Graph Mining Techniques and Their Applications

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

Tutorial Outline

- Data Mining Overview
- Need for Graph Mining
  - Applications
- Graph Mining Approaches
  - **Subdue, AGM**
  - ***FSG, gSpan***
- ***SQL-Based Graph Mining***
  - ***HDB-Subdue***
  - ***DB-FSG***
- Conclusions
- References

Motivation

Fraud division, some large telephone company:

"How do we find these guys? There are 10 billion records on 10 million customers in the main database. With all this information we have about our customers and all the calls they make, can't you just ask the database to figure out which lines have been set-up *temporarily* and exhibited *similar* calling patterns in the *same time periods*? The information is in there, I just know it …"

Problem

- "Find-similar" problem just described is hard
  - e.g., "What products need to be improved?"
  - e.g., "Which books won't be checked out and can be taken off the shelves?"

Why?

- Massive amounts of data
  - More and more online data stores (e.g., Web, click streams, corporate databases, etc.)
- No easy way to describe what to look for
- Traditional, interactive approaches fail
  - Size of data, different purposes

Another Example

- Marketing cellular phones
  - Churn is too high
    - Turnover after the initial contract is too high
  - What is a good strategy
    - Giving new phone to everyone is too expensive (and wasteful)
    - Bringing back customers after they leave is very difficult

What to do

- A few months before the contract expires, if one can predict which customers are likely to quit,
  - Give incentive to those who are likely to quit
  - Don't do anything for those who are NOT likely to quit
- How do I predict future behavior???
  - Corporate Palm reading !
  - Human intuition !!
  - Data mining (DM) or knowledge discovery (KDD)

Data Mining

- *Data Mining* (DM) is part of the knowledge discovery process carried out to extract valid patterns and relationships in very large data sets
  - Usually don't know what to look for, like a "voyage into the unknown"
- Regarded as unsupervised learning from basic facts (axioms) and data
- Roots in AI and statistics
  - Uses techniques from machine learning, pattern recognition, statistics, database, visualization, etc.

Constituents of Data Mining?

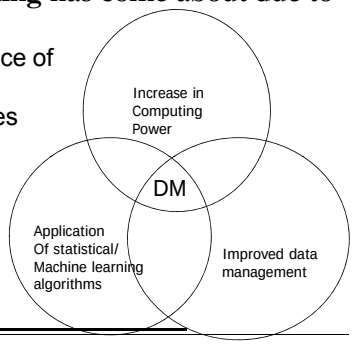
- There is an element of discovery.
- What is discovered may be counter-intuitive even to the expert.
- Exhaustive scan/processing of the available data (as against sampling)
- Verification of conjecture or hypothesis

Enablers

- Reduced cost of storage
- Reduced cost of processing
- Ability to store, process, and manage large volumes of data
- New techniques such as association rules, sequence data processing, text mining
- However,
  - Scalability, visualization of results, filtering very large outputs are new issues!

Data Mining has come about due to

- Convergence of multiple technologies



Drivers

- Today, business information (or BI) systems are as important to corporations as transaction processing systems were earlier
- Mass personalization and better utilization of data
- Identify new and profitable markets, and channels to enter them
- Increase customer loyalty, profitability, life-time value
- Decrease risk

AI and Statistics

- If DM is rooted in AI and statistics, what is new about it?
  - AI traditionally dealt with small samples
  - The emphasis was an learning, extrapolation, and generalization
  - The emphasis in DM is on processing the actual data, not just samples!
  - DM tries to leverage the collected, accumulated data, and derive tangible rules/conclusions (generalization is also possible)

Machine Learning

- Observation
- Analysis
- Theory/model
- Prediction

Either the predictions are correct in which case the theory is corroborated, or the predictions are wrong.

New theory or exceptions!

DM Vs. Machine learning

- ML methods form the core of DM
- Amount of data makes a difference
  - accessing examples can be a problem
  - missing values and incomplete data
- DM has more modest goals: automating the tedious discovery tasks

DM Vs. Statistics

- Similar goals; additional methods
- Amount of data
- DM as a preliminary stage for statistical analysis
- Challenge to DM: better ties with statistics

Data Mining is NOT …

- Data warehousing
- Ad hoc query/reporting
- Online Analytical Processing (OLAP)
- Data Visualization
- Agents/mediators,
- Pervasive computing, …

What DM will NOT do !

- Substitute for human intuition and discovery
- I don't think a DM system will (ever?) discover $e = mc^2$
- I don't think DM will (ever?) discover PV = RT
- I don't think DM will (ever?) discover gravity, Newton's law's of motion
- On the other hand, It may discover new black holes !

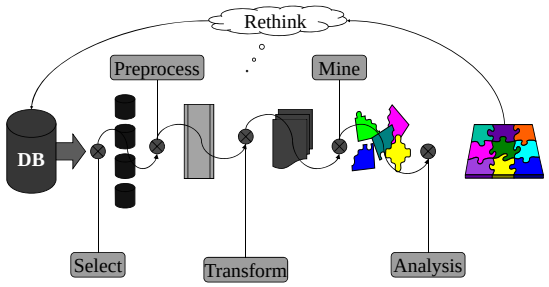
Applications

- Customer profiling
  - Find new customers,
- *Market basket analysis*
  - Manage inventory, transportation, display …
- Risk analysis
  - Insurance, loan, stock, …
- Text analysis
  - Library, Web, …
- Fraud detection
  - Financial transactions, social networks
- CRM, Scientific discovery, forecasting, …

DM Applications Vs. DM

- Problem, goal and task definition (10%)
- Data Warehousing: data collection and organization (50%)
- Data Mining: data analysis and knowledge discovery (30%)
- Decision support / optimization: assess pros and cons, take actions (10%)

Data Mining Cycle

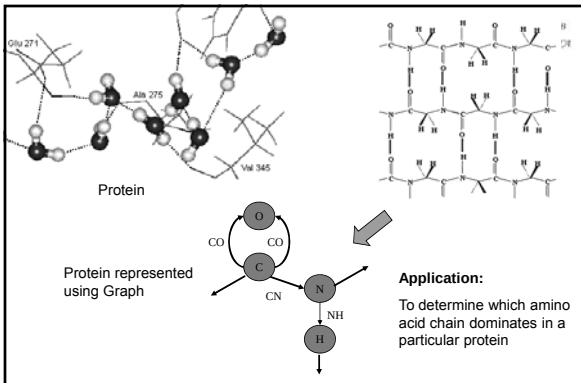


Common Pitfalls

- Misinterpretation of results
- Statistical significance
- Dirty data
- Too much information generated
- Legality
- Privacy/Ethics

Need for Graph Mining

- Association Rule Mining, decision trees… mine transactional data.
- Graph based mining techniques are used for mining data that are structural in nature (chemical compounds, complex proteins, VLSI circuits, social networks, …) as mapping them to other representations is not possible or will lead to loss of structural information
- Significant work in the area includes the Subdue substructure discovery algorithm (Cook & Holder), HDB-Subdue (chakrvarthy, Padmanabhan), the apriori graph mining (AGM) (Inokuchi, Washio, and Motoda), the frequent subgraph (FSG) technique (Karypis & Kuramochi), and the gSpan approach (J. Han) (also SPIN (Huan, Wang, Prins, and Yang))



Application Domains

- Chemical Reaction chains
- CAD Circuit Analysis
- Social Networks
- Credit Domains
- Web analysis
- Games (Chess, Tic Tac toe)
- Program Source Code analysis
- Chinese Character data bases
- Geology
- Aviation Data Bases

Graph Based Data Mining

- A Graph representation of the database is an intuitive and an obvious choice.
- Graphs can be used to accurately model and represent scientific data sets. Graphs are suitable for capturing arbitrary relations between the various objects.

Data Instance	Graph Instance
Object	Vertex
Object's Attributes	Vertex Label
Relation Between Two Objects	Edge
Type Of Relation	Edge Label

- Graph based data mining aims at discovering interesting and repetitive patterns within these structural representations of data.

Graph Mining Overview

- A substructure is a connected subgraph; need to differentiate between substructures and substructure instances
- A connected subgraph is a subgraph of the original graph where there is a path between any two vertices
- A subgraph $G_s = (V_s, E_s)$ of $G = (V, E)$ is induced if E_s contains all the edges of E that connect vertices in V_s
- Directed and undirected edges are needed; multiple edges between two nodes need to be accommodated; cycles need to be handled

Graph Mining: Complexity

- Enumerating all the substructures of a graph has exponential complexity
- Subgraph isomorphism is NP complete
- Generating canonical labels is O(|V|!), where V is the number of vertices
- All approaches have to deal with the above in order to be able to work on large data sets
- Different approaches do it differently; scalability depends on it and the use of buffers

Subdue

- One of the earliest work in Graph based data mining
  - Uses sparse adjacency matrix for graph representation
- Substructures are evaluated using a metric called Minimum Description Length principle based on adjacency matrices
- Capable of matching two graphs, differing by the number of vertices specified by the threshold parameter, inexactly
- Performs hierarchical clustering by compressing the input graph with best substructure in each iteration

Subdue

- Capable of supervised discovery using positive and negative examples
- Available main memory limits the largest dataset that can be handled
- An SQL-based subdue addresses scalability
- A computationally constrained beam-search is used for subgraph generation
- A branch and bound algorithm is used for inexact match

AGM

- First to propose apriori-type algorithm for graph mining
- Detects frequent induced subgraphs for a given support
- Follows apriori algorithm
- Not much optimization; hence performance is not that good and is not scalable!

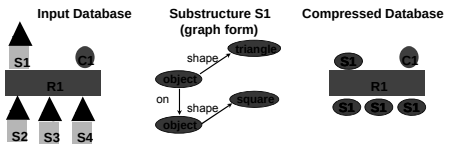
FSG

- FSG is used for frequent subgraph discovery
- Given a graph dataset $G = \{G1, G2, G3, \cdots\}$, it discovers all connected subgraphs that are found in at least the support threshold percent of the input graphs
- Uses a (sparse) adjacency matrix for graph representation
- A canonical label is generated by flattening the adjacency matrix of a graph (optimization)
- At each iteration FSG generates candidate subgraphs by adding one edge to the previous iteration's frequent subgraph (optimization)
- Graph isomorphism is checked by comparing canonical labels (optimization)

gSpan

- Avoids candidate generation
- Builds a new lexicographical ordering among graphs and maps each graph to a unique minimum DFS code as its canonical label
- Seems to outperform FSG
- Amenable to parallelization
- Does not handle cycles and multiple edges

Subdue Example



Subdue Substructure Discovery System

- Subdue Substructure discovery system is a graph-based data mining system that discovers interesting and repetitive patterns within graph representations of data.
- It accepts as input a forest and identifies the substructure that best compresses the input graph using the minimum description length (MDL) principle.
- It is capable of identifying both exact and inexact (isomorphic) substructures within a graph
- It uses a branch and bound algorithm for inexact matches (substructures that vary slightly in their edge and vertex descriptions).

Subdue

- Unsupervised learning
  - Subdue finds the most prevalent substructure from a set of unclassified input graphs
- Supervised learning
  - Subdue finds discriminating patterns from a set of classified (positive – G+ and negative – G- graphs)
- Hierarchical conceptual clustering
  - Compresses G with S and iterate
- Incremental Subdue
  - Uses unsupervised learning

Subdue

- Inferring graph grammars and graph primitives from examples
- Applications
  - Data mining
  - Pattern recognition
  - Machine learning

Graph Representation

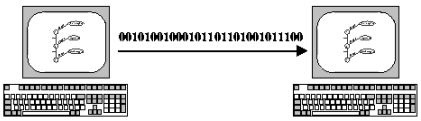
- Subdue represents data as labeled graph.
  - Vertices represent objects or attributes
  - Edges represent relationships between objects
  - Input: Labeled graph
  - Output: Discovered patterns and instances and their compression.
- A substructure is a connected subgraph
- Graph isomorphism is used to identify similar substructures

MDL Principle

- Theory to minimize description length (DL) of data
- information theoretic approach
- Has been shown to be good across domains
- Evaluates substructures based on their ability to compress DL of graph
- Description length =DL(S) + DL(G/S)
  - Depends upon the representation
  - Substructure that best compresses the original is chosen

MDL Principle

- Best theory: minimizes description length of data
- Evaluate substructure based ability to compress DL of graph
- Description length = DL(S) + DL(G|S)



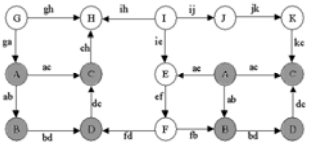
MDL Principle (cont.)

- Minimizes description length (DL) of data
- Substructures are evaluated based on their ability to compress the DL of the entire graph
- MDL = description length of the compressed graph / description length of the original graph

$$\text{MDL} = \frac{\text{DL(G)}}{\text{DL(S)} + \text{DL(G}\,|\,\text{S)}}$$

- DL(G) – Description length of the input graph
- DL(S) – Description length of sub graph
- DL(G|S) – Description length of the graph given the sub graph

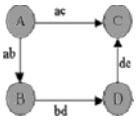
Example: Subdue



Input

- The input is a file, with all the vertex labels, vertex numbers, edges (using vertex numbers) and the edge directions

```
v 1 A
v 2 B
v 3 C
v 4 D
d 1 2 ab
d 1 3 ac
d 2 4 bd
d 4 3 dc
```



- 'd' stands for a directed edge and 'u' stands for undirected. 'e' stands for directed unless specified as –undirected at the command prompt.

Subdue Approach

- Create a substructure for each unique vertex label
- Expand each substructure by adding an edge (and may be a vertex)
- Maintain **beam** number of substructures for expansion
- Halting conditions
  - Discovered substructures > **limit**
  - List maintaining the substructures to be expanded becomes empty
  - **Max size** of substructure to be discovered is reached

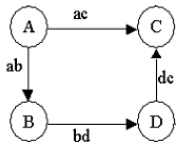
Output

- Output

  Substructure: MDL value = 1.21789, instances = 2

  Graph (4v,4e):

  v 1 A
  v 2 C
  v 3 B
  v 4 D
  d 1 2 ac
  d 1 3 ab
  d 4 2 dc
  d 3 4 bd



Subdue Parameters

- *Threshold* determines the amount of variation permissible in the vertex and edge descriptions during inexact graph match.
- *Nsubs* determines the maximum number of substructures that are returned as the set of best substructures
- *Beam* determines the maximum number of substructures that are retained for expansion in the next iteration of the discovery algorithm
- Minsize constrains the size of substructures returned as best to be equal to or more than the specified parameter value

Algorithm details contd.

```
SUBDUE( Graph, BeamWidth, MaxBest, MaxSubSize, Limit )
    ParentList = {}
    ChildList = {}
    BestList = {}
    ProcessedSubs = 0
    Create a substructure from each unique vertex label and its single-vertex
        instance; insert the resulting substructures in ParentList
    while ProcessedSubs <= Limit and ParentList is not empty do
        while ParentList is not empty do
            Parent = RemoveHead( ParentList)
            Extend each instance of Parent in all possible ways
            Group the extended instances into Child substructures
            foreach Child do
                if SizeOf( Child ) <= MaxSubSize then
                    Evaluate the Child
                    Insert Child in ChildList in order by value
                    if Length( ChildList ) > BeamWidth then
                        Destroy the substructure at the end of ChildList
            ProcessedSubs = ProcessedSubs + 1
            Insert Parent in BestList in order by value
```

Algorithm details contd.

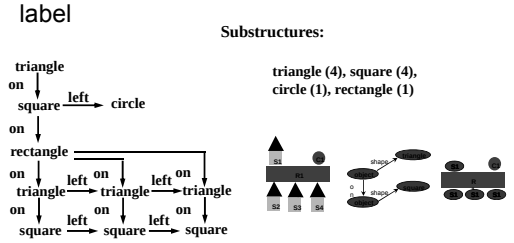
```
if Length( BestList ) > MaxBest then
        Destroy the substructure at the end of BestList
Switch ParentList and ChildList
return BestList
```


Algorithm

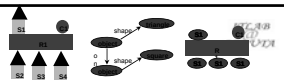
- SUBDUE(G, limit, beam)
  - S = vertices(G)
  - While (computation < limit) and (S <> {})
    - Order S from best to worst using MDL and background knowledge rules
    - S = first beam structures of S
    - b = first(S)
    - E = (b extended by one edge in all possible ways}
    - S = S U E
  - Return the substructure that produces the best compression ratio

Algorithm (Contd.)

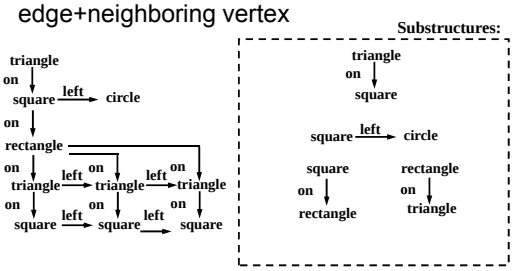
1. Create substructure for each unique vertex label



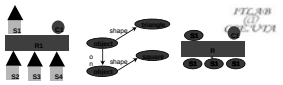
Algorithm (Contd.)



2. Expand best substructure by an edge or edge+neighboring vertex



Algorithm (cont.)



3. Keep only best substructures on queue (specified by beam width)
4. Terminate when queue is empty or #discovered substructures >= limit
5. Compress graph and repeat to generate hierarchical description

- Constrained to run in polynomial time

Graph Match

- Exact Graph match
- Inexact Graph match

Exact graph match is likely to be restrictive for real life applications.

Inexact Graph Match

- Some variations may occur between instances
- Want to abstract over minor differences
- Difference = cost of transforming one graph to make it isomorphic to another
- Match if cost/size < threshold

Inexact Graph Match

- Minimum graph edit distance

cumulative cost of graph changes required to transform the first graph into a graph isomorphic to the second graph.

- Uniform Cost Search

- Exact graph match is NP complete
- Bunke and Allerman's approach
  Each distortion is assigned a cost.
  A distortion is a basic transformation such as deletion, insertion of vertices and edges (and their labels)
  Fuzzy graph match is a mapping f: $N_1 \rightarrow N_2 \cup \{\lambda\}$, N_1 and N_2 are sets of nodes of graph 1 and graph 2. A node $v \in N_1$ that is mapped to λ is deleted
- If matchcost <= threshold then two graphs are said to be isomorphic
- Employing computational constraints such as bound on the number of substructures considered makes subdue run in polynomial time

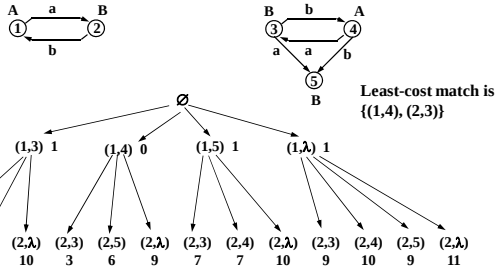
Graph 1

Graph 2

Mapping : 1 → 1, 2 → 2, 3 → 3, 4 → null
1) Delete Edge 3 → 4 (cost = 2) edge + label
2) Delete vertex 4 (cost = 2) node + label
3) Substitute vertex label C by D for vertex 3 (cost = 1)

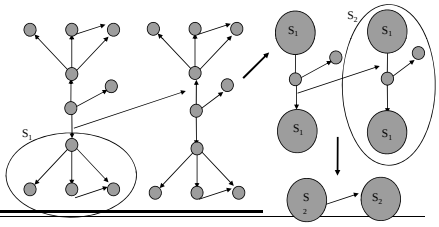
Total cost = 5 for this mapping

Inexact Graph Match

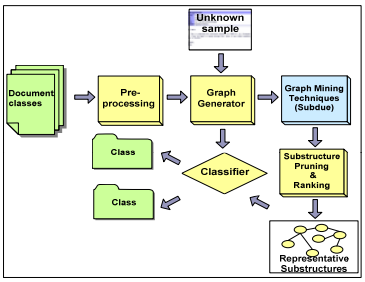


Hierarchical Reduction

- Input is a labeled graph
- A substructure is connected subgraph
- A substructure instance is a subgraph isomorphic to substructure definition
- Multiple iterations can create hierarchy



Document Classification Example



Control flow in the InfoSift classification system

Variants of Subdue

- Concept learner using positive and negative examples
- Hierarchical reduction
- Similarity detection in social networks
- Database approach to some of the above

AGM

- Apriori-based Graph Mining
  - Combines adjacency matrix with the efficient level-wise search of the frequent canonical matrix code
  - Adjacency matrix elements are numbers (e.g., edge numbers) instead of being binary
  - Support and confidence are redefined for the graph domain

AGM (Contd.)

- The subset property is preserved by using normal forms of adjacency matrix
- If the adjacency matrix generated is not in the normal form, it has to be transformed to the normal form.
- Support counting is done on the database as in an apriori algorithm.
- An efficient indexing is used for this purpose

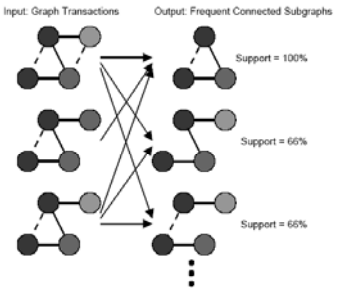
FSG

- 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

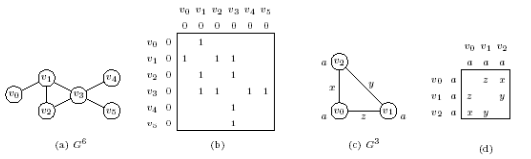
Example of Frequent sub-graph discovery



Key Features Of FSG

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

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|!)

Key Features Of FSG

- Introduces various optimizations for candidate generation and frequency counting
  - (Subdue has pruning, search space minimum detection etc.)

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

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.

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.

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

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.
- 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.

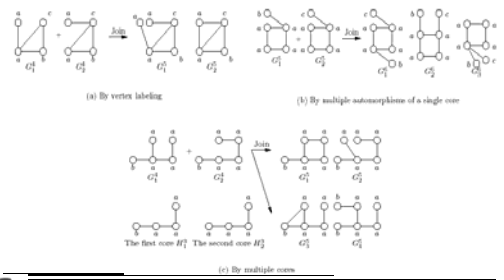
FSG

- Frequent subgraphs are found based on the set covering approach (frequency)
  - In Subdue subgraphs are found based on MDL (the graph that minimizes the description length of the input)
- User defined support threshold – minimum percentage of graphs in which a subgraph has to be found

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
- Subdue extends subgraph in every possible way via an edge and a vertex

Joining of two k-subgraphs



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.

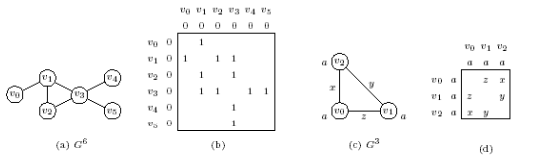
Downward Closure

- 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
- 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|!)

Why canonical labeling?

- use the canonical label repeatedly for comparison without the 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

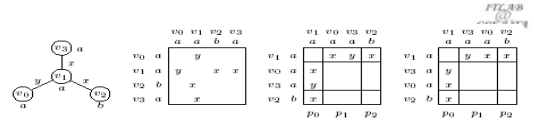
- Vertex invariants – do not change across isomorphism mappings (e.g., degree or label of a vertex)
- Do not asymptotically change the computational complexity; in practicce it is useful

Vertex Invariants

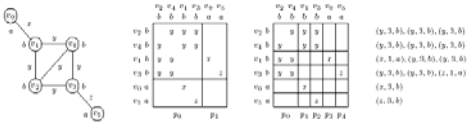
- attributes or properties assigned to a vertex which do not change across isomorphism mappings.
- partition the vertices into equivalence classes such that all the vertices assigned to the same partition have the same values for the vertex invariants.
- only maximize over those permutations that keep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

Invariants

- degree or label of a vertex
- the labels and degrees of their adjacent vertices (neighbor list)
- information about their adjacent partitions

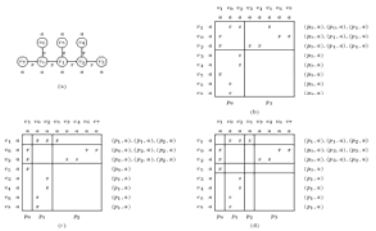


only 1! * 2!= 2 permutations although the total permutations 4! = 24.



- ($l(e)$; $d(v)$; $l(v)$) $l(e)$ 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! To 2! (1! * 2! * 1!).

Iterative Partitioning



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.

FSG vs. Subdue

- No inexact graph matching
- No iterative discovery
- Restricts input in order to be more efficient
  - Undirected edges only
  - Set of disconnected graphs
- Optimizations rely on additional space for increased speed

gSpan

- Given a graph dataset, D = $\{G_0,G_1,\ldots G_n\}$ and any subgraph g, the problem of frequent subgraph mining is to find any subgraph g such that support(g) > minSupport
- Unlike FSG, gSpan discovers frequent substructures without candidate generation.
- gSpan builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan adopts the depth-first search strategy to mine frequent connected subgraphs efficiently.
- gSpan discovers all the frequent subgraphs without candidate generation and false positives pruning. It combines the growing and checking of frequent subgraphs into one procedure, thus accelerating the mining process.

gSpan features

- In the context of frequent subgraph mining, the Apriori-like algorithms meet two challenges:
  - candidate generation: the generation of size (k+1) subgraph candidates from size k frequent subgraphs is more complicated and costly and
  - pruning false positives: subgraph isomorphism test is an NP complete problem, thus pruning false positives is costly.

If the entire graph dataset can fit in main memory, gSpan can be applied directly; otherwise, one can first perform graph-based data projection and then apply gSpan

- Subgraph isomorphism is achieved using the canonical labeling techniques, DFS lexicographic order and minimum DFS code.

DFS Code

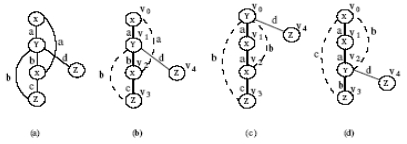


Figure 1. Depth-First Search Tree

edge	(Fig 1b) α	(Fig 1c) β	(Fig 1d) γ
0	$(0, 1, X, a, Y)$	$(0, 1, Y, a, X)$	$(0, 1, X, a, X)$
1	$(1, 2, Y, b, X)$	$(1, 2, X, a, X)$	$(1, 2, X, a, Y)$
2	$(2, 0, X, a, X)$	$(2, 0, X, b, Y)$	$(2, 0, Y, b, X)$
3	$(2, 3, X, c, Z)$	$(2, 3, X, c, Z)$	$(2, 3, Y, b, Z)$
4	$(3, 1, Z, b, Y)$	$(3, 0, Z, b, Y)$	$(3, 0, Z, c, X)$
5	$(1, 4, Y, d, Z)$	$(0, 4, Y, d, Z)$	$(2, 4, Y, d, Z)$

Table 1. DFS codes for Fig. 1(b)-(d)

Min DFS Code and Isomorphism

- If we order all DFS codes according to the < order, we can take a minimum, min(G)
- min(G) is unique
- Two graphs are isomorphic iff min(G) = min(G')

Observations

- gSpan is a main memory algorithm
- Performance is reported for data sets up to only 320 KB
- Running time scales exponentially with large numbers of graph labels
- gSpan typically needs random access to elements of the graph database and to its projections

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	✓	✓	✓	✓	×

Why Database Mining?

- Proliferation of relational DW and the need to mine them
- Data mining must ``co-exist'' with OLAP and other decision-support applications
- DM will be a sub-process in next generation Business Intelligence (BI) Systems
- Leverage the RDBMS technology for mining
- Provide an integrated decision-support environment for analysts

Data Mining Vs. Database Mining

- Data mining refers to main memory algorithms for mining
  - **+ Can use arbitrary data structures**
  - **+ Can optimize algorithms with proper representation (hash tree for example)**
  - **- Limited memory, add buffer management**
  - **- Data has to be siphoned out of its location (mostly a DBMS or a Data Warehouse)**
  - **- Works well only for small data sizes (no scalability)**
  - **- Every time data is added to the DB, the process has to be repeated**

Solution? *Database Mining* – ***Bringing algorithms to data instead of taking data to algorithms***

SQL-based Mining: Advantages

- Leverage 2+ decades of DBMS R&D
- No specialized data structures and memory management
- Fast development of mining algorithms
- SMP parallelism for free for parallel database engines
- Data is not replicated outside of DBMS
- SQL may be extended to include *ad hoc* mining queries

Scalability Issues

- Subdue is a main memory algorithm.
- Good performance for small data sizes
- Entire graph is constructed before applying the mining algorithm
- Takes a very long time to initialize for 1600K edges and 800K vertices graph
- Scalability is an issue
- Performs hierarchical reduction of input

SQL-Based Graph Mining

- We have mapped the Subdue algorithm using SQL (exact graph match)
  - Handles multiple edges
  - Handles cycles
  - Performs Hierarchical reduction
  - Have developed DMDL tailored to databases
- Can handle graphs of Millions of edges and vertices
- Working on inexact matching

Data Mining Evolution

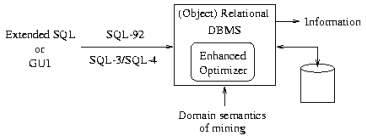
- **File-Based** or Main Memory mining algorithms
  - Data mining
- **SQL-Based** mining algorithms
  - Database mining
- Parallel mining Algorithms
  - Both main memory and SQL-based

Database Mining Spectrum

- Database Mining
  - Single database (Directly) e.g. Intelligent Miner
  - Single relation (using JDBC)
  - Layered (multiple relations, using JDBC)
  - Layered (Across databases, using JDBC)
  - Integrated Database mining

Long-Term Vision

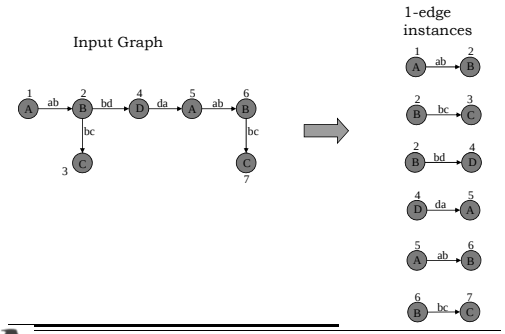
- Unbundle bulky mining operations
- Identification of Common operators
- Integration of the above into the Query Optimizer
- No distinction between OLAP and mining



HDB-Subdue

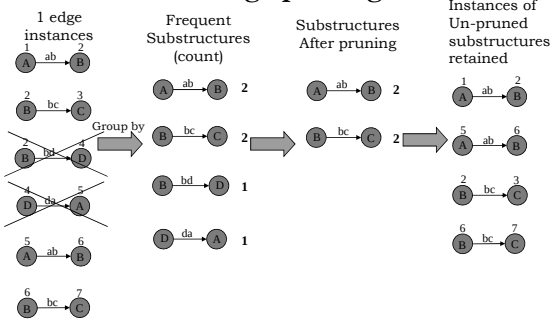
- Graph representation using relations
- Joins used for iterative generation of larger substructures
- Pseudo duplicate elimination involves a number of joins
- DMDL is used to identify the best substructure (count/frequency can be used as well)

HDB-Subdue



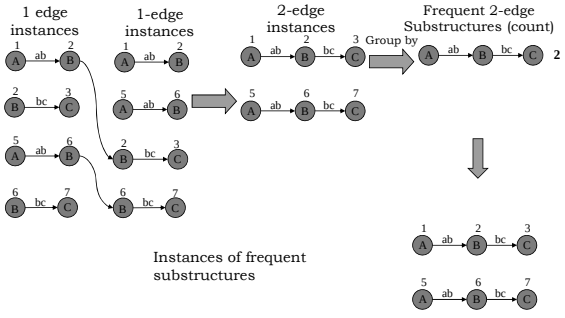
h13

1 edge pruning



h14

Generating 2 edge substructures



Slide 107

h13 during this presentation i might interchange the use of substructure and instances, but strictly speaking substructure is free of the vertex numbers. subgraphs with same vertex and edge labels as a substructure, but different vertex numbers are called its instances
hari, 11/13/2005

Slide 108

h14 hari, 11/17/2005

Representing input graph using relations

Input stored in two tables

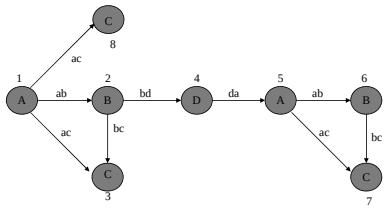
Vertices

vertexno	vertexlabel
1	A

Edges

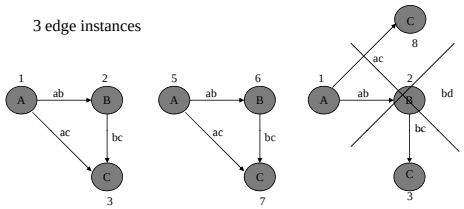
vertex1	vertex2	edgename
1	2	ab

Representation of a Substructure



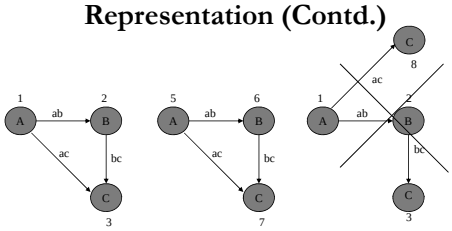
Representation (Contd.)

3 edge instances



Isomorphic instance Count 3

Representation (Contd.)

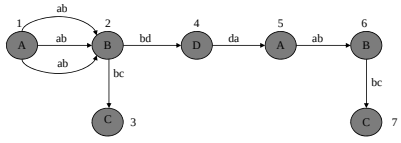


HDB instance_3 (instances of size 3)

V1	V2	V1	V3	V2	V4	V1L	V2L		V3L		V4L		E1		E2		E3		F1	T1	F2	T2	F3	T3
1	2	1	3	2	0	3	A	3	B	A	C	B	-	C	ab	C	bc	ab	ac	b1	a2	2	3	1	3
5	6	5	7	6	0	7	A	7	B	A	C	B	-	C	ab	C	bc	ab	ac	b1	a2	2	3	1	3
1	2	1	3	2	8	3	A	8	B	A	C	B	C	C	ab	C	bc	ab	ac	b1	a2	2	3	1	4

Count 2

Need for Edge numbers



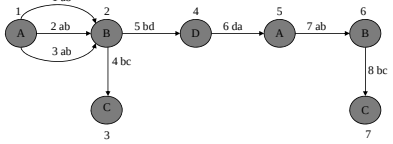
vertex1	vertex2	edge1	vertex1name	vertex2name
1	2	ab	A	B
1	2	ab	A	B
1	2	ab	A	B
.	.	.	.	.

EDB-Oneedge table

vertex1	vertex2	edge No	edge1	vertex1name	vertex2name
1	2	1	ab	A	B
1	2	2	ab	A	B
1	2	3	ab	A	B
.	.	.	.	.	.

HDB-Oneedge table

Constrained Expansion



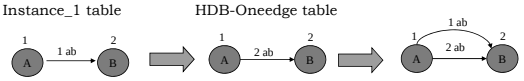

```
INSERT INTO instance_n
    ( SELECT  i.vertex1 ..  i.vertex(n),  o.vertex2,  i.vertex1name ..  i.vertex(n)name,
              o.vertex2name,  i.edge1 ..  i.edge(n),  o.edge, i.ext1 , 1
      FROM    instance(n-1) i, oneedge o
      WHERE  i.vertex(k) = o.vertex1  and  i.vertex(k+1) < o.vertex2  and .. i.vertex(n) <
              o.vertex2
    )
```

Slide 112

h15 short hand notation, V for vertex, E refers to edge, L refers to label, X refers to extension
hari, 11/13/2005

Unconstrained expansion

Instance_1 table HDB-Oneedge table



vertex1	vertex2	edge1No	edge1	vertex1name	vertex2name
1	2	1	ab	A	B
.	.	.	.	.	.

Instance_1 table

<> ⋈

=

vertex1	vertex2	edgeNo	edge	vertex1name	vertex2name
1	2	2	ab	A	B
.	.	.	.	.	.

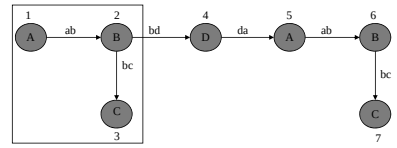
HDB-Oneedge table

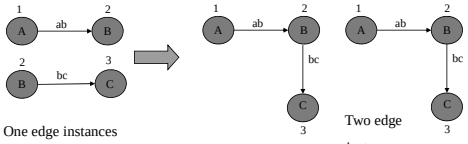
WHERE i.vertex1=o.vertex1 and o.edgeNo <> i.edge1No

Expansion

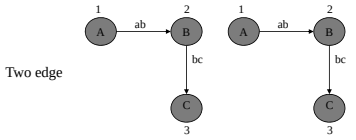
- To expand a 1-edge substructure with 2 vertices V1 and V2, we need to do:
  - Self edge (loop) substructures on v1 and v2
  - Multiple edges between v1 and v2
  - Outgoing edges from v1 and v2
  - Incoming edges to v1 and v2
- In general, to expand a substructure of size n, we need n**2 + 2n queries

Unconstrained expansion & Pseudo duplicates





Pseudo duplicates (Contd..)



Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2
1	2	3	A	B	C	AB	BC	1	2	2	3
.	.	.	.	.	.	.	.	.	.	.	.

Pseudo duplicates (Contd..)

In SQL, only rows can be sorted

Tasks:

- Convert **columns into rows**
- **Sort** the **rows**
- Convert **rows into columns**

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2
2	3	1	B	C	A	BC	AB	1	2	3	1
.	.	.	.	.	.	.	.	.	.	.	.

Unsorted_V

V	VL	POS
2	B	1
3	C	2
1	A	3
.	.	.

E	F	T
BC	1	2
AB	3	1
.	.	.

Unsorted_E

h16

Updating connectivity attributes

Unsorted_V

V	VL	POS
2	B	1
3	C	2
1	A	3
.	.	.

Sort on V

Sorted_V

V	VL	POS	NEW POS
1	A	3	1
2	B	1	2
3	C	2	3
.	.	.	.

3 Way Join

Sorted_V

V	VL	POS	NEW POS
1	A	3	1
2	B	1	2
3	C	2	3
.	.	.	.

Unsorted_E

E	F	T
BC	1	2
AB	3	1
.	.	.

Sorted_V

V	VL	POS	NEW POS
1	A	3	1
2	B	1	2
3	C	2	3
.	.	.	.

Updated_E

E	F	T
BC	2	3
AB	1	2
.	.	.

Sort on F, T

Sorted_E

E	F	T
AB	1	2
BC	2	3
.	.	.

Slide 120

h16 position changes and the f and t values pointing to old position
hari, 11/14/2005

Reconstructing instance table

Sorted_V

V	VL	POS	NEW POS
1	**A**	3	1
2	B	1	2
3	C	2	3
.	.	.	.

Sorted_V

V	VL	POS	NEW POS
1	A	3	1
2	**B**	1	2
3	C	2	3
.	.	.	.

Sorted_V

V	VL	POS	NEW POS
1	A	3	1
2	B	1	2
3	**C**	2	3
.	.	.	.

Sorted_E

E	F	T
AB	**1**	**2**
BC	2	3
.	.	.

Sorted_E

E	F	T
AB	1	2
BC	**2**	**3**
.	.	.

2n+1 Way Join for reconstruction

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2
1	2	3	A	B	C	AB	BC	1	2	2	3
.	.	.	.	.	.	.	.	.	.	.	.

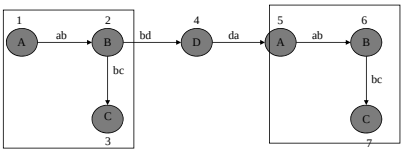
Pseudo duplicates (Contd..)

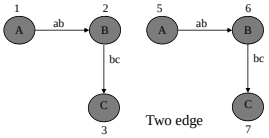
- Group By Vertex numbers and edge direction attributes
- Retain one and eliminate other

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2
1	2	3	A	B	C	AB	BC	1	2	2	3
~~1~~	~~2~~	~~3~~	~~A~~	~~B~~	~~C~~	~~AB~~	~~BC~~	~~1~~	~~2~~	~~2~~	~~3~~
.	.	.	.	.	.	.	.	.	.	.	.

Instance_2 table

Canonical label ordering



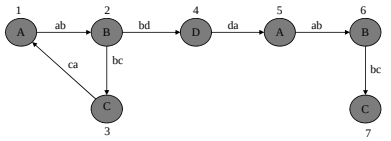


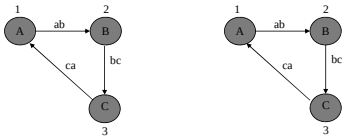
Two edge instances

Canonical label ordering

- These two are NOT duplicates
- They need to be recognized as isomorphic to each other
- In order to do this vertices and labels need to be canonically ordered
- This process for canonical ordering is similar to the process used for psuedo duplicate elimination

Cycles – Marking repeated vertex



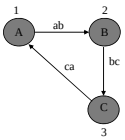


Three edge starting with ab

Three edge starting with bc

h17

Cycles – Marking repeated vertex (Contd..)



V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
1	2	3	1	A	B	C	A	AB	BC	CA	1	2	2	3	3	4
2	3	1	2	B	C	A	B	BC	CA	AB	1	2	2	3	3	4

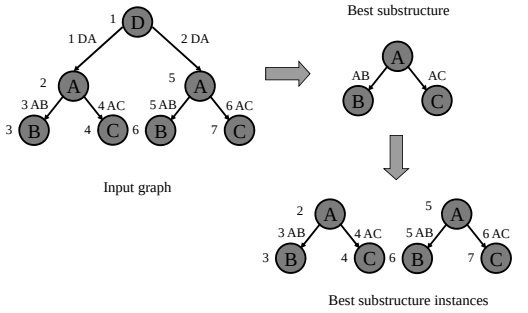
Instance_3 table

V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
~~0~~	~~1~~	~~3~~	0	A	~~A~~	~~C~~	-	AB	BC	CA	~~1~~	~~3~~	~~3~~	~~4~~	~~4~~	~~1~~
~~0~~	~~3~~	~~1~~	0	B	~~C~~	~~A~~	-	~~BC~~	~~CA~~	~~AB~~	~~1~~	~~3~~	~~3~~	~~4~~	~~4~~	~~1~~

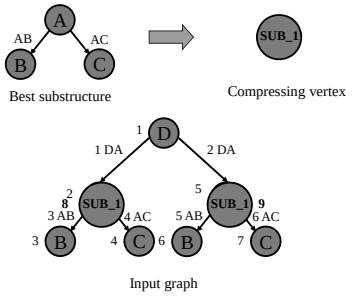
Instance_3 table with vertex invariants 0's and –'s

h17 we will not refer to the vertices marked with invariance markers
hari, 11/14/2005

Hierarchical Reduction



Hierarchical Reduction (Contd..)



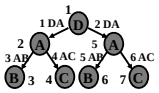
Hierarchical Reduction (Contd..)

Tasks in Hierarchical Reduction:

1. Select the best substructure after each iteration
2. Identify the instances of the best substructure
3. For each instance
   (1) Remove the vertices from the input graph
   (2) For every instance removed include a new vertex to the input graph
   (3) Remove the edges from the input graph
   (4) Update the vertex number of the edges that are incident on or going out of the compressed instance
4. The compressed input graph participates in the next iteration

h18

Selecting the best substructure

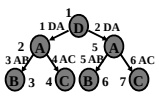


subfold_2

V1L	V2L	V3L	E1	E2	F1	T1	F2	T2	COUNT	DMDL
A	B	C	AB	AC	1	2	1	3	2	1.71
D	A	B	DA	AB	1	2	2	3	2	1.34
D	A	C	DA	AC	1	2	2	3	2	1.34
D	A	A	DA	DA	1	2	1	3	1	0.75

Best substructure is one with highest DMDL value

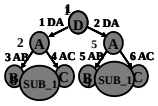
Identify instances of best substructure



V1	V2	V3	V1L	V2L	V3L	E1N	E2N	E1	E2	F1	T1	F2	T2	DMDL
2	3	4	A	B	C	3	4	AB	AC	1	2	1	3	1.71
5	6	7	A	B	C	5	6	AB	AC	1	2	1	3	1.71

BestInstances

Updating vertex table



V1	V2	V3	V1L	V2L	V3L	E1N	E2N	E1	E2	F1	T1	F2	T2	DMDL
2	3	4	A	B	C	3	4	AB	AC	1	2	1	3	1.71
5	6	7	A	B	C	5	6	AB	AC	1	2	1	3	1.71

BestInstances

Vertex table

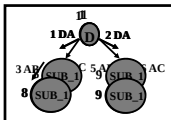
V	VL	VL
1	D	D
~~2~~	~~A~~	SUB_1
~~3~~	~~B~~	SUB_1
4	C	
5	A	
~~6~~	~~B~~	
7	C	

```
DELETE FROM VertexTable
WHERE VertexNo IN (
        (SELECT V1 FROM BestInstances)
UNION (SELECT V2 FROM BestInstances)
        . . . .
UNION (SELECT Vn+1 FROM BestInstances))
```

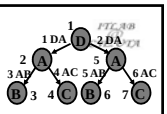
Slide 130

h18 (1) Sort on dmdl value and select one with highest dmdl value because that is the substructure that will compress the graph better.

hari, 10/12/2005



Updating oneedge table



V1	V2	V3	V1L	V2L	V3L	E1N	E2N	E1	E2	F1	T1	F2	T2	DMDL
2	3	4	A	B	C	3	4	AB	AC	1	2	1	3	1.71
5	6	7	A	B	C	5	6	AB	AC	1	2	1	3	1.71

BestInstances

```
DELETE FROM oneedge
WHERE E1N IN (
        (SELECT V1 FROM BestInstances)
UNION (SELECT E2N FROM BestInstances)
UNION (SELECT Vn+1 FROM BestInstances WHERE rownum = 1))
        ....
UNION (SELECT EnN FROM BestInstances))
UPDATE oneedge set V2 = MAX_VERTEX
WHERE V2 IN (
        (SELECT V1 FROM BestInstances WHERE rownum = 1)
        ....
UNION (SELECT Vn+1 FROM BestInstances WHERE rownum = 1))
```

Oneedge table

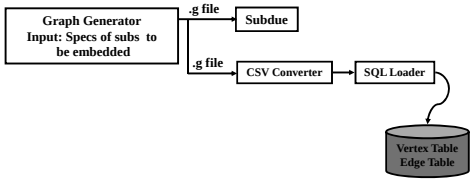
EL	EN	V1L	V2L	V	V1	V2
DA	1	D	SUB_1	1	1	2
DA	2	D	SUB_1	1	1	5
AB	3	A	B	2	3	
AC	4	A	C	2	4	
AB	5	A	B	5	6	
AC	6	A	C	5	7	

h19

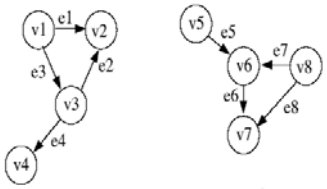
Experimental Results

Setup:

- Input graphs generated using the graph generator developed by AI Lab
- Platform: Linux
- Database: Oracle 10g
- Machine's memory: 2 Gbytes
- Number of processors: 2

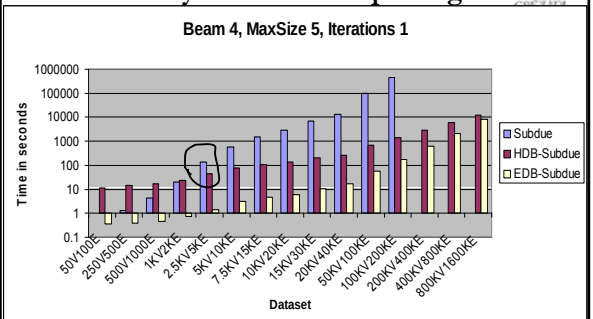


No cycles and multiple edges



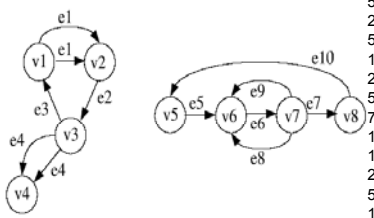
Dataset	Instances
50V100E	4
250V500E	15
500V1000E	30
1KV2KE	60
2.5KV5KE	150
5KV10KE	300
7.5KV15KE	450
10KV20KE	600
15KV30KE	900
20KV40KE	1200
50KV100KE	3000
100KV200KE	6000
200KV400KE	12000
400KV800KE	24000
800KV1600KE	48000

No cycles and multiple edges



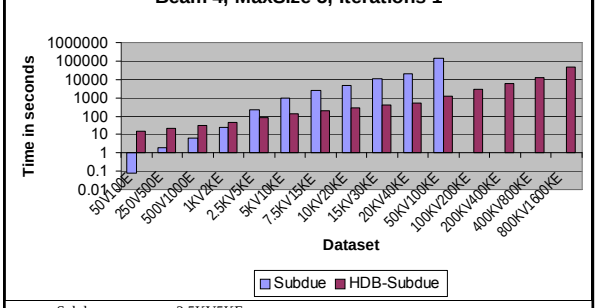
- Running time grows exponentially with graph input size
- Subdue crossover – 2.5KV5KE

With cycles and multiple edges



Dataset	Instances
50V100E	4
250V500E	15
500V1000E	30
1KV2KE	60
2.5KV5KE	150
5KV10KE	300
7.5KV15KE	450
10KV20KE	600
15KV30KE	900
20KV40KE	1200
50KV100KE	3000
100KV200KE	6000
200KV400KE	12000
400KV800KE	24000
800KV1600KE	48000

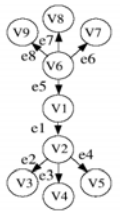
With cycles and multiple edges



- Subdue crossover – 2.5KV5KE

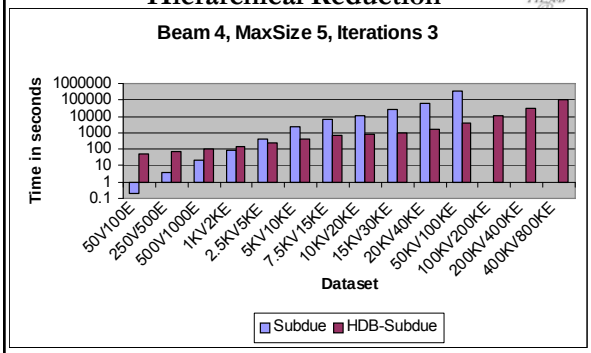
h19 Thanks to AI lab who has given the synthetic graph generator.
hari, 10/13/2005

Hierarchical Reduction



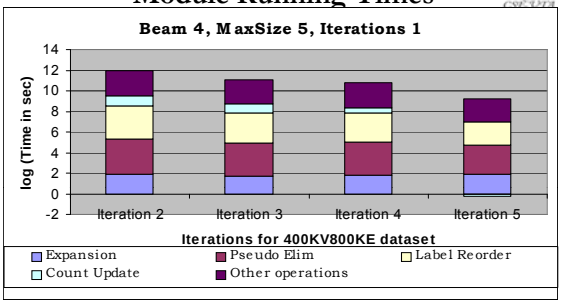
Dataset	Instances
50V100E	2
250V500E	10
500V1000E	20
1KV2KE	40
2.5KV5KE	80
5KV10KE	160
7.5KV15KE	250
10KV20KE	320
15KV30KE	420
20KV40KE	640
50KV100KE	1280
100KV200KE	2560
200KV400KE	5120
400KV800KE	10240
800KV1600KE	20480

Hierarchical Reduction



- Subdue crossover – 2.5KV5KE

Module Running Times



- Pseudo duplicate elimination takes most of the time because of 2n+1 way join
- Expansion time decreases with iteration as the number of tuples handled in each iteration becomes less

DB-FSG

- DB-FSG is a relational database approach for frequent subgraph mining.
- Addresses scalability issues much better than the main memory algorithm.
- It uses database relations to represent a graph
- Steps of DB-FSG
  - Candidate Generation
  - Frequency counting
  - Sub-graph pruning

Comparison chart of FSG and DB-FSG

	FSG	DB-FSG
Sub-graphs Representation	Sparse Adjacency Matrix Graphs without cycles and multiple edges	Tuples of instance_n relation represents subgraphs of size n Can handle graphs with cycles and multiple edges
Candidate Generation	Joins two frequent subgraphs of size n that has same core of size n-1 to generate size n+1 candidate subgraphs Uses canonical labeling to avoid duplicates Uses downward closure property to retain true candidates	Perform SQL join of instance_n relation and one_edge relation to generate candidate subgraphs of size n+1 Uses edge code approach for removing pseudo duplicates.
Frequency Counting	Uses canonical labeling for subgraph isomorphism and frequency counting	Uses Canonical ordering on vertex labels and group by function for frequency counting

Representing Graphs in DB-FSG

Input stored in two tables

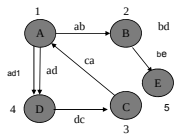
Vertices

Vertex no	Vertex label	gid
1	a	1

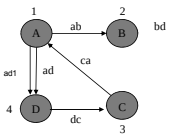
Edges

Vertex1	Vertex 2	Edge label	gid
1	2	ab	1

Representation of Instances of Substructures



Graph Id 1

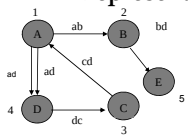


Graph Id 2

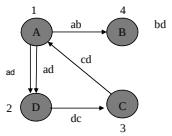
One_edge Table

vertex1	vertex2	edge1	vertex1name	vertex2name	gid
1	2	ab	A	B	1
1	2	ab	A	B	2
.	.	.	.	.	.

Representation of Instances (cont…)



Graph Id 1

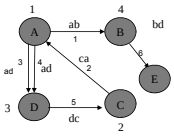


Graph Id 2

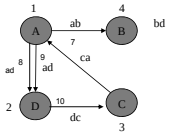
Instance_3 Table

V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F 2	T 2	F 3	T 3	gid
1	2	3	4	A	B	C	D	ab	ad	dc	1	2	1	3	4	3	1
1	4	3	2	A	D	C	B	ad	ab	dc	1	2	1	4	2	3	2
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Representing Cycles and Multiple Edges



Graph Id 1



Graph Id 2

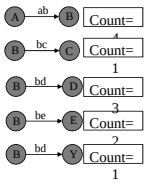
Instance_3 Table

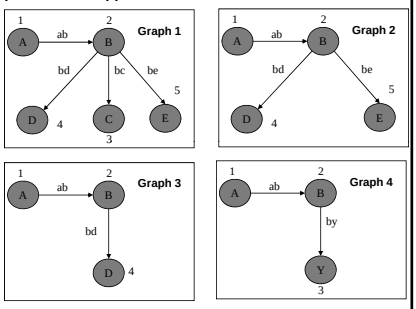
V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F 1	T 1	F 2	T 2	F 3	T 3	gid	ecode
1	3	4	0	A	C	D	-	ad	ca	dc	1	3	2	1	3	2	1	1,2,4,5
1	3	4	0	A	C	D	-	ad	ca	dc	1	3	2	1	3	2	2	2,7,9,10
1	3	4	0	A	C	D	-	ad	ad	dc	1	3	1	4	3	2	1	1,3,4,5
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Overview of DB-FSG

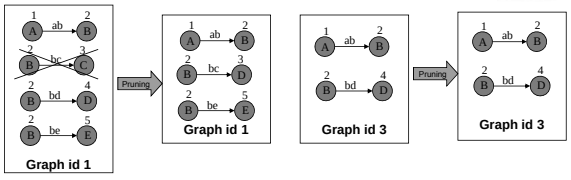
Input parameters support = 50% and size = 3

Frequency of the edges

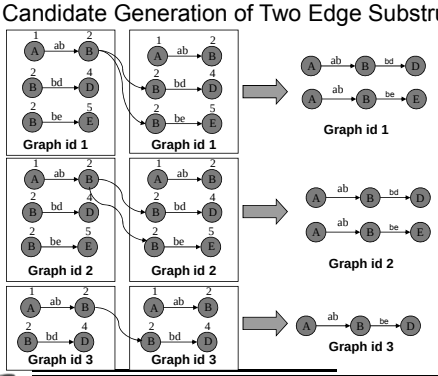




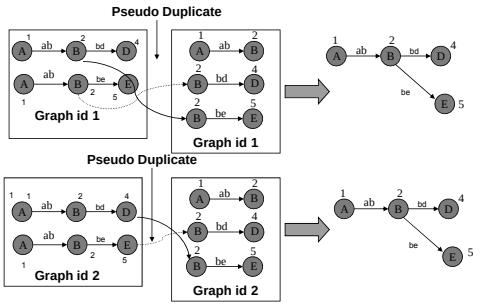
Candidate Generation of One Edge Substructure



Candidate Generation of Two Edge Substructure

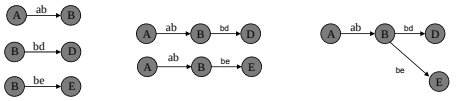


Candidate Generation of Three Edge Substructure

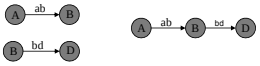


Output

Output for 50% support

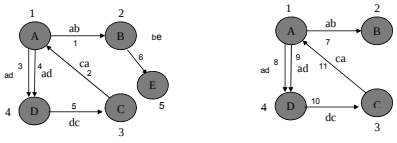


Output for 75% support



Need for Edge Numbers

- To avoid redundant expansion on edges that already exists in the instance
- To detect pseudo-duplicates

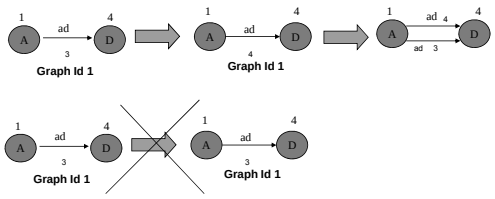


Oneedge Table

vertex1	vertex2	edge1	edgeno	vertex1name	vertex2name	gid
1	2	ab	1	A	B	1
1	2	ab	7	A	B	2
.	.	.	.	.	.	.

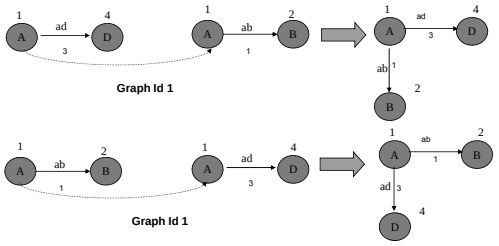
Need for Edge Numbers (cont…)

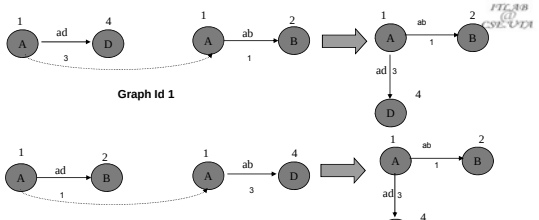
WHERE i.vertex1 = o.vertex1 and i.vertex2=o.vertex2 and i.gid = o.gid and **i.edgeno <> o.edgeno**



Pseudo Duplicates

- Due to the unconstrained expansion of substructure there is the probability of same instances being expanded in various ways.





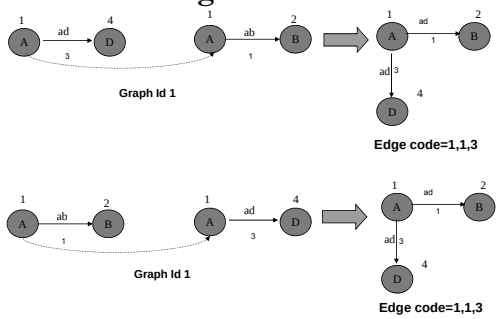
Instance_2 Table

V1	V2	V3	V1L	V2L	V3L	E1	E2	EN1	EN2	F1	T1	F2	T2	gid
1	4	2	A	D	B	ad	ab	3	1	1	2	1	3	1
1	2	4	A	B	D	ab	ad	1	3	1	2	1	3	1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Detecting Pseudo duplicates using Edge Code

- Each edge in a graph has unique edge number
- All pseudo duplicates have same edges and edge number in different order.
- Hence, we can construct a unique code based on edge numbers and gid
- Edge code is a string formed by concatenating gid with edge numbers sorted in ascending order and separated by comma

Detecting Pseudo duplicates using Edge Code



Detecting Pseudo duplicates using Edge Code

- If we have edge code for instance of size n then constructing edge code for instance of size n+1 expanded from the instance is just placing the new edge number in the proper position in edge code
- Hence the complexity is O(n) for constructing edge code for n+1 size instance from expanded from n sized instance

Pseudo Duplicates

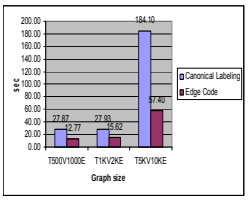
- HDB-Subdue uses canonical ordering on vertex number for elimination of pseudo duplicates.
- Canonical ordering requires maintaining six intermediate tables, sorting of two intermediate tables, one 3 way join and one 6n+2 way join.

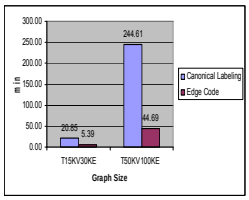
Processing Time of Edge Code Approach VS Canonical Ordering

- Max Substructure size 5

Graph Size	Canonical Ordering on Vertex No	Edge Code	Efficiency
500V1KE	27.87 sec	12.77 sec	218 %
1KV2KE	27.93 sec	15.62 sec	179 %
5KV10KE	184.10 sec	57.40 sec	321 %
15KV30KE	1251.06 sec	316.87 sec	387 %
50KV100KE	14676.83 sec	2653.88 sec	547 %

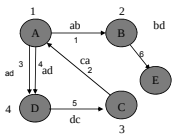
Edge Code Approach VS Canonical Ordering



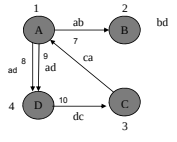


Canonical Ordering on Vertex Label

- Due to unconstrained expansion, instances of same substructure may be in different order.



Graph Id 1



Graph Id 2

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2	Gid	ecode
1	**2**	**4**	**A**	**B**	**D**	**ab**	**ad**	**1**	**2**	**1**	**3**	**1**	**1,1,4**
1	**4**	**2**	**A**	**D**	**B**	**ad**	**ab**	**1**	**2**	**1**	**3**	**2**	**2,7,9**
.	.	.	.	.	.	.	.	.	.	.	.		

Canonical Ordering on Vertex Label

SQL allows to sort only the rows

Convert **columns into rows**

- **Sort** the **rows**
- Convert **rows into columns**

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2	gid	ecode	Id
1	2	4	A	B	D	ab	ad	1	2	1	3	1	1,1,4	1
1	4	2	A	D	B	ad	ab	1	2	1	3	2	2,7,9	2
.	.	.	.	.	.	.	.	.	.	.	.			

gid	Id
1	1
2	2
.	

Label_gid_V

V	VL	POS	Id
1	A	1	2
4	D	2	2
2	B	3	2
.	.	.	

Unsorted_V

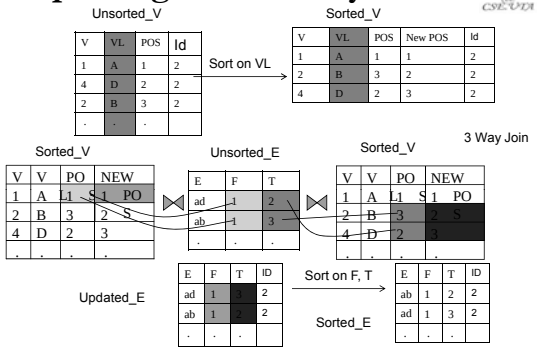
E	F	T	Id
ad	1	2	2
ab	1	3	2
.	.	.	

Unsorted_E

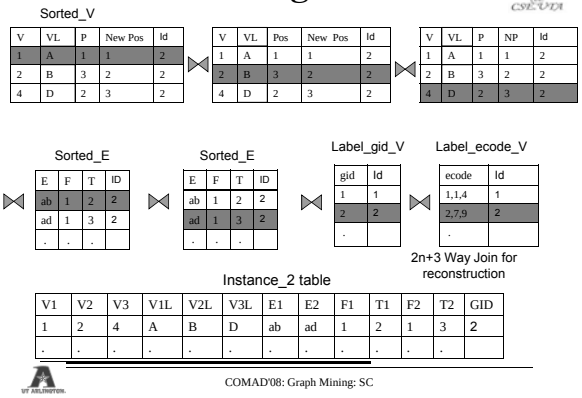
ecode	Id
1,1,4	1
2,7,9	2
.	

Label_ecode_V

Updating connectivity attributes



Reconstructing instance table



Canonical ordering

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2	gid	ecode	Id
1	2	4	A	B	D	ab	ad	1	2	1	3	1	1,1,4	1
1	4	2	A	D	B	ad	ab	1	2	1	3	2	2,7,9	2
.	.	.	.	.	.	.	.	.	.	.	.			

Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2	Gid
1	**2**	**4**	**A**	**B**	**D**	**ab**	**ad**	**1**	**2**	**1**	**3**	**1**
1	**2**	**4**	**A**	**B**	**D**	**ab**	**ad**	**1**	**2**	**1**	**3**	**2**
.	.	.	.	.	.	.	.	.	.	.	.	

Frequency Counting and Substructure Pruning

- Support count=(support X #Graphs)/100
- For each graph only one instance per substructure is included for frequency counting of substructures
- Instances of substructure with frequency more than or equal to support count is retained.

Slide 165

h20 position changes and the f and t values pointing to old position

hari, 11/14/2005

Frequency Counting and Substructure Pruning

- Insert into dist_i (
  select distinct v1L,v2L,…, vi, e1, e2 ,…, ei, F1,T1, F2, T2,…, Fi, Ti, gid
  from instance_n)
- Insert into sub_fold_i (select v1L,v2L,…, vi, e1, e2 ,…, ei, F1,T1, F2, T2,…, Fi, Ti
  from dist_n
  group by v1L,v2L,…, vi, e1, e2 ,…, ei, F1,T1, F2, T2,…, Fi, Ti
  having count(*)>=support count)

Frequency Counting and Substructure Pruning

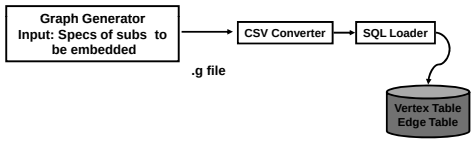
- Insert into instanceiter_i (
  select s.v1, s.v2 ,…,s. vn, s.v1L, s.v2L, …, s.vi, s.e1, s.e2 ,…, s.ei, s.e1n, s.e2n, …, s.ein, s.F1, s.T1, s.F2, s.T2,…, s.Fi, s.Ti, s.gid, s.ecode
  from instance_i s, sub_fold_i b
  where s.v1L=b.v1L and s.v2L=b.v2L and …and s.viL = b.vil and s.e1=b.e1 and s.e2=b.e2 and … and s.ei=b.ei)

121

Experimental Results

Setup:

- **Modification to the graph generator (developed by AI Lab) was done to generate input dataset**
- **Platform: Linux**
- **Database: Oracle 10g**
- **Machine's memory: 2 Gbytes**
- **Number of processors: 2**



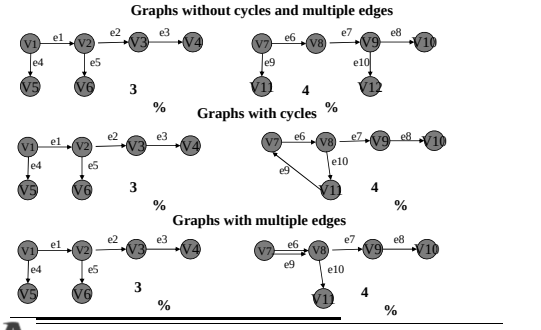
Experiment Dataset DB-FSG

- Data sets without cycles and multiple edges
  - Substructures with support values 3% and 4%
- Data sets with cycles
  - Substructures with support values 3% and 4%
- Data set with multiple edges
  - Substructure with support values 3% and 4%

Experiment Dataset DB-FSG

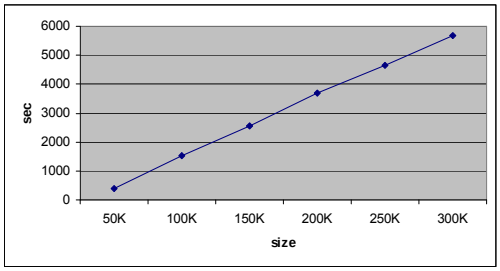
- Graph size
  - #vertices = 40; #edges = 40
- Graphs: 50K,100K,150K, …, 300K
- Total Number of nodes and edges:
  - 200K to 1.2M
- Input parameters
  - Max substructure size 5; Support 1%

Embedded Graphs

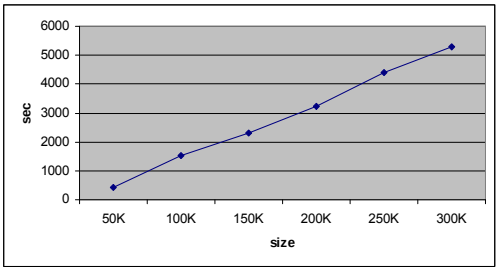


h21 Thanks to AI lab who has given the synthetic graph generator.
hari, 10/13/2005

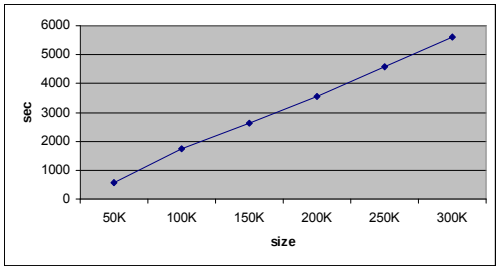
Graphs without cycles and multiple edges



Graphs with cycles



Graphs with Multiple Edges



Conclusions

- Mining algorithms can be mapped to SQL
- Absence of grouping over columns makes it not so efficient
- Hence, canonical forms are complex
- Scalability is easily obtained

Challenges

- Primitive operators inside DBMS
- Optimization of self-joins
- Efficient pseudo duplicate elimination
- Query optimization and plan generation
- Mining-aware DBMSs and SQL-aware mining systems
- Perhaps concurrency control and recovery are not needed and if turned off, can result in better performance

References

- D. J. Cook and L. B. Holder,Graph Based Data Mining, *IEEE Intelligent Systems*, 15(2), pages 32-41, 2000.
- D. J. Cook and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. In *Journal of Artificial Intelligence Research*, Volume 1, pages 231-255, 1994.
- L. B. Holder, D. J. Cook and S. Djoko. Substructure Discovery in the SUBDUE System. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 169-180, 1994
- L. B. Holder and D. J. Cook. Discovery of Inexact Concepts from Structural Data. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 5, Number 6, pages 992-994, 1993
- D. J. Cook, L. B. Holder, and S. Djoko. Scalable Discovery of Informative Structural Concepts Using Domain Knowledge. In *IEEE Expert*, Volume 11, Number 5, pages 59-68, 1996.
- Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami: Mining Association Rules between Sets of Items in Large Databases. *SIGMOID Conference 1993*: 207-216
- Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo: Discovery of frequent episodes in event sequences. Report C-1997-15, Department of Computer Science, University of Helsinki, February 1997. 45 pages.
- Diane J. Cook, Edwin O. Heierman, III Automating Device Interactions by Discovering Regularly Occurring Episodes. Knowledge Discovery in Databases 2003.
- Michihiro Kuramochi and George Karypis, Discovering Frequent Geometric Subgraphs *Proceedings of IEEE 2002 International Conference on Data Mining (ICDM '02)*, 2002

References

- Michihiro Kuramochi and George Karypis, Frequent Subgraph Discovery *Proceedings of IEEE 2001 International Conference on Data Mining (ICDM '01)*, 2001.
- X. Yan and J. Han, gspan: graph-based substructure pattern mining," Proceedings of the IEEE International Conference on Data Mining, 2002
- http://www.cse.iitd.ernet.in/~csu01124/btp/specifications.htm
- H. Bunke and G. Allerman, \Inexact graph match for structural pattern recognition," *Pattern Recognition Letter*s, pp. 245{253, 1983.
- Fortin., S., The graph isomorphism problem. 1996, Department of Computing Science, University of Alberta.
- A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD'00*, pages 13.23, 2000.
- J. Huan, W. Wang, J. Prins, and J. Yang, "SPIN: Mining Maximal Frequent Subgraphs from Graph Databases, KDD 2005, Seattle, USA.
- X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. *KDD'03*, 2003.
- Mr. Srihari Padmanabhan, "Relational Database Approach to Graph Mining and Hierarchical Reduction", Fall 2005 http://itlab.uta.edu/itlabweb/students/sharma/theses/pad05ms.pdf
- Mr. Subhesh Pradhan, "DB-FSG: An SQL-Based Approach to Frequent Subgraph Mining", Summer 2006 http://itlab.uta.edu/itlabweb/students/sharma/theses/pra06ms.pdf
- R. Balachandran, "Relational Approach to Modeling and Implementing Subtle Aspects of Graph Mining", Fall 2003. http://www.cse.uta.edu/Research/Publications/Downloads/CSE-2003-41.pdf
- M. Aery and S. Chakravarthy, "eMailSift: Email Classification Based on Structure and Content", in the Proc. of ICDM (international Conference on Data Mining), Houston, Nov 2005.