The University of Texas ARLINGTON

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
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

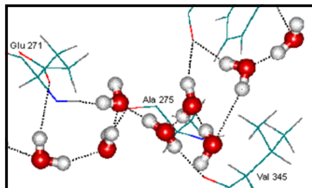
- ***Graph Mining Approaches***
  - Subdue
  - AGM
  - *FSG*
- ***SQL-Based Graph Mining***
  - *HDB-Subdue*
  - *DB-FSG (may be)*
- ***Graph mining applications***
  - Email classification
- Conclusions
- References

Need for Graph Mining

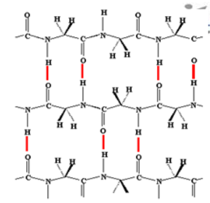
- Association rule mining, decision trees and others mining approaches
  - mine transactional data
  - Do not make use of any structural information
- Graph based mining techniques are used for mining data that are structural in nature
  - chemical compounds, complex proteins, VLSI circuits, social networks, …
  - as mapping them to other representations is not possible or will lead to loss of structural information

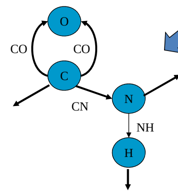
Need for Graph Mining

- Significant work in this area includes
  - Subdue substructure discovery algorithm (Cook & Holder),
  - HDB-Subdue (Chakrvarthy, Padmanabhan),
  - Apriori graph mining (AGM) (Inokuchi, Washio, and Motoda),
  - the frequent subgraph (FSG) technique (Karypis & Kuramochi), and
  - gSpan approach (J. Han), also SPIN (Huan, Wang, Prins, and Yang)
- PageRank and HITS are also graph based



Protein





Protein represented using Graph

Application:

To determine which amino acid chain dominates in a particular protein

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
- Web and social network analysis

Graph Based Data Mining

- A Graph representation is an intuitive and an obvious choice for a database that has structural information
- Graphs can be used to accurately model and represent scientific data sets. Graphs are suitable for capturing arbitrary relations between the various objects.
- Graph based data mining aims at discovering interesting and repetitive patterns within these structural representations of data.

Graph Mining: Mapping

Entities/objects	→	***Vertices***
Object's attributes	→	***Vertex label***
Relations between objects	→	***Edges between vertices***
Type of relation	→	***edge label***
Substructure	→	***Connected subgraph***
Substructure Instance	→	***Set of vertices & edges in input graph that match graph representation 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 possible; 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 (or subgraph matching) is NP-complete
- However, graph isomorphism although belongs to NP is neither known to be solvable in polynomial time nor 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 the approach and the use of representation

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 can address 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

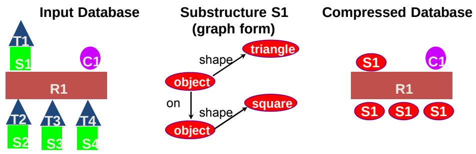
- FSG is used for frequent subgraph discovery
- Given a graph dataset $G = \{G1,G2,G3, \cdots\}$, it discovers all connected subgraphs that are found in at least the support threshold percent of the input graphs
- Uses a (sparse) adjacency matrix for graph representation
- A canonical label is generated 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

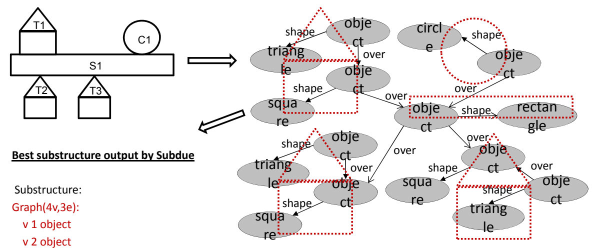
- Avoids candidate generation
- Builds a new lexicographical ordering among graphs and maps each graph to a unique minimum DFS code as its canonical label
- Seems to outperform FSG
- Amenable to parallelization
- Does not handle cycles and multiple edges

Subdue Example

- Came from AI
- Examples are different from what we normally see in mining



SUBDUE : Overview



Substructure:
Graph(4v,3e):
v 1 object
v 2 object
v 3 triangle
v 4 square
u 1 2 over
u 1 3 shape
u 2 4 shape

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

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 (not merely exact) 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)

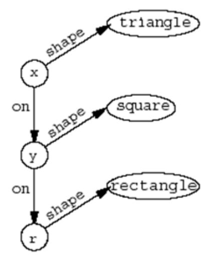
MDL Principle (cont.)

- 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

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

- High MDL value is desirable !
- 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

MDL



$$\begin{array}{r} x \\ triangle \\ y \\ square \\ r \\ rectangle \end{array} \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

MDL Details

vbits = Total number of bits to encode unique vertex labels

rbits = Total number of bits to encode adjacency matrix

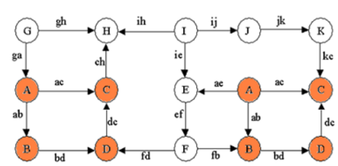
ebits = Total number of bits to encode edge labels

I(G) = vbits + rbits + ebits

$$MDL = \frac{I(G)}{I(S) + I(G|S)}$$

Subdue values substructures with high MDL as better substructures

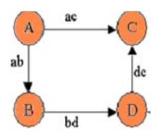
Example: Subdue



Input (partial)

- 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
- 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

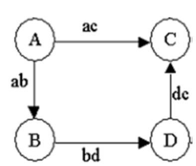
Output

- 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
- Limit is a upper bound on the number of substructures detected

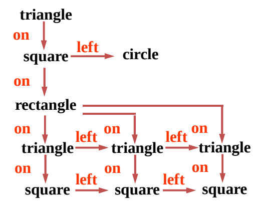
Subdue Algorithm

```
Subdue(Graph, BeamWidth, MaxBest, MaxSubSize, Limit)
   ParentList = {};     ChildList = {};     BestList = {}
   ProcessedSubs = 0
   Create a substructure from each unique vertex label and its single-vertex instances; 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    //by using MDL
            Insert Child in ChildList in order by value  //highest to lowest MDL 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
      if Length(BestList) > MaxBest then Destroy the substructure at the end of BestList
      Switch ParentList and ChildList
return BestList
```

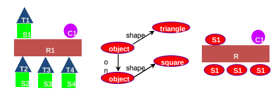

Algorithm (Contd.)

1. Create substructure for each unique vertex label

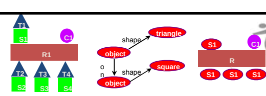
Substructures:



triangle (4), square (4), circle (1), rectangle (1)

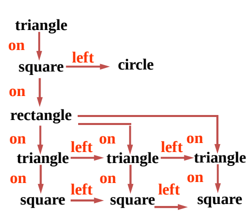


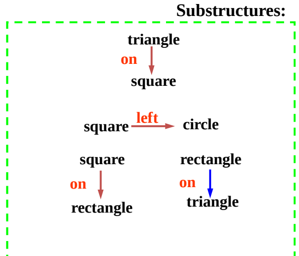
Algorithm (Contd.)



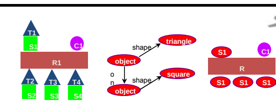
2. Expand best substructure by an edge or edge+neighboring vertex

Substructures:





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
6. 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

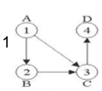
- Minimum graph edit distance

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

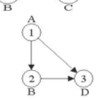
- Uses Branch and bound algorithm

- Exact graph match is NP complete
- Bunke and Allerman's approach (1983, pattern recognition letters)
  - 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

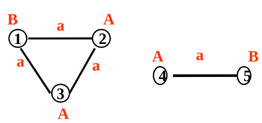
Costs

- Cost of deleting (inserting) a node with a label a is given as DELNODE(a) (INSNODE(a))
- Cost of substituting a node with syntactic label a by a node with syntactic label b are given by SUBNODE(a, b)
- Similarly, SUBBRANCH(a, b) are costs of branch substitution
- Branch deletion and insertion involves bordering nodes
- DELBRANCH(a) (INSBRANCH(a)) and DELBRANCH'(A) (INSBRANCH'(A)) are used, respectively, when the border nodes are not deleted (or not inserted), and at least one of the border nodes are deleted (or inserted)

Inexact match

- Has three cases:
  - A node can be deleted
  - A node can be substituted, and
  - A node can be inserted
- Mapping of nodes induces a mapping of branches, too. A branch is substituted/inserted/deleted when nodes are mapped

Inexact match

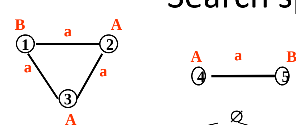


DELNODE(x) = 2
SUBNODE(A, B) = SUBNODE(B, A) = 1
DELBRANCH(y) = 2
DELBRANCH'(y) = 1
INSBRANCH(y) = 2

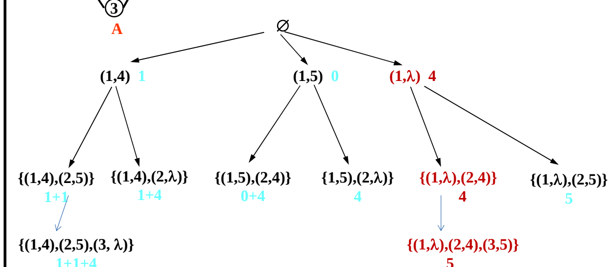
F1: 2 → 4, 3 → 5, 1 → ∅
Cost: 0 + 1 + 2 + 1 + 1 = 5

F2: 1 → 4, 2 → 5, 3 → ∅
Cost: 1 + 1 + 2 + 1 + 1 = 6

Search space



Least-cost match is {(1, λ), (2,4), (3,5)}

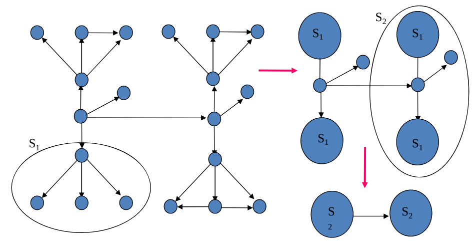


Variants of Subdue

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

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



Supervised Concept Learning Using Subdue

- Need for non-logic-based relational concept learner

SubdueCL

- Accept positive and negative graphs as input examples
  - Find hypotheses that describes positive examples and not negative examples

SubdueCL

- Find substructure compressing positive graphs, but not negative graphs
- Find substructure *covering* positive graphs, but not negative graphs
- Learn multiple rules

Concept Learning Subdue

- Positive graph G+, Negative graph G-
- Find substructure that compresses positive instances but not (or more than) negative instances
  - Value (G+, G-, S) = DL(S) + DL(G+|S) + DL(G-) – DL(G-|S)
- One of the limitations of this compression-based concept learner is that it only looks for substructures which compress the entire positive graph more than the entire negative graph.
- Therefore, it is biased to look for a substructure that offers more compression as compared to a substructure that covers a greater number of positive examples.

Concept Learning SUBDUE

- Positive graph G+
- Negative graph G-
- Alternative set covering measure
  - Error (substructure) $= \dfrac{\#PosExNotCovered + \#NegExCovered}{\#PosEx + \#NegEx}$
  - For a substructure to be good, Error should be minimum
  - Hence, Value (of a substructure) = 1 - Error

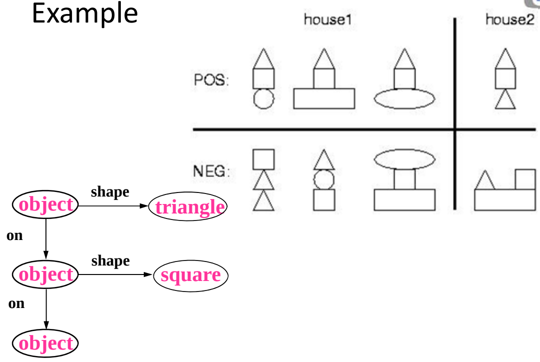
Hypotheses detection using coverage

```
Main(Gp, Gn, Beam, Limit)
   H = {};
   repeat
     repeat
        BestSub = SubdueCL(Gp, Gn, Beam, Limit)
         if BestSub = {}
              then Beam m= Beam * 1.1
     until (BestSub <> {})
     Gp = Gp  - {p in Gp | BestSub covers p}
     H = H + BestSub
   until Gp = {}
   return H
end
```


SubdueCL Algorithm

```
SubdueCL(Gp, Gn, Limit, Beam)
   ParentList = (All substructures of one vertex in Gp) mod Beam
   repeat
      BestList = {}
      Exhausted = TRUE
      i = Limit
      while ( (i > 0 ) and (ParentList ≠ {}) )
         ChildList = {}
         foreach substucture in ParentList
            C = Expand(Substructure)
            if CoversOnePos(C,Gp)
               then BestList = BestList ∪ {C}
            ChildList = ( ChildList ∪ C ) mod Beam
            i = i – 1
         endfor
         ParentList = ChildList mod Beam
      endwhile
      if BestList = {} and ParentList ≠ {}
         then Exhausted = FALSE
               Limit = Limit * 1.2
   until ( Exhausted = TRUE )
   return first(BestList)
end
```

Example



Empirical Results

- Comparison with ILP (inductive logic programming) systems
- Non-relational domains from UCI repository

	Golf	Vote	Diabetes	Credit	TicTacToe
FOIL	66.67	93.02	70.66	66.16	100.00
Progol	33.33	76.98	51.97	44.55	100.00
SubdueCL	66.67	94.88	64.21	71.52	100.00

- Subdue has also been extended for multiple classes

Graph-based Anomaly Detection [KDD03]

- Anomalous substructure detection
  - Examine entire graph
  - Report unusual (low MDL compression) substructures
    - low count
    - lower MDL
    - lower compression in subsequent passes

size * count can be used as a heuristic

Graph-based Anomaly Detection [KDD03]

- Anomalous subgraph detection
  - Partition graph into distinct, separate structures (subgraphs)
  - Determine how anomalous each subgraph is compared to others
    - How early compressed?
    - How much compression?

Graph Regularity

- Regularity (predictability) affects degree of anomaly
  - "In an insane society, the sane man appears insane"
- Conditional Substructure Entropy: for substructures of a given size, how many bits are needed, on average, to describe the surroundings of the substructure?

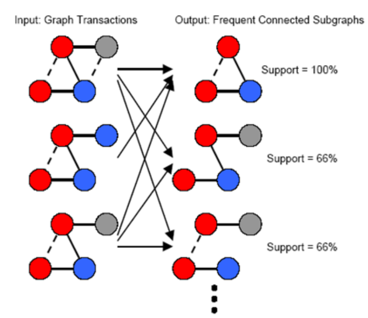
FSG

- 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

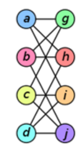
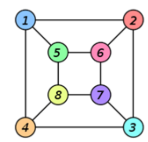


Definitions

- 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 $G1 = (V1;E1)$ and $G2 = (V2;E2)$ are *isomorphic* if they are topologically identical to each other, that is, there is a mapping from $V1$ to $V2$ such that each edge in $E1$ is mapped to a single edge in $E2$ and vice versa
- An automorphism : an isomorphism mapping where G1 = G2 (on the same graph).

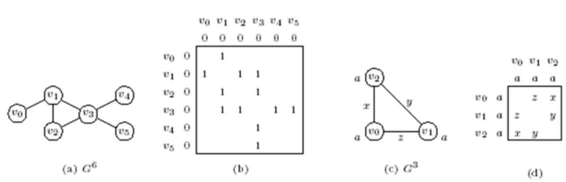
Example (from wiki)

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

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

- 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

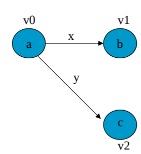
Canonical Labeling

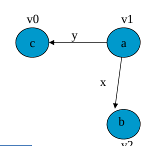


"000000 1 01 011 0001 00010"   "*aaa z xy*"

- 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|!)

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

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 and deterministic way, regardless of the original vertex and edge ordering.

Key Features Of FSG

- The complexity of finding a canonical labeling can, in practice, be substantially reduced by using the concept of vertex invariants
- Introduces various optimizations for candidate generation and frequency counting
  - (Subdue has pruning, search space minimum detection etc.)

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 with threshold as zero)
- ONLY undirected edges; I believe 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

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

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

Subdue -FSG

- Subdue discovers conceptually interesting substructures in a single graph- FSG is used to discover subs-graphs that occur frequently in a data set of graphs

- The concept of Inexact graph match implemented by Subdue cannot be implemented by canonical labeling.

Subdue-FSG

- Subdue requires to know the graph structure in the candidate generation phase but FSG does not need the adjacency list in the candidate generation phase, candidates are generated using the existing frequent substructures.
- FSG prunes the candidate substructures using the 'downward closure property', which allows a k+1 size substructure to be a candidate only if all its k subs-graphs are frequent substructures.
- Grows Vs. generates

Subdue - 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

Algorithm fsg(*D; t*)

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

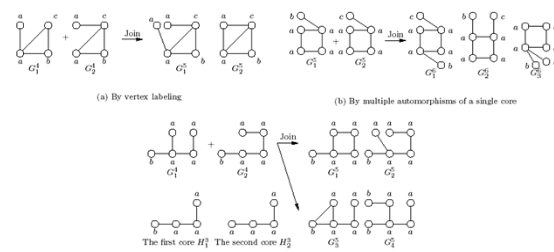

Candidate generation

- 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

Candidate generation

- Unlike joining of two k-itemsets which lead to a unique k_1-itemset, the joining of two graphs of size k can lead to multiple distinct subgraphs of size k+1
  1. Difference between the shared core and the two subgraphs can be a vertex that has the same label in both subgraphs
  2. Core itself may have multiple automorphisms and each can lead to a different k+1 candidate (worst case k!)
  3. Two frequent subgraphs may have multiple cores

Joining of two k-subgraphs



(a) By vertex labeling

(b) By multiple automorphisms of a single core

(c) By multiple cores

Candidate Generation

Algorithm 2 fsg-gen(F^k) (Candidate Generation)

$C^{k+1} \leftarrow \emptyset$
for each pair of $G_i^k, G_j^k \in F^k, i \le j$ such that $\mathrm{cl}(G_i^k) \le \mathrm{cl}(G_j^k)$ **do**
  $H^{k-1} \leftarrow \{H^{k-1} \mid$ a core H^{k-1} shared by G_i^k and $G_j^k\}$
  for each core $H^{k-1} \in H^{k-1}$ **do**
    $\{B^{k+1}$ is a set of tentative candidates$\}$
    $B^{k+1} \leftarrow$ fsg-join(G_i^k, G_j^k, H^{k-1})
    for each $G_j^{k+1} \in B^{k+1}$ **do**
      {test if the downward closure property holds}
      flag $\leftarrow$ true
      for each edge $e_l \in G_j^{k+1}$ **do**
        $H_l^k \leftarrow G_j^{k+1} - e_l$
        if H_l^k is connected and $H_l^k \notin F^k$ **then**
          flag $\leftarrow$ false
          break
      if flag = true **then**
        $C^{k+1} \leftarrow C^{k+1} \cup \{G_j^{k+1}\}$
return C^{k+1}

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.

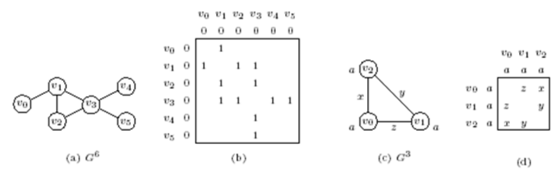
Downward Closure

- 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
- 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

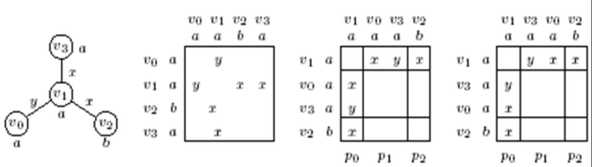
Vertex Invariants

- 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

- 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 as invariant

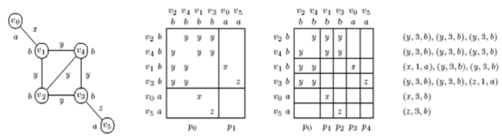


only 1! * 2!= 2 permutations although the total permutations 4! = 24.

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



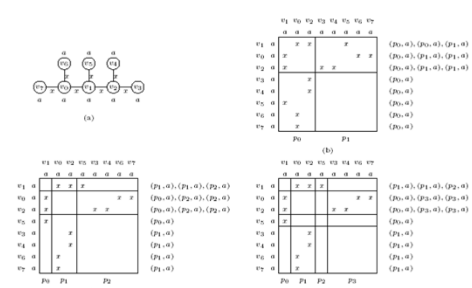
- $(l(e); d(v); l(v))$
    - $l(e)$ is the label of the incident edge e,
    - $d(v)$ is degree of the adjacent vertex v, and
    - $l(v)$ is its vertex label.
- same partition if and only if $nl(u) = nl(v)$
- reduce from 4! x 2! to 2! (1! * 2! * 1! * 1! * 1!).

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.

Chemical compound data set

Support σ [%]	Running Time[s] with Optimizations: Degree-Label Partitioning	Inverted Index	Partition Ordering	Neighbor List	Iterative Partitioning	Largest Pattern Size k^*	Candidates $\|C\|$	Frequent Patterns $\|F\|$
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

More experimental results in the paper

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

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	✓	✓	✓	✓	×

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 even to initialize for 1600K edges and 800K vertices graph
- Scalability is an issue

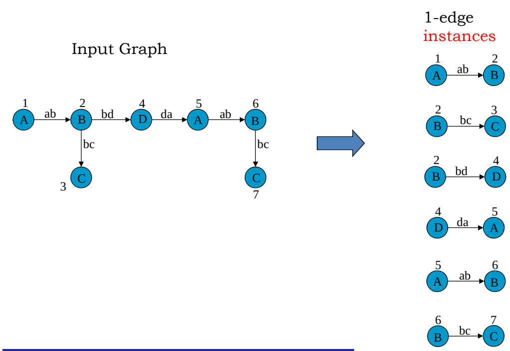
SQL-Based Graph Mining

- We have mapped the Subdue algorithm using SQL (HDB-Subdue)
  - Handles multiple edges between nodes
  - Handles cycles/loops
  - Performs Hierarchical reduction
  - Dveloped DMDL tailored to databases
- Can handle graphs of Millions of edges and vertices
- DB-FSG does frequent subgraph mining
- Working on inexact matching

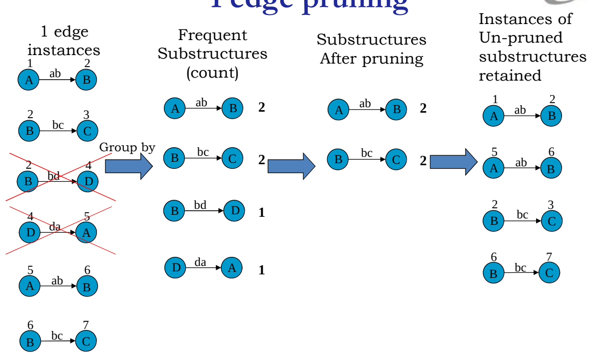
HDB-Subdue

- 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)

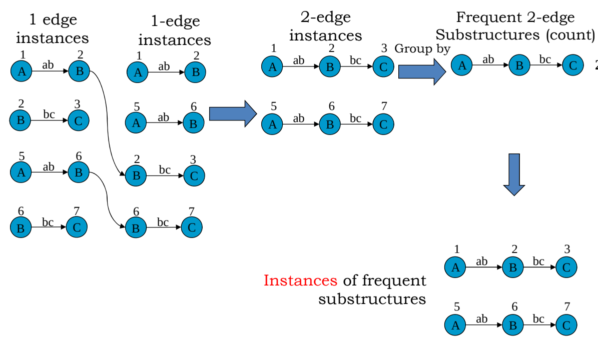
HDB-Subdue



1 edge pruning



Generating 2 edge substructures



Representing input graph using relations

Input stored in two tables

Vertices

vertexno	vertexlabel
1	A

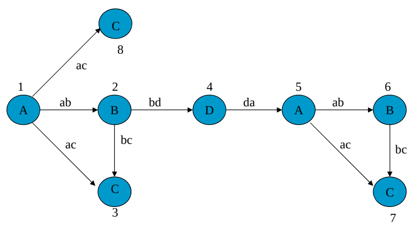
Edges

vertex1	vertex2	edgename
1	2	ab

The pseudo code of HDB-Subdue

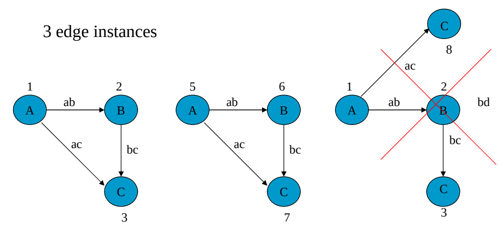
```
INPUT : (Input graph flat file, MaxSize, Beam, NumberOfIterations)
BEGIN HDB-Subdue
  Load vertices and edges from flat file into relations;
  Remove the edges which have only one instance from the edge table
  For (i=1 to NumberOfIterations) { // For Hierarchical reduction
    For (j=2 to MaxSize) { // Substructure expansion
      Join instance_(j-1) with oneedge table to generate instance_j
      Eliminate pseudo duplicates from instance_j table
        Group instance_j by vertex label and edge label to obtain substructures and its
     count
      Calculate DMDL for substructures and insert them into sub_fold_j table
        Retain top Beam substructures in sub_fold_j table and remove  the rest
      Retain only the instances of sub_fold_j table in instance_j table and eliminate
     others
    }
    --- End of iteration i ---
   Select the best substructure of iteration i
   Compress the substructure by updating vertex table and oneedge table
  }
End HDB-Subdue
```

Representation of a Substructure



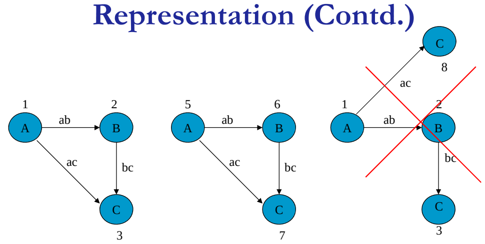
Representation (Contd.)

3 edge instances



Isomorphic instance Count 3

Representation (Contd.)



EDB instance_3 (instances of size 3)

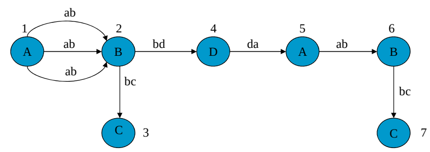
V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
1	2	3	0	A	B	C	-	ab	bc	ac	1	2	2	3	1	3
5	6	7	0	A	B	C	-	ab	bc	ac	1	2	2	3	1	3
1	2	3	8	A	B	C	C	ab	bc	ac	1	2	2	3	1	4

Count 3

Representation

- To represent an n-edge substructure, we need 6n+2 attributes
  - n+1 vertex numbers
  - n+1 vertex labels
  - n edge numbers
  - n edge labels, and
  - n from/to pairs
- Given that databases support about 1000 attributes now, the max substructure that can be represented is about 165 edges

Need for Edge numbers



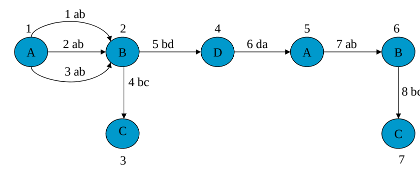
vertex1	vertex2	edge1	vertex1name	vertex2name
1	2	ab	A	B
1	2	ab	A	B
1	2	ab	A	B
.	.	.	.	.

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

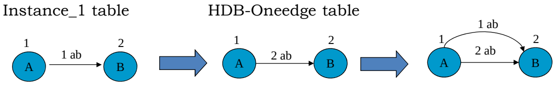
Constrained Expansion




```
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
    )
```


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

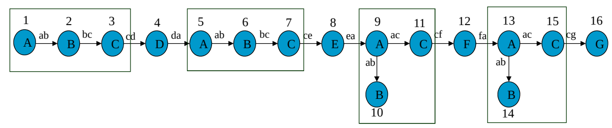
- To expand a 1-edge substructure with 2 vertices V1 and V2, we need to:
  - expand self-edges on V1 or V2 (in general n)
  - expand multiple edge from v2 to v1 (or n*(n-1))
  - expand incoming edge on V1 or V2 (or n)
  - expand outgoing edge on V1 or V2 (or n)
- In general, to expand a substructure of size n, we need n**2 + 2n queries
- These queries are generated (can be one union query)
- If you know that multiple edges and/or cycles do not exist, less number of queries can be used

Expansion

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2	COUNT	DMDL
A	B	C	AB	AC	1	2	1	3	3	1.8
A	B	D	AB	BD	1	2	2	3	1	0.9
B	D	A	BD	DA	1	2	2	3	1	0.9
D	A	B	DA	AB	1	2	2	3	1	0.9
D	A	C	DA	AC	1	2	2	3	1	0.9

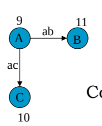
Substructure Evaluation metric

Example

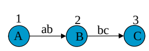


2 Edge substructure instances

Subdue MDL



Count = 2



Count = 2

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

	9	10	11
9	0	1	1
10	0	0	0
11	0	0	0

MDL = 1.14539

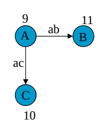
Adjacency matrix

	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0

MDL = 1.12849

Adjacency matrix

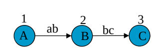
DMDL



count = 2
sub_vertices = 3
non-zero rows = 1

	9	10	11
9	0	1	1
10	0	0	0
11	0	0	0

DMDL = 1.1481



count = 2
sub_vertices = 3
non-zero rows = 2

	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0

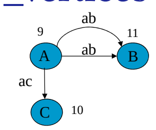
DMDL = 1.1071

$$\text{DMDL} = \frac{\text{Value (G)}}{\text{Value (S)} + \text{Value (G|S)}}$$

- Value(G) = graph_vertices + graph_edges
- Value(S) = sub_vertices + non-zero rows
- Value(G|S) = (graph_vertices – sub_vertices * count + count) + (graph_edges – sub_edges * count)

sub_vertices

Three edge



EDB – Subdue:

No of vertices = No of edges + 1

No of ~~vertices~~ = 3 + 1 = 4

V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
9	**11**	**10**	**0**	A	B	-	-	AB	AB	AC	**1**	**2**	**1**	**2**	**1**	**3**
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Instance_3 table

Repeating vertex marked by vertex invariant (0's and –'s)

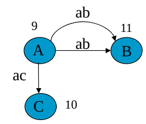
No of vertices = count of non zero vertex numbers = 3

OR

No of vertices = count of unique connectivity attributes = 3

non-zero rows

Three edge



	9	10	11
9	0	1	1
10	0	0	0
11	0	0	0

Adjacency matrix

Non Zero Rows = 1

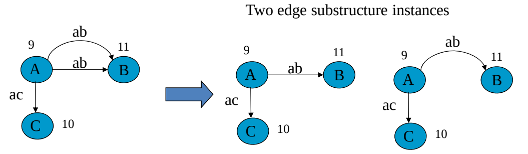
Instance_3 table

V1	V2	V3	V3	V1L	V2L	V3L	V3L	E1	E2	E3	F1	T1	F2	T2	F3	T3
9	11	10	0	A	B	-	-	ab	ab	ac	**1**	2	**1**	2	**1**	3
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

- Non zero row in adjacency matrix will have out degree of at least 1
- The Fn attribute represents outgoing vertex
- Count of unique Fn attributes gives number of non-zero rows

count

Two edge substructure instances



Count = 2

- Value(G|S) = (graph_vertices – sub_vertices * count + count) + (graph_edges – sub_edges * count)
  = (3 – 3*2 + 2) + (3 – 2*2) = -2
  = (3 – 3*1 + 1) + (3 – 2*1) = 2
- Tells how many vertices of the entire graph, the substructure can compress
- Both the instances remove same set of vertices, therefore count = 1

count (Contd..)

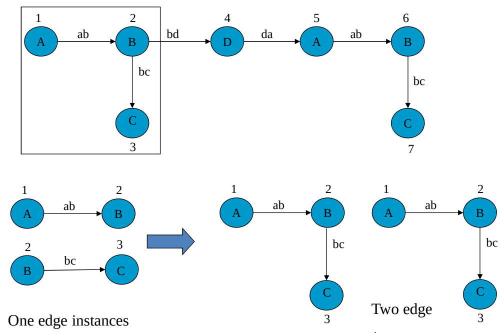
Instance_2 table

V1	V2	V3	V1L	V2L	V3L	E1	E2	F1	T1	F2	T2
9	11	10	A	B	C	ab	ac	1	2	1	3
9	11	10	A	B	C	ab	ac	1	2	1	3
.	.	.	.	.	.	.	.	.	.	.	.

Count = 2
1 Group with 2 instances

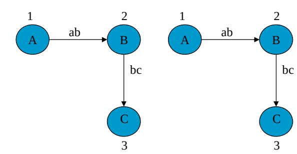
- GROUP BY vertex, edge label and connectivity attributes gives count of all instances, i.e. 2
- GROUP BY vertex number, within instances of a particular substructure will return group of instances containing unique vertex numbers
- Count on the number of such groups gives the updated count of instances. In this case it is 1
- Therefore Instead of counting 2, we count as 1

Pseudo duplicates



Pseudo duplicates (Contd..)

Two edge



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
~~1~~	~~2~~	~~3~~	~~B~~	~~C~~	~~A~~	~~BC~~	~~AB~~	~~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
.	.	.

Unsorted_E

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

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
.	.	.

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
.	.	.	.	.	.	.	.	.	.	.	.

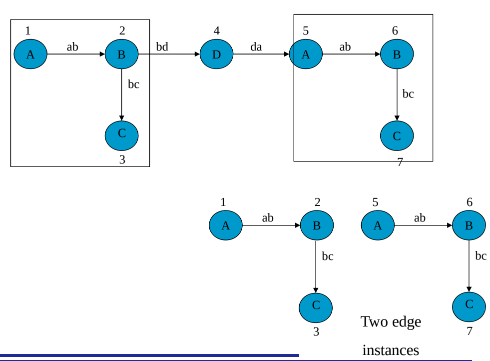
Pseudo duplicates (Contd..)

- 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

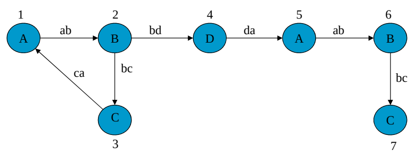
Canonical label ordering

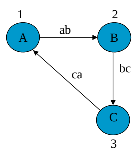


Canonical label ordering

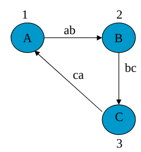
- These two are NOT duplicates
- They need to be recognized as isomorphic to each other
- In order to do this, vertices and labels need to be canonically ordered
- This process for canonical ordering is similar to the process used for psuedo duplicate elimination

Cycles – Marking repeated vertex



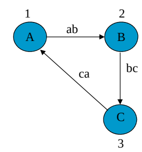


Three edge starting with ab



Three edge starting with bc

Cycles – Marking repeated vertex (Contd..)



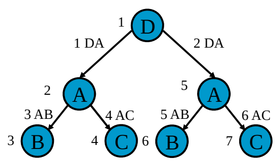
V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
1	2	3	1	A	B	C	A	AB	BC	CA	1	2	2	3	3	4
2	3	1	2	B	C	A	B	BC	CA	AB	1	2	2	3	3	4

Instance_3 table

V1	V2	V3	V4	V1L	V2L	V3L	V4L	E1	E2	E3	F1	T1	F2	T2	F3	T3
0	1	2	0	A	B	C	-	AB	BC	CA	1	2	2	3	3	1
~~0~~	~~1~~	~~2~~	~~0~~	~~B~~	~~C~~	~~A~~	~~-~~	~~BC~~	~~CA~~	~~AB~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	~~1~~

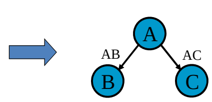
Instance_3 table with vertex invariants 0’s and –’s

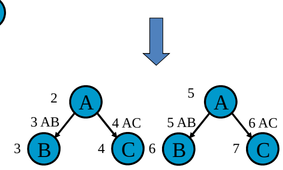
Hierarchical Reduction



Input graph

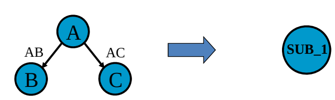
Best substructure





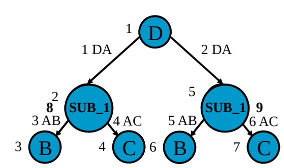
Best substructure instances

Hierarchical Reduction (Contd..)



Best substructure

Compressing vertex



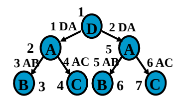
Input graph

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

Selecting the best substructure

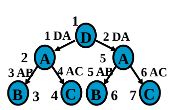


subfold_2

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

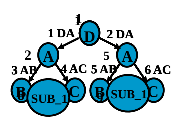
Best substructure is one with highest DMDL value

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

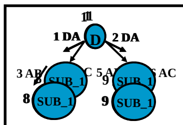
```
DELETE FROM VertexTable
WHERE VertexNo IN (
        (SELECT V1 FROM BestInstances)
UNION (SELECT V2 FROM BestInstances)
                    . . . .
UNION (SELECT Vn+1 FROM BestInstances))
```

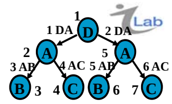
Vertex table

V	VL
1	D
~~8~~	~~A~~
~~9~~	~~B~~
~~4~~	~~C~~
5	A
6	B
7	C

VL
D
SUB_1
SUB_1

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

Oneedge table

EL	EN	V1L	V2L	V1	V2
DA	1	D	SUB_1	1	8
DA	2	D	SUB_1	1	9
~~AB~~	~~3~~	~~A~~	~~B~~	~~2~~	~~3~~
~~AC~~	~~4~~	~~A~~	~~C~~	~~2~~	~~4~~
~~AB~~	~~5~~	~~A~~	~~B~~	~~5~~	~~6~~
~~AC~~	~~6~~	~~A~~	~~C~~	~~5~~	~~7~~

```
DELETE FROM oneedge
WHERE EdgeNo IN (
    (SELECT E1N FROM BestInstances)
UNION (SELECT E2N FROM BestInstances)
    ....
UNION (SELECT EnN FROM BestInstances))
```

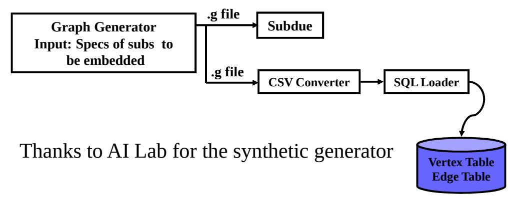
```
UPDATE oneedge set V1=MAX_VERTEX
WHERE V1 IN (
    (SELECT V1 FROM BestInstances WHERE rownum = 1)
 UNION (SELECT Vn+1 FROM BestInstances WHERE rownum = 1))
```

```
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))
```


Experimental Results

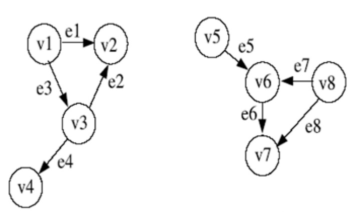
Setup:

- 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



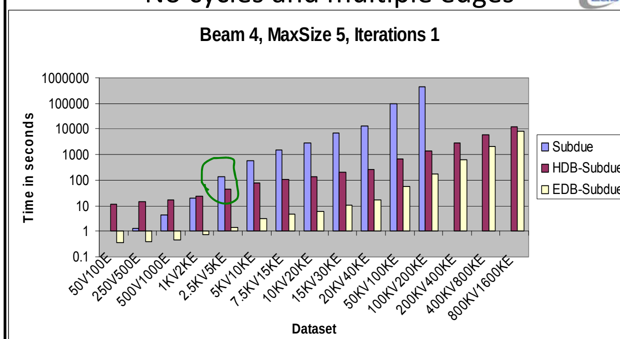
Thanks to AI Lab for the synthetic generator

No cycles and multiple edges



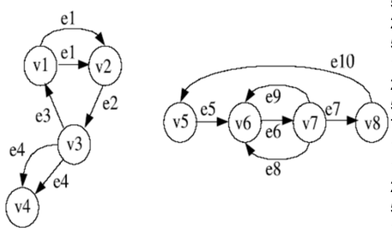
Dataset	Instances
50V100E	4
250V500E	15
500V1000E	30
1KV2KE	60
2.5KV5KE	150
5KV10KE	300
7.5KV15KE	450
10KV20KE	600
15KV30KE	900
20KV40KE	1200
50KV100KE	3000
100KV200KE	6000
200KV400KE	12000
400KV800KE	24000
800KV1600KE	48000

No cycles and multiple edges



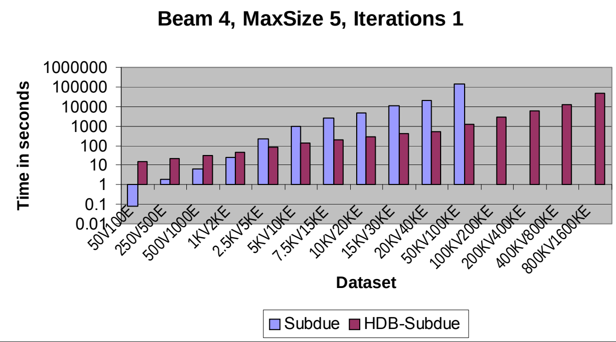
- Running time grows exponentially with graph input size
- Subdue crossover – 2.5KV5KE

With cycles and multiple edges



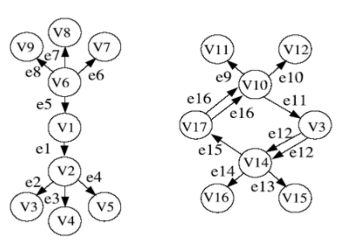
Dataset	Instances
50V100E	4
250V500E	15
500V1000E	30
1KV2KE	60
2.5KV5KE	150
5KV10KE	300
7.5KV15KE	450
10KV20KE	600
15KV30KE	900
20KV40KE	1200
50KV100KE	3000
100KV200KE	6000
200KV400KE	12000
400KV800KE	24000
800KV1600KE	48000

With cycles and multiple edges



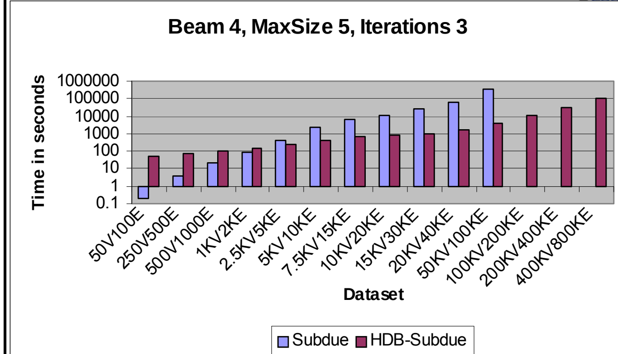
- Subdue crossover – 2.5KV5KE

Hierarchical Reduction



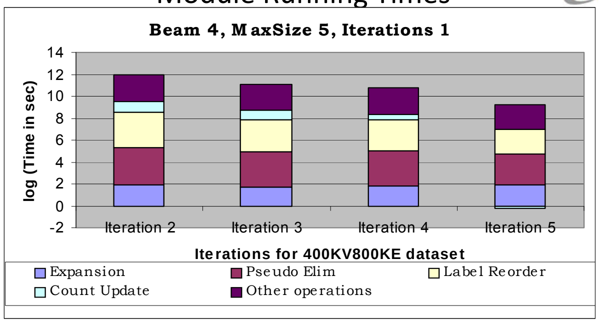
Dataset	Instances
50V100E	2
250V500E	10
500V1000E	20
1KV2KE	40
2.5KV5KE	80
5KV10KE	160
7.5KV15KE	250
10KV20KE	320
15KV30KE	420
20KV40KE	640
50KV100KE	1280
100KV200KE	2560
200KV400KE	5120
400KV800KE	10240
800KV1600KE	20480

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

Conclusion

- Captured complete graph information using new schema
- Unconstrained expansion can expand multiple edges and a scheme for eliminating pseudo duplicates was provided
- DMDL has been modified to account for multiple edges
- Handles Cycles in the input graph
- Performs hierarchical graph reduction
- Tested to mine input graphs of size 800KV1600KE

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

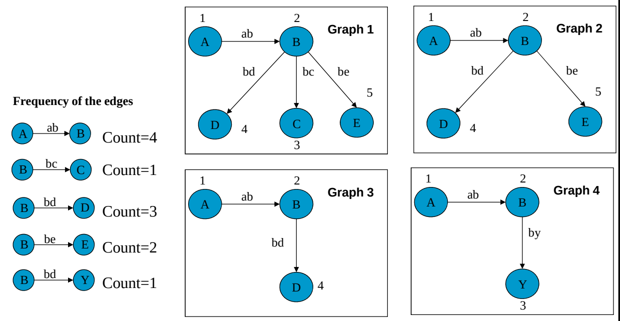
Vertex no	Vertex label	gid
1	a	1

Edges

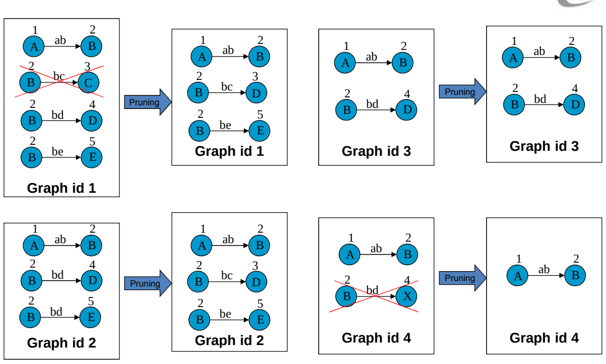
Vertex1	Vertex 2	Edge label	gid
1	2	ab	1

Overview of DB-FSG

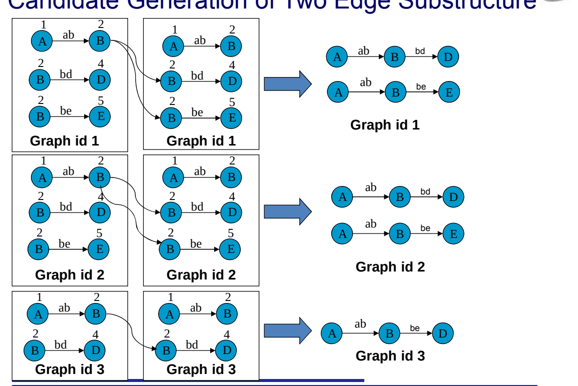
Input parameters support = 50% and size = 3



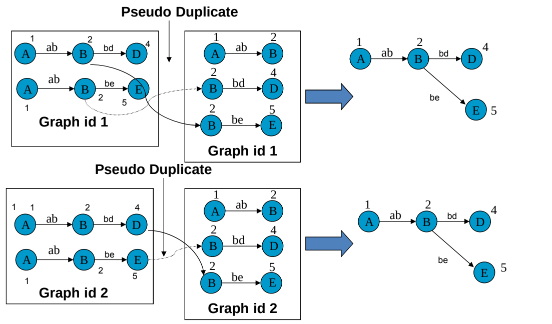
Candidate Generation of One Edge Substructure



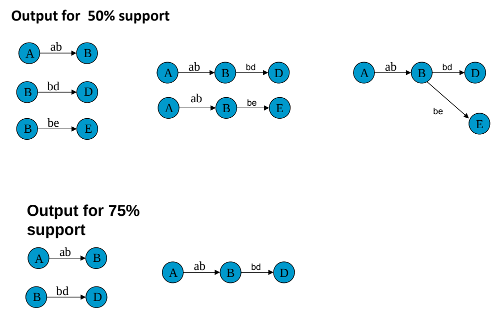
Candidate Generation of Two Edge Substructure



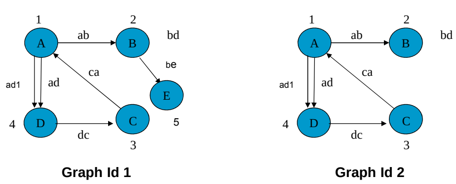
Candidate Generation of Three Edge Substructure



Output



Representation of Instances of Substructures

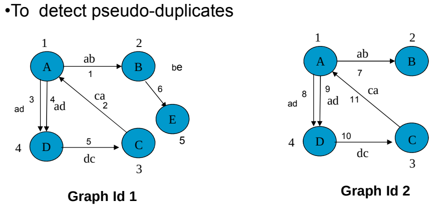


Oneedge Table

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

Need for Edge Numbers

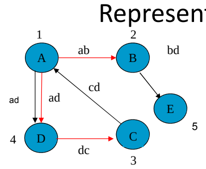
- To avoid redundant expansion on edges that already exists in the instance
- To detect pseudo-duplicates



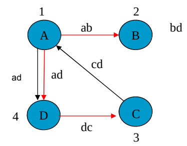
Oneedge Table

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

Representation of Instances (cont…)



Graph Id 1

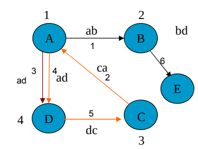


Graph Id 2

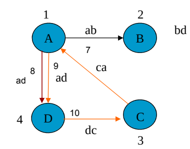
Instance_3 Table

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
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

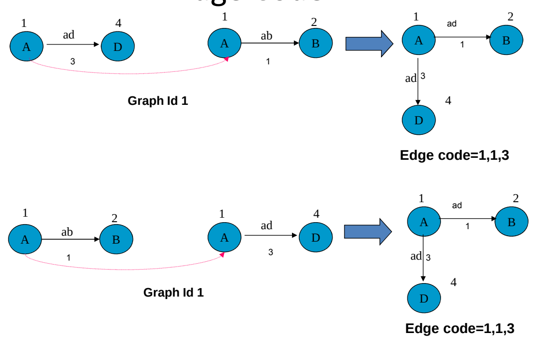
Detecting Pseudo duplicates using Edge Code

- 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

- 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

Detecting Pseudo duplicates using Edge Code



Pseudo Duplicates

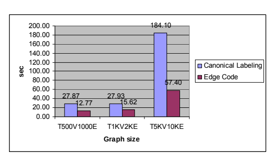
- HDB-Subdue uses canonical ordering on vertex number for elimination of pseudo duplicates.
- Canonical ordering requires maintaining six intermediate tables, sorting of two intermediate tables, one 3 way join and one 6n+2 way join.
- Canonical ordering needs to be done as in HDB-Subdue as edge numbers cannot be use for substructure isomorphism

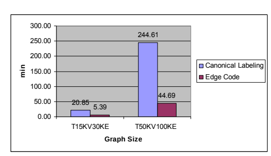
Processing Time of Edge Code Approach VS Canonical Ordering

- Max Substructure size 5

Graph Size	Canonical Ordering on Vertex No	Edge Code	Improvement
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



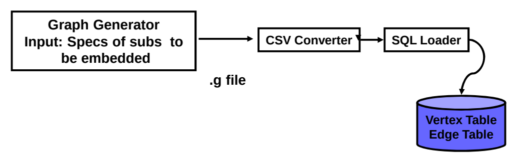


Frequency Counting and Substructure Pruning

- 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.

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



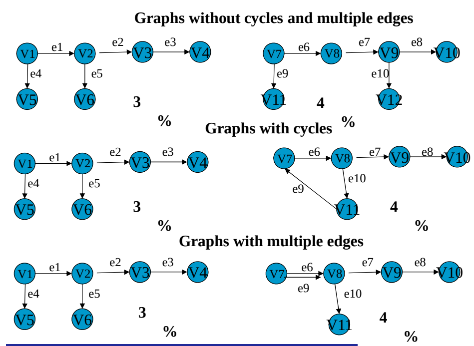
Experiment Dataset DB-FSG

- 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%

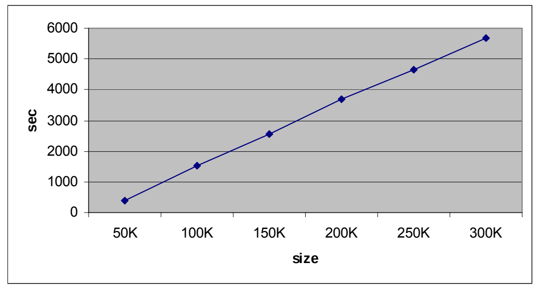
Experiment Dataset DB-FSG

- 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%

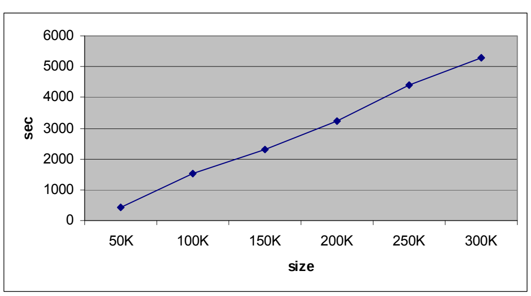
Embedded Graphs



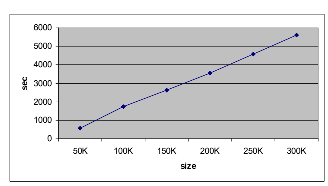
Graphs without cycles and multiple edges



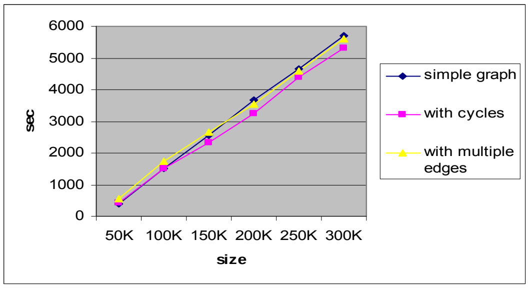
Graphs with cycles



Graphs with Multiple Edges

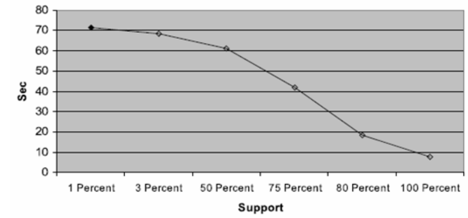


Experiment results



Processing time Vs Support

- # Graph 1K
- Substructures with support of
- 1%, 3%, 5%, 50%, 75%, 80%, 100%



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 Letters*, 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.