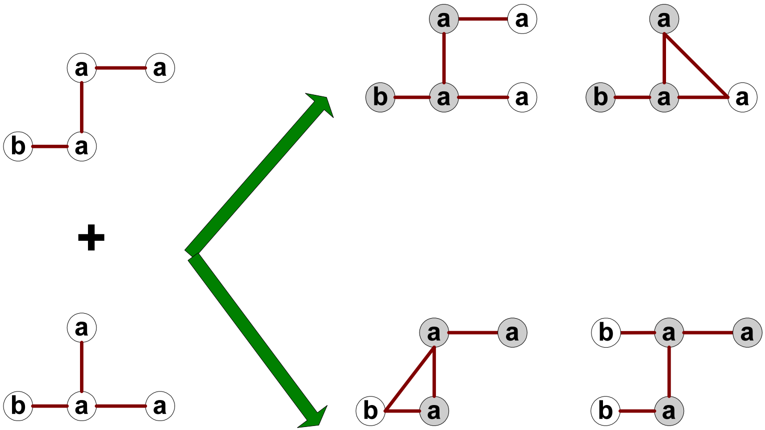
- Case 2: Core contains identical labels



Core: The (k-1) subgraph that is common between the joint graphs

Multiplicity of Candidates (Edge growing)

- Case 3: Core multiplicity

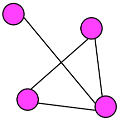


Graph Isomorphism

- Test for graph isomorphism is needed:
  - During candidate generation step, to determine whether a candidate has been generated
  - During candidate pruning step, to check whether its (*k-1*)-subgraphs are frequent
  - During candidate counting, to check whether a candidate is contained within another graph

Graph Isomorphism

- Use canonical labeling to handle isomorphism
  - Map each graph into an ordered string representation (known as its code) such that two isomorphic graphs will be mapped to the same canonical encoding
  - Example:
    - Lexicographically largest adjacency matrix



$$\longrightarrow \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

String: 0010001111010110 **Canonical: 0111101011001000**