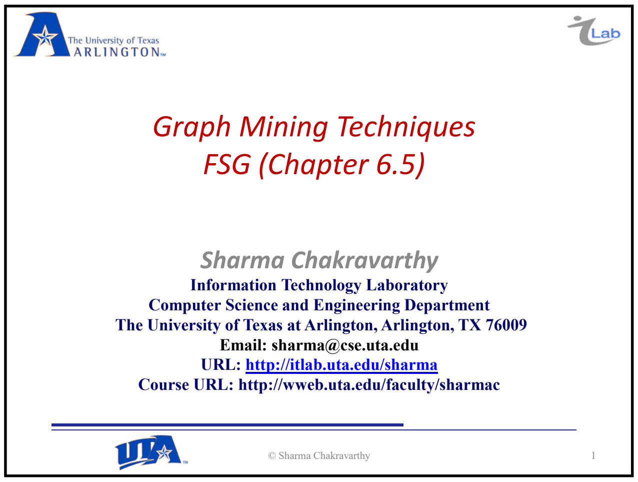
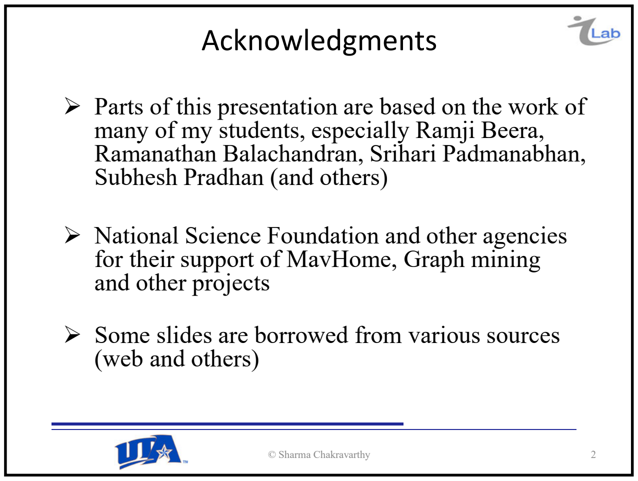


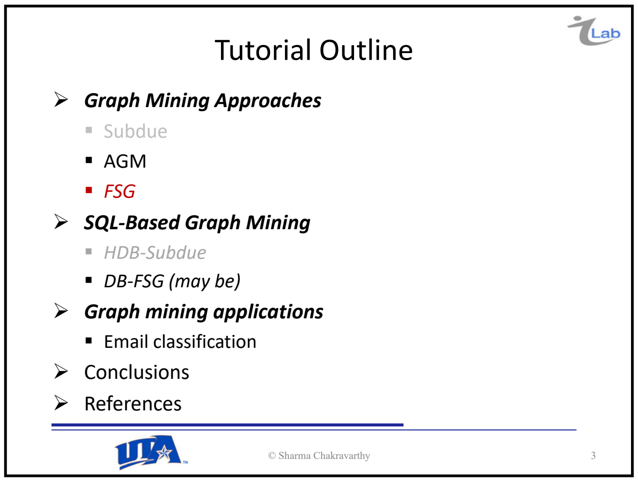
# Graph Mining Techniques
# FSG (Chapter 6.5)

*Sharma Chakravarthy*
**Information Technology Laboratory**
**Computer Science and Engineering Department**
**The University of Texas at Arlington, Arlington, TX 76009**
**Email: sharma@cse.uta.edu**
**URL: http://itlab.uta.edu/sharma**
**Course URL: http://wweb.uta.edu/faculty/sharmac**

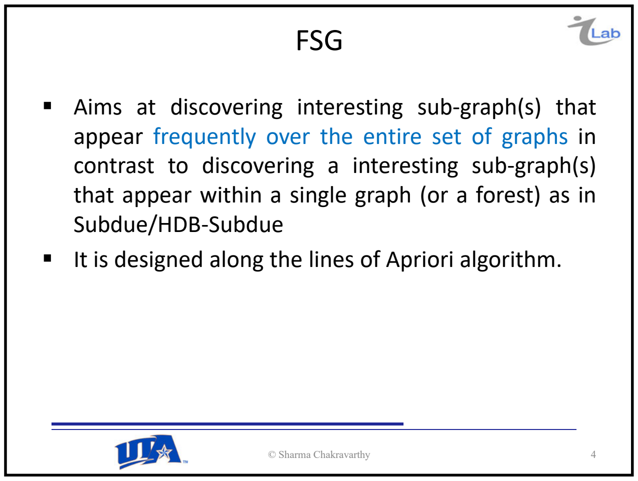


## Acknowledgments

- Parts of this presentation are based on the work of many of my students, especially Ramji Beera, Ramanathan Balachandran, Srihari Padmanabhan, Subhesh Pradhan (and others)
- National Science Foundation and other agencies for their support of MavHome, Graph mining and other projects
- Some slides are borrowed from various sources (web and others)



## Tutorial Outline



## FSG

- Aims at discovering interesting sub-graph(s) that appear frequently over the entire set of graphs in contrast to discovering a interesting sub-graph(s) that appear within a single graph (or a forest) as in Subdue/HDB-Subdue
- It is designed along the lines of Apriori algorithm.

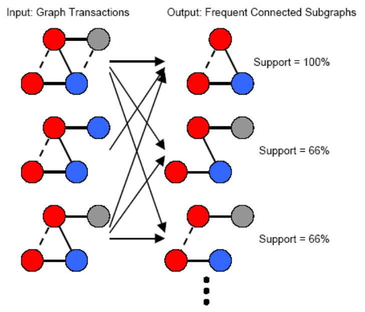
## Problem Definition

- discovering all connected subgraphs that occur frequently over the entire set of graphs.
  - Subdue: best n are output (n is user defined)
- vertex : corresponds to an entity
- edge : correspond to a relation between two entities

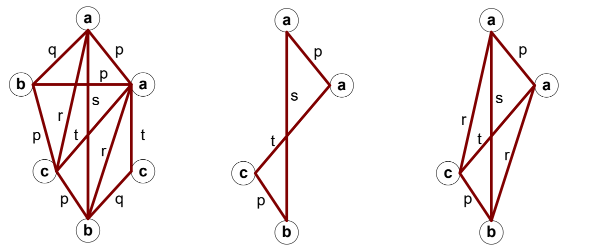
## Definitions

- Gs will be an induced subgraph of G if Vs is a subset of V and Es contains all the edges of E that connect vertices in Vs.
- Two graphs *G*1 = (*V*1;*E*1) and *G*2 = (*V*2;*E*2) are *isomorphic* if they are topologically identical to each other, that is, there is a mapping from *V*1 to *V*2 such that each edge in *E*1 is mapped to a single edge in *E*2 and vice versa
- An automorphism : an isomorphism mapping where G1 = G2 (on the same graph).

## Example of Frequent sub-graph discovery


Input: Graph Transactions
Output: Frequent Connected Subgraphs
Support = 100%
Support = 66%
Support = 66%

## Graph Definitions

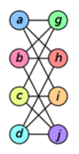
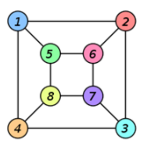


(a) Labeled Graph (b) Subgraph (c) Induced Subgraph

## Example (from wiki)

- The two graphs shown below are isomorphic, despite their different looking drawings

Two functions for mapping
Vertexes and edges are used.
Labels need to be mapped as well.

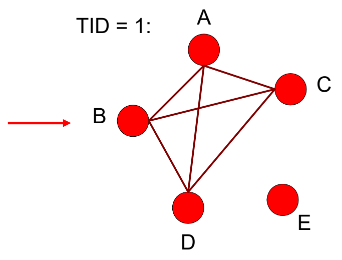
| Graph G | Graph H | An isomorphism between G and H |
|---|---|---|
|  |  | *f*(*a*) = 1<br>*f*(*b*) = 6<br>*f*(*c*) = 8<br>*f*(*d*) = 3<br>*f*(*g*) = 5<br>*f*(*h*) = 2<br>*f*(*i*) = 4<br>*f*(*j*) = 7 |

- The formal notion of "isomorphism", e.g., of "graph isomorphism", captures the informal notion that some objects have "the same structure" if one ignores individual distinctions of "atomic" components of objects in question

## Representing Transactions as Graphs

- Each transaction is a clique of items

| Transaction Id | Items |
|---|---|
| 1 | {A,B,C,D} |
| 2 | {A,B,E} |
| 3 | {B,C} |
| 4 | {A,B,D,E} |
| 5 | {B,C,D} |



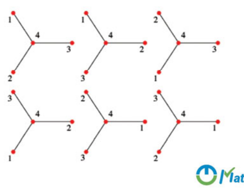
## Automorphism

- The star graph S_4 has six automorphisms: (1, 2, 3, 4), (1, 3, 2, 4), (2, 1, 3, 4), (2, 3, 1, 4), (3, 1, 2, 4), (3, 2, 1, 4), illustrated above.

an **automorphism** is an isomorphism from a mathematical object to itself

Multiple mappings exist unlike Itemset due to graph representation

**AUTOMORPHISM**



MathsOne

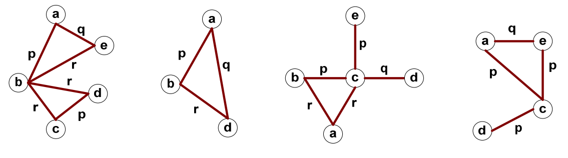
## Definitions

- The canonical label of a graph G = (V;E), cl(G) : unique code (e.g., string) that is invariant on the ordering of the vertices and edges in the graph.
- Two graphs will have the same canonical label if they are isomorphic
  - We will use this and discuss it later!
- Canonical labels are useful to (i) compare two graphs (ii) establish a complete ordering of a set of graphs in a unique and deterministic way, regardless of the original vertex and edge ordering.

## Challenges

- Node may contain duplicate labels
- Support and confidence
  - How to define them?
- Additional constraints imposed by pattern structure
  - Support and confidence are not the only constraints
  - Assumption: frequent subgraphs must be connected
- Apriori-like approach:
  - Use frequent k-subgraphs to generate frequent (k+1) subgraphs
    - ◆What is k?

## Example: Dataset



G1 G2 G3 G4

| | (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 |

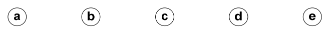
## Challenges…

- Support:
  - number of graphs that contain a particular subgraph
- Apriori principle still holds
  - But testing for apriori property is very expensive
- Level-wise (Apriori-like) approach:
  - Vertex growing: used by AGM
    - ◆ k is the number of vertices
  - Edge growing: used by FSG
    - ◆ k is the number of edges

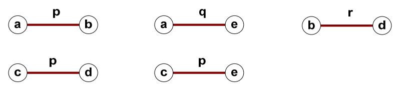
## Example

Minimum support count = 2

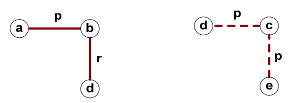
k=1 Frequent Subgraphs



k=2 Frequent Subgraphs



k=3 Candidate Subgraphs



(Pruned candidate)

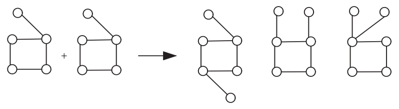
## FSG is more challenging

- Complexity
  - Finding support is difficult on graphs as compared to itemsets (cannot do group by)
    - Need subgraph isomorphism
  - Number of candidate subgraphs is far larger (than in market basket analysis)
    - An itemset can be arranged in more ways as a graph
    - An edge label may appear multiple times in a graph
- Apriori still holds for subgraphs
  - A k-graph is frequent only if all of its (k-1)-subgraphs are frequent

## Algorithm fsg(*D; t*)

```
1: F(1) = detect all frequent 1-subgraphs in D
2: F(2) = detect all frequent 2-subgraphs in D
3: k = 3
4: while F (k-1) != NULL ; do
5:    C(k) = fsg-gen(F(k-1))
6:    for each candidate G(k) in C(k) do
7:       G(k).count = 0
8:       for each transaction T in D do
9:          if candidate G(k) is included in transaction T then
10:            G(k).count++
11: F(k) = {G(k) in C(k) | G(k).count >= t|D| }
12: k++
13: return F(1);F(2); ……… ;F(k-2)
```

## Apriori-Based, Breadth-First Search



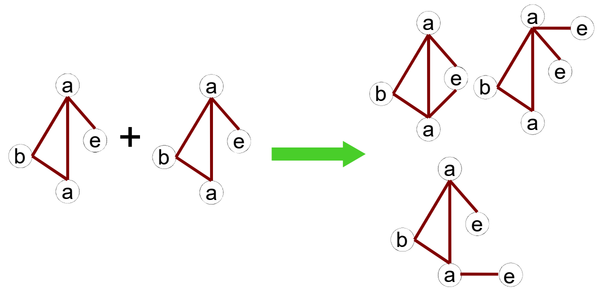
- FSG (Kuramochi and Karypis ICDM'01)
  - generates new graphs with one more edge
- In contrast, when you join two itemsets, you get **an** itemset!

## FSG Algorithm

- Two graphs P and Q are ***joinable*** if the join of the two graphs produces a non-empty set
- Two graphs P and Q are joinable if P ∩ Q is a graph with size |P| -1 or share a common "core" with size P-1

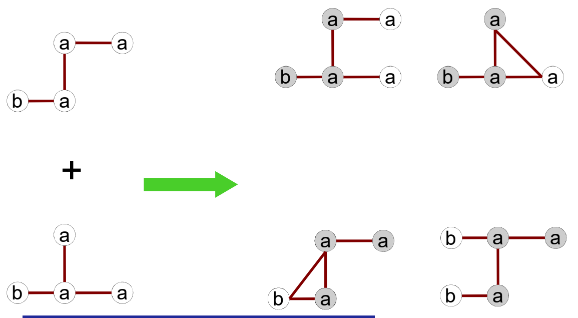
## Multiplicity of Candidates

➢Case 1: identical vertex labels



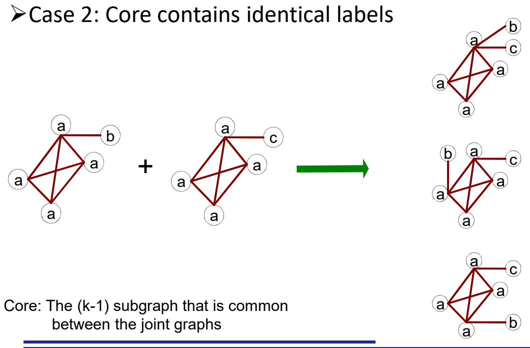
## Multiplicity of Candidates

➢Case 3: Core multiplicity



## Multiplicity of Candidates

➢Case 2: Core contains identical labels



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

## Key Features Of FSG

- Uses sparse graph representation that minimizes storage and computation
- Increases the size of frequent subgraphs by adding one edge at a time (apriori)
- Uses canonical labeling to uniquely identify subgraphs
- ONLY undirected edges; it cannot handle multiple edges and cycles
  - (Unlike subdue)

## FSG Components

- Candidate Generation
- Graph Isomorphism
- **Interestingness metric**

  Frequency is considered to be an interestingness metric. That is, the frequent sub-graph that appears in most graph databases is considered interesting

## Key Aspects

- interested in subgraphs that are connected
- allow the graphs to be labeled
- both vertices and edges may have labels associated with them which are not required to be unique.

## Graph Isomorphism

- FSG uses canonical labeling for isomorphism.
- Canonical labeling assigns a unique code for each substructure and two substructures have the same canonical code only if the substructures are isomorphic.
- Canonical labeling is an easier and faster way of finding the isomorphic substructures, but it suffers from the fact that canonical labeling cannot be used for graphs that have multiple edges between the vertices and cycles

## FSG

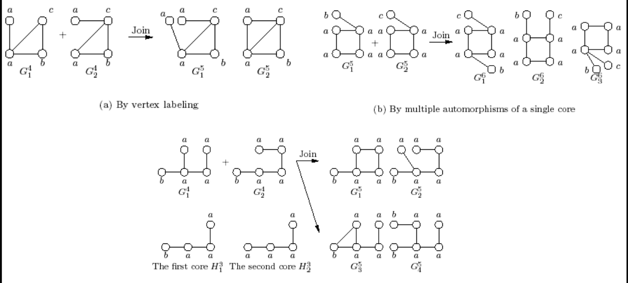
- Input to FSG
  - Set of graphs (transactions)
  - Labeled edges and vertices
  - Edges are undirected
  - No inexact match

  - Subdue can take a single connected graph or a forest of graphs
  - Edges can be directed or undirected
  - Both edges and vertices can have labels
  - Multiple edges between nodes is supported
  - Cycles are supported

## Algorithm fsg(*D; t*)

```
1: F(1) = detect all frequent 1-subgraphs in D
2: F(2) = detect all frequent 2-subgraphs in D
3: k = 3
4: while F (k-1) != NULL ; do
5:    C(k) = fsg-gen(F(k-1))
6:    for each candidate G(k) in C(k) do
7:       G(k).count = 0
8:       for each transaction T in D do
9:          if candidate G(k) is included in transaction T then
10:            G(k).count++
11: F(k) = {G(k) in C(k) | G(k).count >= t|D| }
12: k++
13: return F(1);F(2); ……… ;F(k-2)
```

## Joining of two k-subgraphs



(a) By vertex labeling (b) By multiple automorphisms of a single core (c) By multiple cores

## Candidate generation

- Candidates are the substructures which would be searched and counted in the given graph databases
- create a set of candidates of size *k*+1, given frequent *k*-subgraphs.
- by joining two frequent *k*-subgraphs (using downward closure property)
- must contain the same (*k*-1)-subgraph (common *core*)
- Self-join required for unlabeled graphs

## Join

**Algorithm 3** fsg-join($G_1^k, G_2^k, H^{k-1}$) (Join)

```
1: $e_1 \leftarrow$ the edge appears only in $G_1^k$, not in $H^{k-1}$
2: $e_2 \leftarrow$ the edge appears only in $G_2^k$, not in $H^{k-1}$
3: $M \leftarrow$ generate all automorphisms of $H^{k-1}$
4: $B^{k+1} = \emptyset$
5: for each automorphism $\phi \in M$ do
6:    $B^{k+1} \leftarrow B^{k+1} \cup$ {all possible candidates of size $k+1$ created from a set of $e_1$, $e_2$, $H^{k-1}$ and $\phi$}
7: return $B^{k+1}$
```

## Key computational steps in candidate generation

- core identification
- Joining
- using the downward closure property

## Core Identification

- for each frequent *k*-subgraph, store the canonical labels of its frequent (*k*-1)-subgraphs
- Cores are the intersection of these lists.
- complexity : quadratic on |F(k)|

- *inverted indexing scheme*
- for each frequent (k-1)subgraph, maintain a list of child k-subgraphs.
- form every possible pair from the child list of every (k-1) frequent subgraph.
- complexity of finding an appropriate pair of subgraphs: square of the number of child k-subgraphs (which is much smaller)

## Speeding automorphism computation

- cache previous automorphisms associated with each core
- look them up instead of performing the same automorphism computation again.
- saved list of automorphisms is discarded once *Ck*+1 has been generated.

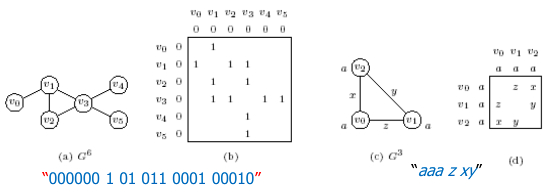
## Downward Closure

- uses canonical labeling to substantially reduce the complexity of the checking whether or not a candidate pattern satisfies the downward closure property of the support condition

## Canonical labels

- Canonical labels are computed for subgraphs
  - Complexity O(|V|!)
- These labels are used for subgraph comparison (instead of isomorphism)
- A number of optimizations are proposed to reduce the complexity from O(|V|!)
- But once computed, they can be cached and used quickly for comparison

## Canonical Labeling



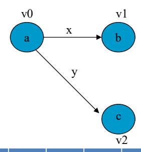
"000000 1 01 011 0001 00010"

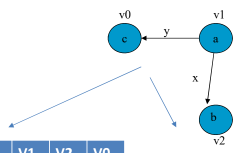
"*aaa z xy*"

- Different orderings of the vertices will give rise to different codes
- Try every possible permutation of the vertices and choose the ordering which
- gives lexicographically the largest, or the smallest code.
- O(|V|!)

- For the above, the canonical orderings are 000000111110010000100 and aaazyx which needs different ordering of vertices!
  - (v3 v1 v2 v4 v5 v0) and (v1 v0 v2) respectively!

## Canonical labeling





| | | V0 | V1 | V2 |
|---|---|---|---|---|
| | | a | b | c |
| V0 | a | | x | y |
| V1 | b | | | |
| c2 | c | | | |

abc xy

| | | V1 | V2 | V0 |
|---|---|---|---|---|
| | | a | b | c |
| V1 | a | | x | y |
| V2 | b | | | |
| v0 | c | | | |

abc xy

| | | V0 | V1 | V2 |
|---|---|---|---|---|
| | | c | a | b |
| V0 | c | | | |
| V1 | a | y | | x |
| v2 | b | | | |

cab yx

## Why using canonical labeling is important?

- use the canonical label repeatedly for comparison without recalculation.
- by regarding canonical labels as strings, we get the total order of graphs.
- sort them in an array
- index by binary search efficiently.

## Canonical label optimizations

- Vertex invariants – do not change across isomorphism mappings (e.g., degree or label of a vertex)
- Do not asymptotically change the computational complexity; in practicce it is useful

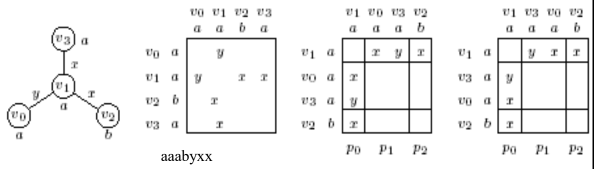
## Vertex Invariants

- attributes or properties assigned to a vertex which do not change across isomorphism mappings.
- partition the vertices into equivalence classes such that all the vertices assigned to the same partition have the same values for the vertex invariants.
- only maximize over those permutations that keep the vertices in each partition together.

## Invariants

- degrees and labels of a vertex
  - Partition into disjoint groups
    - Each partition has vertices with the same label and degree
- the labels and degrees of their adjacent vertices (neighbor lists)
- information about their adjacent partitions

## Vertex degree and label as invariant



aaabyxx

Find canonical label of each partition based on an invariant, compute its canonical label and concatenate it.

Instead of 4! Or 24 label computations for this, only 2 are needed!

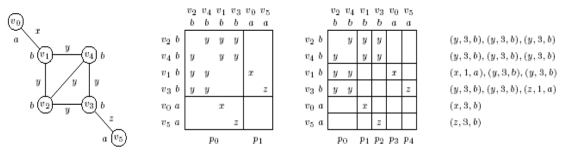
only 1! * 2!= 2 permutations although the total permutations 4! = 24.

aaabyxx

## Invariants

- degrees and labels of a vertex
  - Partition into disjoint groups
    - Each partition has vertices with the same label and degree
- the labels and degrees of their adjacent vertices (neighbor lists)
- information about their adjacent partitions

## Neighbor lists



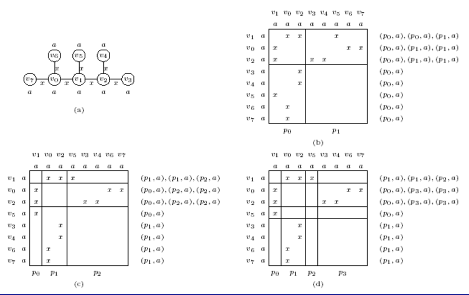
- ($l(e)$; $d(v)$; $l(v)$)
  - $l(e)$ is the label of the incident edge $e$,
  - $d(v)$ is degree of the adjacent vertex $v$, and
  - $l(v)$ is its vertex label.
- same partition if and only if nl($u$) = nl($v$)
- reduce from 4! x 2! to 2! (1! * 2! * 1! * 1! * 1!).

## Invariants

- degrees and labels of a vertex
  - Partition into disjoint groups
    - Each partition has vertices with the same label and degree
- the labels and degrees of their adjacent vertices (neighbor lists)
- Iterative partitioning (generalization of neighbor lists)
  - Use p(v) and l(e)

## Iterative Partitioning

p(v): identifier of a partition
l(e): label of the incident edge to the neighbor vertex v



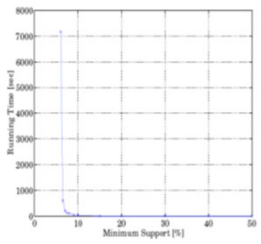
## Frequency Counting

- for each frequent subgraph, keep a list of transaction identifiers that support it.
- to compute the frequency of G(k+1), first compute the intersection of the TID lists of its frequent k-subgraphs.
- If the size of the intersection is below the support, G(k+1) is pruned - subgraph isomorphism computations avoided
- Otherwise use subgraph isomorphism on the set of transactions in the intersection of the TID lists.

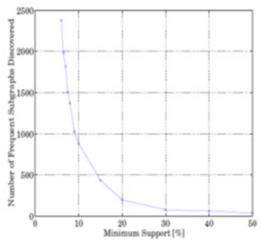
## Apriori-based method

- Experiment Result

-Chemical Compound Dataset, which contains 340 compounds,24 different atoms (vertices)

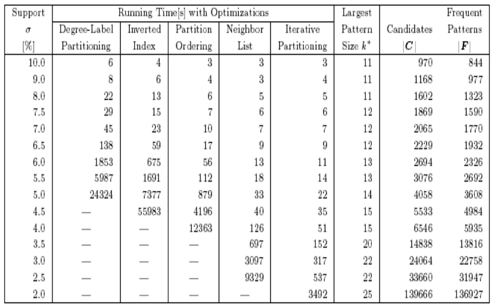


(a) Minimum support σ and running time



(b) Minimum support σ and the number of discovered frequent subgraphs

## Chemical compound data set

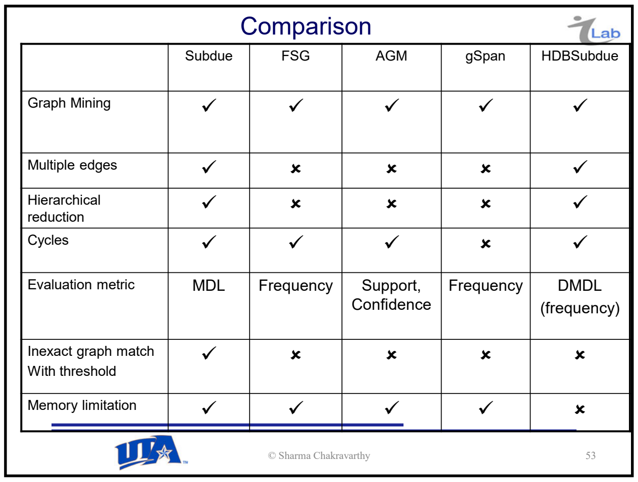


| Support | Running Time[s] with Optimizations | | | | | Largest | | Frequent |
|---|---|---|---|---|---|---|---|---|
| σ | Degree-Label | Inverted | Partition | Neighbor | Iterative | Pattern | Candidates | Patterns |
| [%] | Partitioning | Index | Ordering | List | Partitioning | Size $k^*$ | $\|C\|$ | $\|F\|$ |
| 10.0 | 6 | 4 | 3 | 3 | 3 | 11 | 970 | 844 |
| 9.0 | 8 | 6 | 4 | 3 | 4 | 11 | 1168 | 977 |
| 8.0 | 22 | 13 | 6 | 5 | 5 | 11 | 1602 | 1323 |
| 7.5 | 29 | 15 | 7 | 6 | 6 | 12 | 1869 | 1590 |
| 7.0 | 45 | 23 | 10 | 7 | 7 | 12 | 2065 | 1770 |
| 6.5 | 138 | 59 | 17 | 9 | 9 | 12 | 2229 | 1932 |
| 6.0 | 1853 | 675 | 56 | 13 | 11 | 13 | 2694 | 2326 |
| 5.5 | 5987 | 1691 | 112 | 18 | 14 | 13 | 3076 | 2692 |
| 5.0 | 24324 | 7377 | 879 | 33 | 22 | 14 | 4058 | 3608 |
| 4.5 | — | 55983 | 4196 | 40 | 35 | 15 | 5533 | 4984 |
| 4.0 | — | — | 12363 | 126 | 51 | 15 | 6546 | 5935 |
| 3.5 | — | — | — | 697 | 152 | 20 | 14838 | 13816 |
| 3.0 | — | — | — | 3097 | 317 | 22 | 24064 | 22758 |
| 2.5 | — | — | — | 9329 | 537 | 22 | 33660 | 31947 |
| 2.0 | — | — | — | — | 3492 | 25 | 139666 | 136927 |

340 compounds, 24 unique element names, 66 element types, 4 bonds
Avg Tx size is 27.4 edges, 27 vertices; largest 214V214E, 2GB ram
Amount of time taken by FSG to find all frequently occurring subgraphs

## Conclusions

- Graph mining is a powerful approach needed by many real-world applications
- There is need for both Subdue class of mining algorithms and frequent subgraph class of algorithms
- Scalability is an extremely important issue
- Our approach to using SQL has yielded very promising scalability results (800K vertices and 1600K edges)



## Comparison

| | Subdue | FSG | AGM | gSpan | HDBSubdue |
|---|---|---|---|---|---|
| Graph Mining | ✓ | ✓ | ✓ | ✓ | ✓ |
| Multiple edges | ✓ | ✗ | ✗ | ✗ | ✓ |
| Hierarchical reduction | ✓ | ✗ | ✗ | ✗ | ✓ |
| Cycles | ✓ | ✓ | ✓ | ✗ | ✓ |
| Evaluation metric | MDL | Frequency | Support, Confidence | Frequency | DMDL (frequency) |
| Inexact graph match With threshold | ✓ | ✗ | ✗ | ✗ | ✗ |
| Memory limitation | ✓ | ✓ | ✓ | ✓ | ✗ |

© Sharma Chakravarthy

53