HDB-SUBDUE, A RELATIONAL DATABASE APPROACH TO GRAPH MINING AND HIERARCHICAL REDUCTION

by

SRIHARI PADMANABHAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

To my Family and Friends.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his constant guidance and support, and for giving me a wonderful opportunity to work on this topic. I am grateful to Dr. Gautam Das and Dr. Diane Cook for serving on my defense committee.

I would like to thank Raman Adaikalavan, Ramanathan Balachandran and Manu Aery for helping me throughout in times of need. I would also like to take time to thank Mr Patrick McGuigan for providing access and support to the DPCC Cluster machines. Thanks are due to my seniors at ITLab, Ajay Eppili, Vamshi Pajjuri, Shravan Chamakura and Dustin for their help and support. I would like to thank my friends Vihang, Dhawal Bhatia, Sunit, Nikhil, Lokendra, Subhesh and my roommates, Sandeep and Kalyan for their cooperation and support. I would also like to thank my parents and brother for their constant love and support throughout my academic career.

This work was supported, in part, by NSF (grants IIS-0097517, IIS-0326505, and EIA-0216500.)

November 18, 2005

ABSTRACT

HDB-SUBDUE, A RELATIONAL DATABASE APPROACH TO GRAPH MINING AND HIERARCHICAL REDUCTION

Publication No. _____

SRIHARI PADMANABHAN, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Sharma Chakravarthy

Data mining aims at discovering interesting and previously unknown patterns from data sets. Transactional mining (association rules, decision trees etc.) can be effectively used to find non-trivial patterns in categorical and unstructured data. For applications that have an inherent structure (e.g., chemical compounds, proteins) graph mining is appropriate, because mapping the structured data into other representations would lead to loss of structure. The need for mining structured data has increased in the past few years. Graph mining uses graph theory principles to perform mining. Database mining of graphs aims at mining structured graph data stored in relational database tables using SQL queries. Various kinds of data such as Social network data, Protein, and other Bioinformatics data can be effectively represented as graphs. Graph mining has been successful in the areas of counter terrorism analysis, credit card fraud detection, drug discovery in pharmaceutical industry etc.

The focus of this thesis is to apply relational database techniques to accommodate all aspects of graph mining. Our primary goal is to address scalability of graph mining

to very large data sets, not currently addressed by main memory approaches. This thesis addressed the most general graph representation including multiple edges between any two vertices, and cycles. This thesis extends previous work (EDB-subdue) in a number of ways: improved substructure representation to avoid false positives during frequency counting, unconstrained substructure expansion with pseudo duplicate elimination for expanding multiple edges, canonical ordering of substructures for getting true count, hierarchical reduction for producing abstract pattern and generalization of DMDL that includes the presence of multiple edges in a subgraph. We also extend the substructure pruning to include ties when selecting top beam substructures.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the advent of automated data collection tools, massive amounts of data are being generated. As the cost per gigabyte is diminishing, the data being stored is increasing. It is important to extract useful knowledge from these stored data to aid in decision-making. Data mining tasks help in discovering non-trivial patterns that are difficult to find manually. Data mining is one of the steps in Knowledge Discovery in Databases (KDD). The steps in KDD are

1. Data collection,
2. Data cleaning and transformation,
3. Choosing the data mining function (summarization, classification, regression, association, clustering)
4. Data Mining,
5. Visualization of results.

Data mining has been a topic of research for quite some time [1, 2, 3] with much of the work on discovering association rules from transactional data represented as (binary) relations. Transactional data mining is widely used in detecting patterns from unstructured data. A popular example is the market basket analysis. For applications that have an inherent structure (e.g., chemical compounds, proteins, social networks) graph mining is appropriate as compared to other techniques as mapping them into other representations would lose the inherent structure. We employ the principles of graph theory to achieve graph mining. The ability to mine over graphs is important, as graphs are capable of representing complex relationships. Graph mining uses the natural structure

of the application domain and mines directly over that structure (unlike others where the problem has to be mapped to transactions or other representations.) Several graph mining approaches are introduced in this chapter.

Subdue [4] is a mining approach that works on graph representation. Subdue identifies concepts describing interesting and repetitive substructures within the structural data. Subdue uses the principle of minimum description length (or MDL) to evaluate the substructures. The minimum description length principle [5] states that the best theory to describe a set of data is a theory that minimizes the description length of the whole data set. Subdue constructs the whole graph and stores it in the form of an adjacency matrix in main memory and then mines by iteratively expanding each vertex into larger subgraphs. Main memory data mining algorithms typically face two problems with respect to scalability. Graphs could be larger than the main memory available and hence cannot be loaded into main memory or the algorithm could be computationally expensive and the space required for computation is more than the available main memory.

This has motivated the development of graph mining algorithms using SQL and stored procedures using Relational database management systems (RDBMSs). Representation of a graph, generation of larger subgraphs, checking for exact and inexact matches of subgraphs are not straightforward in SQL. The input (which is a graph) has to be represented using relations and operations have to use joins (or other relational operations) for mining repetitive substructures. DBMS version of Subdue (DB-Subdue [6, 7]) was developed for the DB2 database. It explains the mapping of substructures to tuples in the database. DB-Subdue was the first attempt to support graph mining using an RDBMS. The goal was to demonstrate the feasibility and scalability of the approach and the test results confirmed that it can easily scale to very large graphs (largest tested graph had 800,000 vertices and 1600,000 edges).

Although DB-Subdue solved the scalability problem, it did not deal with a general form of a graph which can have multiple edges between any two vertices, and cycles as well. It also did not have the equivalent of MDL but used count (or frequency) for determining the best substructure. Only exact matches (of subgraphs) were identified. The frequency metric used for substructure evaluation does not distinguish between two overlapping substructures having the same number of edges and vertices and the same number of instances. Host variables are used for exchanging data (extracted using cursors) between the database and the host programming language. DB2 does not support declaring an array of host variables and hence the generalization for the algorithm could not be achieved. Separate host variables had to be declared for each pass. The DB-Subdue algorithm handles only exact matches between subgraphs. In most of the real-life applications, such as discovering chemical compounds or other domains, there are very few scenarios in which exact graph matches are suitable.

EDB-Subdue [6, 8] extends DB-Subdue and it overcomes some of the limitations of DB-Subdue. EDB-Subdue achieved generalization of the algorithm by using Oracle database which supports the declaration of array of host variables. It also introduced DMDL (Database Minimum Description Length), which follows the trend of Subdue MDL and discriminates substructures with same signature. DB-Subdue does not have the capability to handle cycles in the input graph. EDB-Subdue can detect cycles and it attaches an arbitrary number to the vertex that creates the cycle, to avoid repeated expansions on the vertex that creates the cycle. It also partially explored inexact graph matching in C code using cursors. EDB-Subdue constrains the substructure expansion to avoid duplicate substructure instance generation, and in that process it fails to expand multiple edges between vertices. It uses extension concept to represent direction of edges. Since extensions did not have terminating vertex information of edges, false positives were

introduced during frequency counting. EDB-Subdue also did not perform hierarchical reduction.

The focus of this thesis is to handle multiple edges, cycles, and hierarchical reduction to deal with a general graph. In this thesis we use unconstrained substructure expansion with duplicate elimination, to explore all possible expansions and expand multiple edges. In HDB-Subdue we introduce connectivity attributes which have complete information about the edges with vertex number invariant and connectivity information. Cursors were used in EDB-Subdue for selecting user specified number (beam) of interesting substructures in each iteration. We also avoid the use of cursors and instead use SQL based Analytic functions to implement the beam.

The rest of this thesis is organized as follows. CHAPTER 2 discusses the background and related work in the field of graph-based data mining and the various approaches used for mining structural data. CHAPTER 3 provides the overview of HDB-Subdue. CHAPTER 4 discusses the design issues for the new functionalities and enhancements that have been added to EDB-Subdue. CHAPTER 5 presents implementation details of all the issues that are explained in the design chapter. CHAPTER 6 presents performance evaluation including comparison with the Subdue algorithm. CHAPTER 7 concludes the thesis and identifies potential future work.

CHAPTER 2

RELATED WORK

During the past decade, the field of data mining has emerged as a novel field of research, investigating interesting research issues and developing challenging real-life applications [1]. In this section we briefly describe some of the ongoing research activity in the area of graph mining. Some of the works are Subdue, FSG (Frequent SubGraph discovery), gSpan (graph-based Substructure pattern mining), AGM (Apriori-based Graph Mining), DB-Subdue and EDB-Subdue.

2.1 Subdue

Subdue is one of the earliest works in graph based data mining. It uses adjacency matrix for representing graphs. Apart from doing exact graph match, Subdue [4] has the capability of matching two graphs, differing by the number of vertices specified by the threshold parameter, inexactly. Subdue also has the ability to handle multiple edges between vertices and also handles cycles in the input graph.

2.1.1 Structural Data Representation

The substructure discovery system represents data as a labeled graph. Objects in the data are represented either as vertices or as small subgraphs and the relationships between them are represented as edges. A substructure is a connected subgraph within a graph. An instance of the substructure in the graph is a set of vertices and edges that have the same vertex labels, edge labels and edge directions as that of the substructure.

An edge can be either directed or undirected. The substructures are evaluated based on a metric called Minimum Description Length [9] principle based on adjacency matrices.

2.1.2 Supervised Learning

The Subdue Concept Learner (SubdueCL) [10] is an extension to the Subdue system aimed at supervised pattern discovery. Positive and negative examples of a phenomenon being available for input, SubdueCL searches for a pattern that compresses the positive graphs, but not the negative graphs. For example, given positive graphs describing criminal networks and negative graphs describing normal social networks, Subdue can learn patterns distinguishing the two, and these patterns can be used as a predictive model to identify emerging criminal networks.

2.1.3 Hierarchical Reduction

Cluster analysis [11] or simply clustering, is a data mining technique often used to identify various groupings or taxonomies in real world databases. The purpose of applying clustering to a database is to gain a better understanding of the data, in many cases by highlighting hierarchical topologies. An example of a hierarchical clustering is the classification of vehicles into groups such as cars, trucks, motorcycles, tricycles, and so on, which are then further subdivided into smaller groups based on observed traits. The SUBDUE algorithm compresses the input graph with the best substructure that it discovers in each iteration. The compression is carried on till the user specified limit is reached or when there are no more substructures to compress.

Cobweb [12] and Labyrinth [13] are also some of the clustering approaches.

2.1.4 Inexact Graph match

Though exact graph match comparison discovers interesting substructures, most of the substructures in the graph may be slight variants of another substructure. In order to detect these, the algorithm developed by Bunke and Allerman [14] is used where each distortion is assigned a cost. A distortion is defined as the addition, deletion or substitution of vertices or edges. The two graphs are said to be isomorphic as long as the distortion cost difference falls within the user specified threshold. This algorithm is an exponential algorithm as it compares each vertex with every other vertex in the graph. The branch and bound approach used by Subdue makes the algorithm computationally bound as the number of mappings considered are quite less when compared to all the possible mappings.

2.2 FSG - Frequent Subgraphs

FSG [15] uses graphs to model frequent itemset mining. For example it converts a basket of items into vertices connecting every vertex to every other vertex by an edge. Subgraphs that occur frequently over a large number of baskets will form patterns which include frequent itemsets in the traditional sense. In simple terms FSG can be stated as follows. Given a graph data set $G = \{G_1, G_2, G_3, \cdots\}$ the problem is to discover subgraph(s) that occur in most of the graphs in G.

The key features of FSG are:

1. Uses a sparse graph representation
2. Increases the size of frequent subgraphs by adding one edge at a time
3. Uses canonical labeling and graph isomorphism and
4. Involves candidate generation and frequency counting
5. Restricts discovery by finding only the subgraphs that are connected

The kernel of frequent subgraph mining is candidate generation and subgraph isomorphism test. FSG initially enumerates all the frequent single and double edge graphs. During each iteration it first generates candidate subgraphs whose size is greater than the previous frequent ones by one edge. Next, it counts the frequency for each of these candidates, and prunes subgraphs that do no satisfy the support constraint. Discovered frequent subgraphs satisfy the downward closure property of the support condition which is used for pruning the lattice of frequent subgraphs. Subgraph isomorphism is achieved using canonical labeling.

2.2.1 Canonical labeling

Canonical labeling is performed to get a total order of graphs. A canonical label is a unique code of a given graph [16, 17]. Canonical labels of two graphs should always be same, as long as the graphs have the same topological structure and the same labeling of edges and vertices. Computing canonical labels is equivalent to determining isomorphism between graphs, because if two graphs are isomorphic with each other, their canonical labels must be identical. The canonical label is generated by using a flattened representation of the adjacency matrix of a graph. To compute a canonical label of a graph, all the permutations of its vertices are tried to see which order of vertices gives the minimum adjacency matrix. To narrow down the search space, the vertices are partitioned by their degrees and labels using a technique called vertex invariants [16]. Then all possible permutations of vertices inside each partition are generated.

The vertex invariant partitioning for Fig 2.1 is shown in tables Tab 2.1 and Tab 2.2. The matrix on the left gives a label of $000e_1e_0e_0$, while the right one has a label of $000e_0e_1e_0$. Because e0 < e1 and $000e_0e_1e_0 < 000e_1e_0e_0$ by string comparison, the label of the right matrix becomes the canonical label.

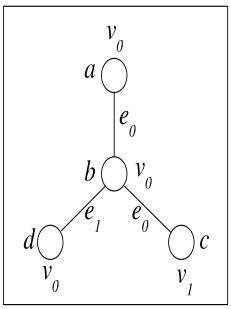

Figure 2.1 Frequent Subgraphs

Table 2.1 Canonical Label

id	d		a	c	b
label	v_0		v_0	v_1	v_0
partition		0		1	2
d	0		0	0	e_1
a	0		0	0	e_0
c	0		0	0	e_0
b	e_0		e_1	e_0	0

Table 2.2 Canonical Label

id	a		d	c	b
label	v_0		v_0	v_1	v_0
partition		0		1	2
a	0		0	0	e_0
d	0		0	0	e_1
c	0		0	0	e_0
b	e_0		e_1	e_0	0

2.3 gSpan - graph based Substructure pattern mining

gSpan [18] is one of the approaches that perform frequent graph based data mining like FSG.

The challenges met by Apriori-like algorithms are:

1. Candidate generation: the generation of size *k+1* subgraph candidates from size k frequent subgraphs is more complicated and costlier than that of itemsets; and
2. Pruning false positives: subgraph isomorphism test is an NP-complete problem, thus pruning false positives is expensive.

Unlike FSG, gSpan discovers frequent substructures without candidate generation and false positives pruning. It builds a 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.

gSpan is a main memory algorithm and if the entire graph does not fit in main memory, graph-based data projection as in PrefixSpan [19] is applied and then gSpan is performed.

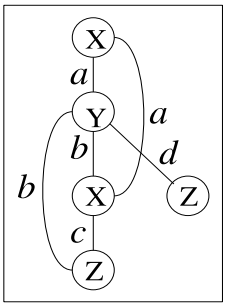

Figure 2.2 gSpan

2.3.1 DFS Coding

A Depth First Search is conducted on a graph to generate DFS trees. Multiple DFS trees can exist for a graph because the root vertex can be any arbitrary vertex in the graph. A DFS code for any tree is a set of 5-tuples (i,j,l_i,$l_{(i,j)}$,l_j), where l_i and l_j are labels of vertex i and vertex j respectively and $l_{(i,j)}$ is the label of the edge between them. Tab 2.3 shows the corresponding DFS codes for Fig 2.2. Since a graph can have more than one DFS tree and hence more than one DFS code we select the Minimum DFS code as the one that occurs earliest in the lexicographic order and designate it as the canonical label for the graph. Thus the problem of mining frequent connected subgraphs is equivalent to mining their corresponding minimum DFS codes. gSpan was shown to perform better than FSG because it combines the growing and the isomorphism checking into a single phase.

Table 2.3 DFS code

Edge	5 Tuple
0	(0,1,X,a,Y)
1	(1,2,Y,b,X)
2	(2,0,X,a,X)
3	(2,3,X,c,Z)
4	(3,1,Z,b,Y)
5	(1,4,Y,d,Z)

2.4 AGM - Apriori based Graph Mining

Apriori based Graph Mining approach [20, 1] mines induced frequent subgraphs in graph structured transactions. Similar to Market-Basket Analysis, the interestingness of a subgraph is defined along the support and the confidence of the Apriori algorithm. The graph represented by using an adjacency matrix can only represent a graph which includes labeled nodes and is not suitable to represent labeled links and self-loop links. A preprocessing is applied to convert the general graph transaction data to the graphs having labeled nodes, unlabeled links and no self-loop links. The rows and the columns corresponding to the graph nodes in the adjacency matrix are lexicographically ordered in terms of the node labels to reduce the ambiguity of graph representation. A code, defined by the non-diagonal elements of the adjacency matrix is taken as the canonical code for a graph. Once the graphs are represented by the codes of their adjacency matrices, the codes of the frequent induced subgraphs are derived in A bottom up manner similar to Apriori algorithm. An induced subgraph is defined by the following definition.

$$V(G') \subseteq V(G), E(G') \subseteq E(G) \; and \tag{2.1}$$

$$\forall u, v \in V(G') u, v \subseteq E(G) \Leftrightarrow u, v \subseteq E(G') \tag{2.2}$$

where V(G) is the set of all nodes in the graph G, and E(G) is the set of all links u, v connecting u and v in the graph G, where u, v $\subseteq$ V(G). G' is an induced subgraph of G, if G' has a subset of the nodes of G and all the links between pairs of nodes as in G. The graph whose frequency exceeds the minimum support is called as a 'frequent induced subgraph'.

2.4.1 Candidate Generation

Two frequent induced subgraphs G(X_k) and G(Y_k) are joined to generate a candidate frequent induced subgraph Z_{k+1} of size k+1. That is, induced k+1-subgraph is generated, if both G(X_k) and G(Y_k) have the same elements in the matrices except for the elements of the k-th row and the k-th column. Frequency of each candidate frequent induced subgraph is counted by scanning the database after generating all the candidates of frequent induced subgraphs and obtaining their canonical forms. Trie data structure is used in implementing frequency counting.

2.5 DB-Subdue

The downside in Subdue main memory algorithm is its scalability to large problems. This has motivated the development of graph mining algorithms using SQL and stored procedures using Relational database management systems (RDBMSs). The graph input is represented using relations, and operations use joins (or other relational operations) for mining repetitive substructures. At the same time, it is necessary that the Relational approaches avoid manipulations that are known to be inefficient (e.g., correlated subqueries, cursors on large relations, in-place updates and deletes from a relation). The DBMS version of Subdue (DB-Subdue) [6, 7] was developed for the DB2 database using the C language. The experiments show that it can mine graphs with millions of vertices and edges with sub-linear growth in the computation time. DB-Subdue has several ap-

proaches for graph mining, such as the cursor-based approach, the UDF based approach and the enhanced cursor-based approach. The enhanced cursor-based approach, which was the last approach implemented, proved that the DB-Subdue scales better for larger graphs as compared to the main memory version. DB-Subdue started to perform better than Subdue for input graphs with more than 500 vertices and 1000 edges. It uses pure SQL statements and indexing techniques to improve the performance of the algorithm. DB-Subdue was able to mine data sets with 800K vertices and 1600K edges.

2.6 EDB-Subdue

EDB-Subdue [8, 6] is an extension of DB-Subdue and was developed in an effort to handle certain aspects of graph input like cycles, overlap and inexact graph match which were not considered in DB-Subdue. DB-Subdue uses frequency as the substructure evaluation metric. Frequency does not differentiate substructures having same signature (having same number of vertices and frequency count). Hence, some of the the best substructures may not be detected because of pruning when the beam is applied based on the count. EDB-Subdue overcomes this limitation by using a substructure evaluation metric called DMDL (Database Minimum Description Length).

EDB-Subdue uses cursors to implement the beam for limiting the number of substructures considered for the next pass. Host variables are declared for exchanging data (extracted using cursors) between the database and the host programming language. Since DB2 does not support declaring an array of host variables and hence the generalization of DB-Subdue algorithm could not be achieved. Separate host variables and queries had to be declared for each pass.

EDB-Subdue is implemented using Oracle DBMS. The advantage with the Oracle database is that it supports declaration of array host variables and hence the generalization can be achieved without having to declare separate host variables for each pass.

This is an improvement over using the DB2 database. EDB-Subdue developed a formula for evaluating the substructures (termed DMDL) which is able to distinguish the best substructure among substructures of equal edge length and frequency. Also it handles inexact graph matching by using cursors and invoking the C subroutine that compares 2 subgraphs for a threshold and return a boolean value. The cursor usage poses some problems when the size of the graph is very large as this approach involves N^2 comparisons.

Tab 2.4 shown below, gives a summary of the approaches discussed above and their capabilities. We can see that all the main memory approaches have a limitation on the largest dataset that they can handle, but they offer more capabilities than the approached based on database. The database approaches are catching up with the main memory counterparts in terms of the capabilities.

Table 2.4 Summary of Related Work

	Subdue	FSG	AGM	DB-Subdue	EDB-Subdue
Graph Mining	√	√	√	√	√
Multiple edges	√	×	×	×	×
Hierarchical Redn	√	×	×	×	×
Cycles	√	√	√	×	√
Eval Metric	MDL	Frequency	Support	Frequency	DMDL
Inex graph match	√	×	×	×	Partial
Memory limitation	√	√	√	×	×

CHAPTER 3

OVERVIEW OF HDB-SUBDUE

3.1 Graph Representation

In this chapter we will describe how graphs are represented in a database and expanded one edge at a time. Since databases only have relations, we need to convert the graph into tuples in a relation. The input which is a connected or a disconnected graph is read from a flat file and loaded into the database using SQL Loader. The vertices in the graph are inserted into a relation called Vertices and the edges are inserted into a relation called Edges. For the graph shown in Fig 3.1, the corresponding vertices and the edges relations are shown in Tab 3.1 and Tab 3.2.

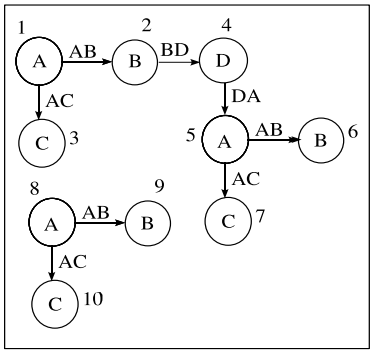

Figure 3.1 Example Graph

From the vertex and edge relation, a new relation called oneedge as shown in Tab 3.3 is created which will contain all instances of substructures of size one. The oneedge relation is created because the edges relation does not contain information about the vertex labels. For a one edge substructure, the edge direction is always from the first

Table 3.1 Vertex table

Vertex No	Vertex Name
1	A
2	B
3	C
4	D
5	A
6	B
7	C
8	A
9	B
10	C

Table 3.2 Edge table

Vertex 1	Vertex 2	Edge Label
1	2	AB
1	3	AC
2	4	BD
4	5	DA
5	6	AB
5	7	AC
8	9	AB
8	10	AC

vertex to the second vertex. This is why there are no attributes in the oneedge table which specify the direction. For a higher edge substructure, we introduce connectivity attributes to denote the direction of edges between the vertices of the substructure. The oneedge table is the base table we would be using for generating higher edge substructures. For each edge in the oneedge table we assign a unique identifier called edge number. We will discuss the usefulness of edge numbers in the design issues chapter. We remove the edges from oneedge, which have only one instance, since these substructures will not expand to have more than one instance in the larger substructures (there may be one instance substructures produced later when repeated substructures of length i grow to substructures of length i + 1). This removal of single instance edges is applicable to exact mach. For inexact matches, single instance edges cannot be removed.

To generate a two edge substructures we join oneedge relation with itself. We term the resulting two edge substructure table as instance_2. In general, substructures of size i are generated by joining instance_(i-1) relation with oneedge relation. Hence, to generate three edge substructures (i.e., instance_3), we would join instance_2 and oneedge relations on the matching vertices. In case of substructures that have 2 or more edges, we would need attributes to denote the direction of the edges. The From and To (F and T for

Table 3.3 Oneedge table

Vertex 1	Vertex 2	Edge No	Edge Label	Vertex1 Name	Vertex2 Name
1	2	1	AB	A	B
1	3	2	AC	A	C
2	4	3	BD	B	D
4	5	4	DA	D	A
5	6	5	AB	A	B
5	7	6	AC	A	C
8	9	7	AB	A	B
8	10	8	AC	A	C

short) attributes in the instance_n table serve this purpose. An n edge substructure is represented by n+1 vertex numbers, n+1 vertex labels, n edge numbers, n edge labels and n From and To pairs. In general, 6n+2 attributes are needed to represent an n-edge substructure. Note that the edge numbers are not part of the input. Edge numbers are assigned internally by the system to distinguish between edges between the same vertices and have the same edge label. Though edge numbers are part of every instance_n table, owing to the space constraint, we will be showing it only in sections where they are necessary. In HDB-Subdue the size denotes the number of edges in the substructure. Instance_2 relation for the graph in Fig 3.1 is shown in Tab 3.4.

Table 3.4 Instance_2

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	F1	T1	F2	T2
1	2	4	A	B	D	1	3	AB	BD	1	2	2	3
1	2	3	A	B	C	1	2	AB	AC	1	2	1	3
2	4	5	B	D	A	3	4	BD	DA	1	2	2	3
4	5	6	D	A	B	4	5	DA	AB	1	2	2	3
5	6	7	A	B	C	5	6	AB	AC	1	2	1	3
4	5	7	D	A	C	4	6	DA	AC	1	2	2	3
8	9	10	A	B	C	7	8	AB	AC	1	2	1	3

Table 3.5 Sub_Fold_2

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

The edge label, which is just a string of text, does not give any information about the vertex it is originating or terminating from. One can get this information from the connectivity attributes Fi and Ti, which tell us that the edge i is originating at Fi and terminating at Ti. Note that the Fi and Ti are relative to the substructure. That is, the values of the attributes Fi and Ti do not indicate the actual vertex numbers but the attributes whose value correspond to the vertex numbers. Let us consider the fourth substructure in the instance_2 table. For edge DA the value of F1 is 1 indicating that the edge originates from the vertex number stored in V1 which is 4 and T1 is 2 meaning it terminates at vertex number stored in V2 which is 5. Hence, the edge DA is between the vertices 4 and 5. Similarly, the edge AB (on the same row) originates from vertex 5 and terminated at vertex 6.

To evaluate a substructure we identify similar substructures and count them. Projection on the vertex labels, edge labels and connectivity attributes in the instance_n table and a GROUP BY on the same attributes will produce the count of each substructure and using the count we calculate the DMDL value of each substructure and insert it into a table called Sub_Fold_n. Sub_Fold_2 is shown in Tab 3.5. We use the term substructures when we refer to the tuples in Sub_Fold_n relation, and instances when referring to the tuples in instance_n. Instances include vertex numbers whereas substructures are described without vertex numbers. Hence a substructure has many instances which vary

essentially with respect to the vertex numbers that make them unique. Substructures of count 1 are eliminated from the Sub_Fold_2 as they do not contribute to the repeated higher edge substructures (As we are considering only exact matches; this will not be the case for inexact matches.) The updated Sub_Fold_2 is shown in Tab 3.6. Often we may be interested only in expanding k best substructures. To achieve this we sort the Sub_Fold_n in the descending order of the evaluation metric (DMDL or Count) and retain only the top k substructures (k corresponds to the beam parameter value). In order to get at the instances corresponding to the substructures in Sub_Fold_n, we join the updated Sub_Fold_n with Instance_n and insert the resulting instances into another table called InstanceIter_n. Only the instances present in InstanceIter_n participate in the next level of expansion.

Table 3.6 Updated Sub_Fold_2

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

Table 3.7 Instanceiter_2

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	F1	T1	F2	T2
1	2	3	A	B	C	1	2	AB	AC	1	2	1	3
5	6	7	A	B	C	5	6	AB	AC	1	2	1	3
8	9	10	A	B	C	7	8	AB	AC	1	2	1	3

3.2 Generalization of Expansion

The InstanceIter_1 relation shown in Tab 3.7 contains all the instances of the single edge substructure that have been chosen for further expansion. Each single edge substructure can be expanded to a two-edge substructure on *any of the two vertices* in the edge. In general, an n edge substructure can be expanded on *n + 1 vertices* in the substructure. All possible single edge substructures are listed in the oneedge relation. So by making a join with the oneedge relation we can always extend a given substructure by one edge. In order to make an extension, one of the vertices in the substructure has to match a vertex in the oneedge relation. The following queries extend a single edge substructure in all possible ways:

```
/* Query to expand self edge on vertex 1 */
INSERT INTO instance_2 (
   SELECT s.vertex1, s.vertex2, 0, s.vertex1name, s.vertex2name, '-',
          s.edge1, o.edge, s.edge1name, o.edgename, s.from_1, s.to_1, 1, 1
   FROM   InstanceIter_1 s,oneedge o
   WHERE  o.vertex1=s.vertex1 and o.edge<>s.edge1 and o.vertex2=s.vertex1)

/* Query to expand multiple edge from V1 to V2*/
INSERT INTO instance_2 (
   SELECT  s.vertex1, s.vertex2, 0, s.vertex1name, s.vertex2name, '-',
           s.edge1, o.edge, s.edge1name, o.edgename, s.from_1, s.to_1, 1, 2
   FROM    InstanceIter_1 s, oneedge o
   WHERE   o.vertex1=s.vertex1 and o.edge<>s.edge1 and o.vertex2=s.vertex2)

/* Query to expand multiple edge from V2 to V1*/
INSERT INTO instance_2 (
   SELECT s.vertex1, s.vertex2, 0, s.vertex1name, s.vertex2name, '-',
```

```
          s.edge1, o.edge, s.edge1name, o.edgename, s.from_1, s.to_1, 2, 1
   FROM   InstanceIter_1 s, oneedge o
   WHERE  o.vertex1=s.vertex2 and o.edge<>s.edge1 and o.vertex2=s.vertex1)

/* Query to expand self edge on vertex 2 */
INSERT INTO instance_2 (
   SELECT  s.vertex1, s.vertex2, 0, s.vertex1name, s.vertex2name, '-',
           s.edge1, o.edge, s.edge1name, o.edgename, s.from_1, s.to_1, 2, 2
   FROM    InstanceIter_1 s, oneedge o
   WHERE   o.vertex1=s.vertex2 and o.edge<>s.edge1 and o.vertex2=s.vertex2)

/* Query to expand on V1 to new vertex V3 */
INSERT INTO instance_2 (
   SELECT s.vertex1, s.vertex2, o.vertex2, s.vertex1name, s.vertex2name,
          o.vertex2name, s.edge1, o.edge, s.edge1name, o.edgename,
          s.from_1, s.to_1, 1, 3
   FROM   InstanceIter_1 s, oneedge o
   WHERE  o.vertex1=s.vertex1 and o.edge<>s.edge1 and  o.vertex2<>s.vertex1
          and  o.vertex2<>s.vertex2)

/* Query to expand on V2 to new vertex V3 */
INSERT INTO instance_2 (
   SELECT s.vertex1, s.vertex2, o.vertex2, s.vertex1name, s.vertex2name,
          o.vertex2name, s.edge1, o.edge, s.edge1name, o.edgename,
          s.from_1, s.to_1, 2, 3
   FROM   InstanceIter_1 s, oneedge o
   WHERE  o.vertex1=s.vertex2 and o.edge<>s.edge1 and o.vertex2<>s.vertex1
```

```
          and  o.vertex2<>s.vertex2)

/* Query to expand incoming edge on V1 from new vertex V3 */
INSERT INTO instance_2 (
   SELECT  s.vertex1, s.vertex2, o.vertex1, s.vertex1name, s.vertex2name,
           o.vertex1name, s.edge1, o.edge, s.edge1name, o.edgename,
           s.from_1, s.to_1, 3, 1
   FROM    InstanceIter_1 s, oneedge o
   WHERE   o.vertex2=s.vertex1 and o.edge<>s.edge1 and  o.vertex1<>s.vertex1
           and  o.vertex1<>s.vertex2)

/* Query to expand incoming edge on V2 from new vertex V3 */
INSERT INTO instance_2 (
   SELECT  s.vertex1, s.vertex2, o.vertex1, s.vertex1name, s.vertex2name,
           o.vertex1name, s.edge1, o.edge, s.edge1name, o.edgename,
           s.from_1, s.to_1, 3, 2
   FROM    InstanceIter_1 s, oneedge o
   WHERE   o.vertex2=s.vertex2 and o.edge<>s.edge1 and  o.vertex1<>s.vertex1
           and  o.vertex1<>s.vertex2)
```

Since the connectivity attributes are different for each type of expansion we need separate queries for each kind of expansion. Two kinds of expansion are possible. One in which both the vertices of the newly added edge are already present in the substructure instance. Second, in which only one of the vertex of the newly added edge is present in the substructure instance. Queries 1 to 4 show the expansions corresponding to the former case. Queries 5 to 8 show the expansion corresponding to the latter case. If both the vertices of newly added edge are already present in the substructure instance, then we are expanding a cycle or a multiple edge and therefore we mark the repeating

vertex number by vertex invariants 0's and -'s which we will be discussing in detail in the design chapter. In general for expanding an n edge substructure, we need n^2 queries for matching V1 of oneedge onto V1 of the instance table and comparing V2 of oneedge to V1, V2..Vn of instance table, then matching V1 of oneedge onto V2 of the instance table and comparing V2 of oneedge to V1, V2..Vn of instance table and so on. In addition to that we need n queries for expanding outgoing edges, matching V1 of oneedge onto V1, V2..Vn of instance table and adding the new vertex V2 of oneedge into the substructure instance. Also we need n queries for expanding incoming edges, matching V2 of oneedge onto V1, V2..Vn of instance table and adding the new vertex V1 of oneedge into the substructure instance. In short to expand n edge substructure to *n+1* edge substructure we need n^2+2*n queries. To avoid the inclusion of an existing edge into the substructure instance we compare the edge number of the incoming edge with all the edge numbers of the existing edges in the instance and allow the inclusion only when it is not already present.

The n edge substructure instances are stored in the InstanceIter_n relation. In order to expand to n+1 edge substructures, an edge is added to the existing n edge substructure. Therefore, the InstanceIter_n relation is joined with the oneedge relation to add an edge to the substructure. The Sub_Fold_n relation is in turn generated from the Instance_n relation. For each substructure the DMDL value is computed. Then we apply user specified beam to retain only best k substructures in Sub_Fold_n. The InstanceIter_n is then generated which has only the instances of the beam tuples that will be used for expansion to the next size. The main halting condition for the algorithm is the user specified parameter termed as MaxSize. Once the algorithm discovers all the substructures of the MaxSize the program terminates. Another halting condition for the algorithm is when no substructures are generated for expansion. This can happen

if all the substructures generated have a count value of 1 in which case they will all be eliminated.

3.3 Limitations of this approach

There are some limitations of representing a subgraph as a tuple of a relation. The approach presented in HDB-Subdue uses a tuple to represent a subgraph. This means that the number of attributes in the relation will grow as the size of the substructure increases. This may eventually place a limit on the size of the maximum substructure that can be detected, as there is a limit on the number of columns a relation can have in a Relational database. It is 1012 for the DB2 database system and 1000 for Oracle 10g. Since, the instance_n would need 6n+2 attributes for describing an n edge substructure, the algorithm can discover substructures of size 165 at the most. It may vary for other commercial databases.

3.4 Summary

In this section we discussed how we capture the graph information in database using relations and how we prune the count one instances to avoid wasteful computation in future expansions. Also we discussed a generalization for substructure expansion covering all possible expansion types and we also talked about the limitations of storing graphs in relational databases.

CHAPTER 4

DESIGN ISSUES

This chapter gives a detailed explanation of the design issues involved in modeling certain aspects of graph mining. Some of the aspects that will be discussed in this chapter are *unconstrained expansion*, *Pseudo duplicate elimination*, *generalization of the DMDL formula*, *Handling multiple edges*, *canonical label ordering*, and *hierarchical reduction*.

We first establish a complete and correct representation of a graph on which the rest of the computations are performed.

4.1 Need for new substructure instance representation

A substructure is represented by a set of attributes to capture the features of a graph. EDB-Subdue uses a set of attributes called extensions (in addition to others) to capture edge information. It captures the direction of the edge by storing the originating vertex attribute number and uses the default value for the other vertex. An extension attribute is either positive or negative. If the extension is positive, then the edge is originating from an existing vertex and terminating in the newly added vertex. If the extension is negative then the edge is originating from the newly added vertex and terminating in an existing vertex. Since one attribute is used for representing an edge, an n-edge substructure is captured with n-1 extension attributes; the direction of the first edge is assumed to be always from Vertex1 (actually the vertex number in vertex1) to Vertex2. Tab 4.1 shows all the three edge substructure instances using EDB-Subdue representation for the graph shown in Fig 4.1. Let us consider the first substructure. The extension E1 with value 1 says that the second edge is originating in vertex 1 and

ending in vertex 2 (the value of V2). The extension E2 with value 3 says that the third edge is originating in vertex 3 and ending in vertex 2 (the value of V4).

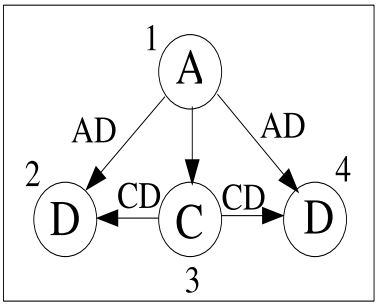

Figure 4.1 Avoiding spurious count

4.1.1 Limitation of extensions

A positive EDB-Subdue extension - essentially the vertex number from which the newly added edge is originating, does not give any information about which vertex it is terminating at. Let us consider the second substructure instance in Tab 4.1. The extension 3 says that the edge CD is originating in V3 which is 3. In the third substructure also extension 3 says that the edge CD is originating in V3 which is 3. But notice that in the first two substructures the edge CD terminates at the vertex D referred to by the edge AD but in the third substructure the edge CD terminates at a different D other than the one referred to by edge AD. But when we project on the vertex labels, edge labels and extensions we get a count of three instead of two, which is incorrect.

4.1.2 Connectivity map

The discrepancy mentioned in the previous section in EDB-Subdue extensions is due to the absence of the terminating vertex information. So the solution is to have both originating and terminating vertex numbers for each edge in the substructure. Table 4.2

Table 4.1 EDB-Subdue instances

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	E1	E2
1	2	3	2	A	D	C	D	AD	AC	CD	1	3
1	4	3	4	A	D	C	D	AD	AC	CD	1	3
1	4	3	2	A	D	C	D	AD	AC	CD	1	3

shows different connectivity map for the third instance differentiating it from the other two. So when we project on the vertex and edge labels we will get the correct count which is two in this case.

Table 4.2 HDB-Subdue instances

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3
1	2	3	-	A	D	C	-	AD	AC	CD	1	2	1	3	3	2
1	4	3	-	A	D	C	-	AD	AC	CD	1	2	1	3	3	2
1	4	3	2	A	D	C	D	AD	AC	CD	1	2	1	3	3	4

The number referred by the connectivity attributes correspond to the column position where the vertex occurs and it does not refer to the actual vertex number. Also in case of cycles or multiple edges when the vertices repeat, we set the repeating vertex's vertex number and vertex label to vertex invariant markers, '0' and '-' respectively. This is explained in detail in the section where we explain how cycles are handled. In the connectivity map, when referring to a vertex index, we always use the first occurrence of that vertex and never point to the attributes containing the vertex invariant markers.

4.2 Unconstrained expansion and Elimination Of Duplicate Instances

The substructure expansion in HDB-Subdue is not constrained using a rule on vertex number as it is done in EDB-Subdue. Hence there is a possibility that the same substructure is expanded in more than one way. For example let us consider a two edge

expansion in Fig 4.2. The edge 'AB' can grow into 'AB, CA' or the edge 'CA' can grow into 'CA,AB' as shown in Tab 4.3. These two substructures are essentially the same substructures but has been grown in two different ways. We term these as pseudo duplicates as they come out to be duplicates if the vertex and connectivity are rearranged without changing the graph structure. Similarly 'AD' can grow into 'AD, CA' or 'CA' can grow into 'CA, AD'. In general an n edge substructure (with n=1 vertices) can grow into an n+1 edge substructure in n+1 (one way from each vertex) ways. If we do not eliminate the pseudo duplicates, the same substructure will come out as different ones. Eventually, this will result in some genuine substructures being pruned when we apply the beam.

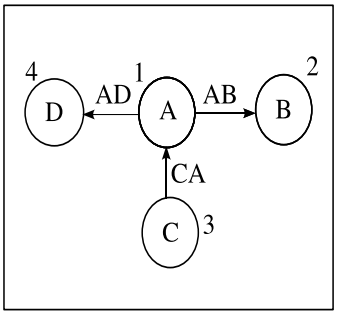

Figure 4.2 Pseudo duplicates

Table 4.3 Instance table - Before canonical ordering

Id	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	1	2	3	A	B	C	AB	CA	1	2	3	1
2	3	1	2	C	A	B	CA	AB	1	2	2	3
3	1	3	4	A	C	D	AD	CA	1	3	2	1
4	3	1	4	C	A	D	CA	AD	1	2	2	3

In order to identify that two substructure instances are pseudo duplicates of each other, vertex numbers and the connectivity attributes are necessary. If two instances have the same vertex numbers and edge directions then we can identify them as pseudo duplicates. In SQL we can identify the pseudo duplicates only if the vertex numbers and connectivity map of all the instances are canonically ordered. Since databases do not allow rearrangement of columns, to obtain canonical ordering, we have to transpose the rows of each substructure into columns, sort them and reconstruct them to get the canonical order.

Owing to the table space constraints, canonical ordering of only the second and fourth instance are shown below, as the first and third instances are already sorted on vertex numbers. We project the vertex numbers and vertex names from the instance table and insert them row wise into a relation called unsorted as shown in Tab 4.4. We also include the position in which the vertex occurs in the original instance. To differentiate between the vertices of different instances we carry the primary key Id from the instance table onto the unsorted table. Next we sort the table on *Id and vertex number* and insert it into a table called Sorted as shown in Tab 4.5 with its New attribute pointing to the new position of the vertex and the attribute Old pointing to the old position of the vertex.

Table 4.4 Unsorted

Id	V	VL	Pos
2	3	C	1
4	3	C	1
2	1	A	2
4	1	A	2
2	2	B	3
4	4	D	3

Table 4.5 Sorted

Id	V	VL	Old	New
2	1	A	2	1
2	2	B	3	2
2	3	C	1	3
4	1	A	2	1
4	3	C	1	2
4	4	D	3	3

Table 4.6 Old_Ext

Id	EL	F	T
2	CA	1	2
2	AB	2	3
4	CA	1	2
4	AD	2	3

Similarly the connectivity attributes are also transposed into a table called Old_Ext as shown in Tab 4.6. Since the sorting on vertex numbers has changed its position we need to update the connectivity attributes to reflect this change. Therefore we do a 3 way join of Sorted and Old_Ext tables on the Old attribute of the Sorted table to get the updated connectivity attributes which we call New_Ext as in Tab 4.7. Next we sort the New_Ext table on Id and the attributes F (From vertex) and T (Terminating vertex).

Table 4.7 New_Ext

Id	EL	F	T
2	CA	3	1
2	AB	1	2
4	CA	2	1
4	AD	1	3

Table 4.8 Sorted_Ext

Id	EL	F	T
2	AB	1	2
2	CA	3	1
4	AD	1	3
4	CA	2	1

Now that we have the ordered vertex as well as connectivity map tables, we can do a *2n+1* way join (where n is the current substructure size) of *n+1* Sorted tables and n Sorted_Ext tables to reconstruct the original instance in the canonical order.

Table 4.9 shows the substructures after canonically ordering the vertex numbers and the connectivity attributes. After having the instances ordered, a GROUP BY on the vertex numbers and the connectivity attributes will bring all the pseudo duplicates together and we can retain the instance with highest Id value and eliminate the rest. The choice of highest Id value is arbitrary and one could have chosen smallest Id value too.

The pseudo duplicate identification and removal essentially eliminates identical graphs represented by two different tuples as they were expanded in different ways.

Table 4.9 Instance table - After canonical ordering

Id	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	1	2	3	A	B	C	AB	CA	1	2	3	1
2	1	2	3	A	B	C	AB	CA	1	2	3	1
3	1	3	4	A	C	D	AD	CA	1	3	2	1
4	1	3	4	A	C	D	AD	CA	1	3	2	1

4.3 Canonical Label ordering

Since the substructure expansion is unconstrained it is likely that two *instances* of the *same structure* start from different initial edge and grow in a different order. Although these instances are similar they may not group together when counting for instances. For example consider a two edge substructure 'AB, AC' in Fig 3.1. There are three instances of this substructure as shown in Tab 4.10. When we project by the vertex and edge labels to count the number of instances of the substructure the second and third instance will group together resulting in a count of two instead of three. This is because the first instance started with the edge AB and expanded to 'AB, AC' and other two instances started with AC to grow to 'AC, AB'.

Table 4.10 Before ordering

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	2	3	A	B	C	AB	AC	1	2	1	3
5	7	6	A	C	B	AC	AB	1	2	1	3
8	10	9	A	C	B	AC	AB	1	2	1	3

To make the count independent of the order of substructure growth, we rearrange the vertex and edge labels in lexicographic order so that all instances of same substructure have their vertex and edge labels occurring in the same order. The sorting is done in the

same way as it was done in pseudo-duplicate elimination; we transpose the instance table to unsorted table and Old_Ext table. Instead of sorting the unsorted table on the vertex numbers, we sort it on the vertex label. Then we obtain the New_Ext table by joining Sorted and Old_Ext relations. Then we sort the New_Ext table on edge label to create Sorted_Ext table. To reconstruct the original instance table we join n+1 copies of Sorted tables with n copies of Sorted_Ext. The resulting instance table will have the vertex and edge labels occurring in a lexicographic order. The table after canonical ordering is shown in Tab 4.11. After ordering if we project by vertex and edge labels we would get the correct substructure count, in this case we would get three.

Table 4.11 After ordering

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	2	3	A	B	C	AB	AC	1	2	1	3
5	6	7	A	B	C	AB	AC	1	2	1	3
8	9	10	A	B	C	AB	AC	1	2	1	3

Canonical label ordering, although similar to vertex and connectivity ordering, serves a different purpose. Here we are ordering the instances, so that the instances of the *same substructure*, even though grown in different order, would group together when we do a GROUP BY on the vertex and edge labels and connectivity attributes. In contrast, canonical vertex and connectivity ordering helped us identify *duplicate structures* that were due to the growing of the *same structure in different ways*.

4.4 Multiple Edges

Most often one can find double or triple bonds occurring in chemical compounds. The atoms (such as carbon, hydrogen etc.) can be treated as vertices and the double and

triple bonds can be viewed as multiple edges between the vertices as shown in figure 4.3. In mining transactional data, the transacting entity would be represented by a vertex and the transaction by an edge. It likely that multiple transactions might occur between the same entities. Social networks are also gaining popularity in the applications of graph mining and often one can find multiple relationships existing between two people. The examples mentioned above are some of the areas where multiple edges are needed for representing the relationships accurately. Since EDB-Subdue does not expand multiple edges HDB-Subdue has been developed in an effort to support multiple edges between vertices.

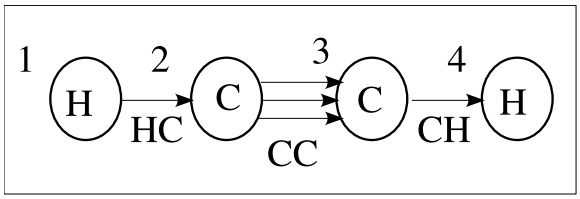

Figure 4.3 Acetylene

Table 4.12 Oneedge table with multiple edges

Vertex 1	Vertex 2	Edge No	Edge Label	Vertex1 Name	Vertex2 Name
1	2	1	HC	H	C
2	3	2	CC	C	C
2	3	3	CC	C	C
2	3	4	CC	C	C
3	4	5	CH	C	H

4.4.1 Significance of Edge numbers

The multiple edges between the same pair of vertices have the same starting and ending vertex number, same connectivity map and may have the same edge label. Hence, the above by themselves are not adequate to differentiate between multiple edges between the same nodes. To distinguish between the multiple edges we use *edge numbers* which are unique across all the edges. From the edge number one can differentiate two substructures based on the edge numbers associated with the edges. One-edge table in Tab 4.12 shows the *Edge No* attribute with unique edge numbers for multiple edges as well as other edges. Note that the the edge numbers are added transparently by the system and is not part of the input.

4.5 Handling Cycles in input graph

A figure similar to one shown in Pseudo duplicate elimination section, but with a cycle is shown in Fig 4.5. From this figure two valid 3-edge expansions are possible. One starting with vertex 1, adding vertex 2 (edge AB), adding vertex 3 (edge BC) and then terminating at vertex 1 (edge CA). Another expansion can start with vertex 2, adding vertex 3 (edge BC), adding vertex 1 (edge CA) and then terminating at vertex 2 (edge AB). Both the expansions occur because of HDB-Subdue's unconstrained expansion.

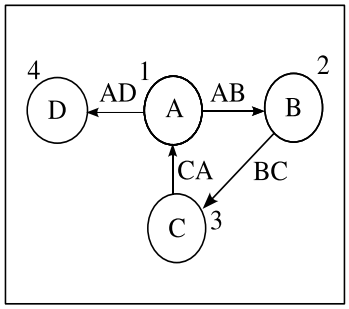

Figure 4.4 Cycle

The instance table with these expansions is shown in Tab 4.13. Both these instances are essentially the same, the second being the duplicate of the first. To identify the duplicate we would order the vertex numbers and the connectivity attributes canonically. Even when we order the instances canonically, we would not be able to identify the pseudo duplicate because the first instance would have the vertex numbers ordered as '1 1 2 3' and the second instance would have the vertex numbers ordered as '1 2 2 3'. So a group by query on the vertex numbers and connectivity attributes will not group these two instances together.

Table 4.13 Instance table without Vertex invariants

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	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

The solution to this problem is to mark each repetition of the vertex by vertex invariants, '0' for a repeating vertex number and a '-' for the corresponding vertex label. For this example, in the first instance the second occurrence of 1 is marked with '0' and its vertex label marked with '-'. In the second instance the second occurrence of 2 is marked with '0' and its vertex label marked with '-'. The instance table with vertex invariant markers is shown in Tab 4.14. Now if we canonically order the instances by vertex numbers and connectivity attributes the vertex numbers of the first instance, '1 2 3 0' will be ordered as '0 1 2 3', and the vertex numbers of the second instance '2 3 1 0' will be ordered as '0 1 2 3' as shown in Tab 4.15. Now when we group by the vertex numbers and the connectivity attributes we can easily identify that the second instance is a duplicate of the first and eliminate the second. Observe that the connectivity attributes do not refer to the columns marked with vertex invariants.

Table 4.14 Instance table with Vertex invariants

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3
1	2	3	0	A	B	C	-	AB	BC	CA	1	2	2	3	3	1
2	3	1	0	B	C	A	-	BC	CA	AB	1	2	2	3	3	1

Table 4.15 Instance table after canonical ordering

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3
0	1	2	3	-	A	B	C	AB	BC	CA	2	3	3	4	4	2
0	1	2	3	-	A	B	C	AB	BC	CA	2	3	3	4	4	2

4.6 Database Minimum Description Length

Database Minimum description length principle (DMDL) is a variant of minimum description length principle (MDL) [5, 9] used in Subdue. This metric is used to differentiate between same signature substructures (substructures having same number of vertices and frequency of occurrence.) Subdue uses adjacency matrices to represent substructures. Let us consider the example graph shown in Fig 4.5 and its two substructures of size two and count two are shown in Fig 4.6. The adjacency matrices corresponding to the two edge substructures are shown in Tab 4.16 and Tab 4.17 respectively. The adjacency matrices for the two graphs are different even though they have the same number of vertices and same number of edges.

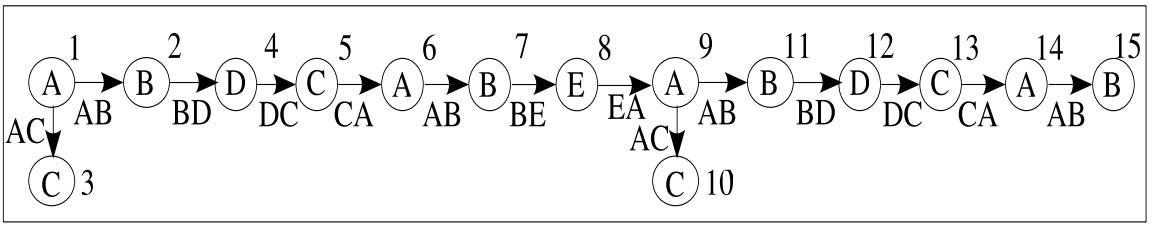

Figure 4.5 Example

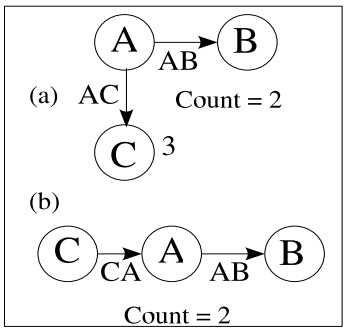

Figure 4.6 2 edge substructures

According to the MDL principle, the number of bits needed to encode matrix 1 is less than the number of bits required to encode matrix 2. This is because that in matrix 1 there is only one non-zero row. But in matrix 2, both the first and second rows have 1's (two non-zero rows). As a result, two rows have to be represented in the second case whereas only one row has to be represented in the first case. For the input graph of Fig 4.6, if graph compression is performed using the first substructure the resultant MDL value is 1.14539 and if the graph is compressed using the second substructure the MDL value obtained is 1.12849. Therefore, the MDL principle ranks substructure (a) higher than substructure (b). The MDL formula is shown below:

$$MDL = \frac{DL(G)}{DL(S) + DL(G|S)} \tag{4.1}$$

In the above formula, lesser the number of bits needed to represent DL (S) + DL (G|S), better is the substructure.

DMDL is calculated by using the formula,

$$DMDL = \frac{Value(G)}{Value(S) + Value(G|S)} \tag{4.2}$$

Table 4.16 Adj matrix of (a)

	A	B	C
A	0	1	1
B	0	0	0
C	0	0	0

Table 4.17 Adj matrix of (b)

	A	B	C
A	0	1	0
B	0	0	0
C	1	0	0

In the above equation, G represents the entire graph, S represents the substructure and G|S represents the graph after it has been compressed using the substructure S.

```
Value(G) = graph_vertices + graph_edges

Value(S) = sub_vertices + nonZeroRows

Value(G|S) = (graph_vertices  sub_vertices * count + count) +

             (graph_edges  sub_edges * count)
```

The graph_vertices and graph_edges parameters have a value of 15 and 14 respectively for the example graph in Fig 4.5. The substructures (a) and (b) in Fig 4.6 have a value of 3 for the sub_vertices parameter. The nonZeroRows parameter for substructure (a) has a value of 1 whereas for substructure (b) it is 2. Since both the substructures occur twice in the input graph the count is 2. If we compute the DMDL value for the substructures using the formula Eqn 4.2, we would obtain 1.074 for substructure (a) and 1.035 for substructure (b). Therefore HDB-Subdue will rank graph (a) higher than the graph (b), which follows the same trend as MDL.

The calculation uses counts of nodes and vertices instead of bit representation as in MDL. However, we have tried to simulate the MDL value in its trend (and not actual value) to validate the algorithm against the output produced by Subdue. In the following subsections we will discuss how we calculate the parameters 'sub_vertices, nonZeroRows and count' that go into the DMDL formula.

4.6.1 Number of substructure vertices

The number of substructure vertices (graph_vertices) is a parameter in the DMDL formula that is used to calculate Value(G). In EDB-Subdue this parameter was assumed to be one plus the number of edges in the substructure, i.e. n+1 vertices for n edges. This assumption holds true in the absence of multiple edges and cycles. A pair of vertices can have more than one edge between them and a substructure containing such multiple edges have fewer vertices than the edges. In case of cycles and self edges n vertices can have more than n edges between them. In HDB-Subdue a vertex number is allowed to occur at most once and a repetition is marked by vertex invariant markers. Since the connectivity map attributes refer only to non-marker vertices, a count on the unique connectivity map attributes will give us the correct count of the vertices in the substructure. For example consider the first three edge substructure 'AB, AB, AB' from Fig 4.7 shown in Tab 4.18. This substructure has three edges and since the unique number of connectivity attributes is two ('1' and '2') we can conclude that only two vertices exist in this substructure.

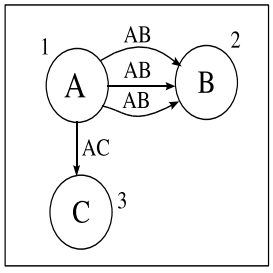

Figure 4.7 DMDL

4.6.2 Non Zero Rows

The DMDL principle attempts to simulate the adjacency matrix representation by using the number of non-zero rows for computing value(s). When the number of

Table 4.18 Subfold_3

VL1	VL2	VL3	V4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3	Count
A	B	-	-	AB	AB	AB	1	2	1	2	1	2	1
A	C	B	-	AC	AB	AB	1	2	1	3	1	3	3

non-zero rows is less the Value(S) in the denominator decreases thereby increasing the DMDL value. In EDB-Subdue the non-zero rows parameter is calculated by using the extensions and since in HDB-Subdue we have the connectivity map instead of extensions we calculate the non-zero rows by counting the unique origin vertices (From_n). Table Tab 4.19 represents the two edge substructures for the example in Fig 4.6. The non-zero rows for the first substructure is 1, because the origin vertex set contains '1,1' and non-zero rows for the second substructure is 2, because the origin vertex set contains '1,2'.

Table 4.19 Subfold_3

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2	Count
A	B	C	AB	AC	1	2	1	3	2
C	A	B	CA	AB	1	2	2	3	2

4.6.3 Instance counting

The count of a substructure represents the number of times it will be used to compress the graph. In other words, it denotes the number of vertices that the substructure removes from the input graph during compression. In EDB-Subdue every instance of a substructure is assumed to compress the graph. In the presence of multiple edges some instances compress the same set of vertices and hence in HDB-Subdue we need to change the way the substructure instances are counted. For example, Tab 4.18 shows three

three-edge substructures ‘AC, AB, AB’, ‘AC, AB, AB’ and ‘AC, AB, AB’ because of the presence of three multiple edges between vertex A and B. Although *three instances* of this substructure exist we need to count it to be *one* because all three substructures compress the *same set of vertices*. Let us consider the substructure ‘AC, AB, AB’ shown in Tab 4.20 and see how we obtain the count of its instances. Grouping by the vertex labels, edge labels and connectivity attributes, we can obtain a count of the instances of a substructure. The instances of the substructure in Tab 4.20 is shown in Tab 4.21.

Table 4.20 Subfold_3

VL1	VL2	VL3	V4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3	Count
A	C	B	-	AC	AB	AB	1	2	1	3	1	3	3

Table 4.21 Instance_3

V1	V2	V3	V4	VL1	VL2	VL3	V4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3
1	3	2	0	A	C	B	-	AC	AB	AB	1	2	1	3	1	3
1	3	2	0	A	C	B	-	AC	AB	AB	1	2	1	3	1	3
1	3	2	0	A	C	B	-	AC	AB	AB	1	2	1	3	1	3

In addition to the above group by, we group again by the vertex numbers of the instances of a particular substructure. This returns the groups of instances having unique vertex numbers and is shown in Tab 4.22. This table shows one group with two instances in it. Finally we count the number of such groups to obtain the updated count. For table Tab 4.22 if we count the number of groups we would get 1 as the count and we update the subfold_3 table in Tab 4.20 with the obtained count.

Table 4.22 Unique vertex number groups

V1	V2	V3	V4	VL1	VL2	VL3	V4	EL1	EL2	EL3	F1	T1	F2	T2	F3	T3
1	3	2	0	A	C	B	-	AC	AB	AB	1	2	1	3	1	3

4.7 Hierarchical Reduction

Hierarchical reduction is the process of compressing the input graph repeatedly to produce an abstract information. This is particularly useful for recursively identifying patterns among the patterns. For example in protein data analysis hierarchical reduction can detect amino acids and also identify how the amino acids group to form polypeptides and also how polypeptides group to form proteins.

An example of hierarchical reduction is shown in figure 4.8. In the first iteration, substructure shown within dotted lines in (a) comes out as the best substructure and each of its instances (there are four of them) is compressed to a node called SUB_1 as shown in (b). Now this graph becomes the input for the second iteration. After the second iteration, the substructure shown within dotted lines in (b) comes out as the best substructure and each of its instances (there are two of them) is compressed to a node called SUB_2. Now the compressed graph as shown in (c) becomes the input for next iteration. Since we cannot compress the input graph anymore we stop after the third iteration. In general each of the best substructure instances after iteration i is compressed to SUB_i.

The following are the tasks involved in each iteration of hierarchical reduction,

1. Identify the best substructure and its instances
2. Compress the input graph using the best substructure instances
3. Submit the compressed graph as input to the next iteration

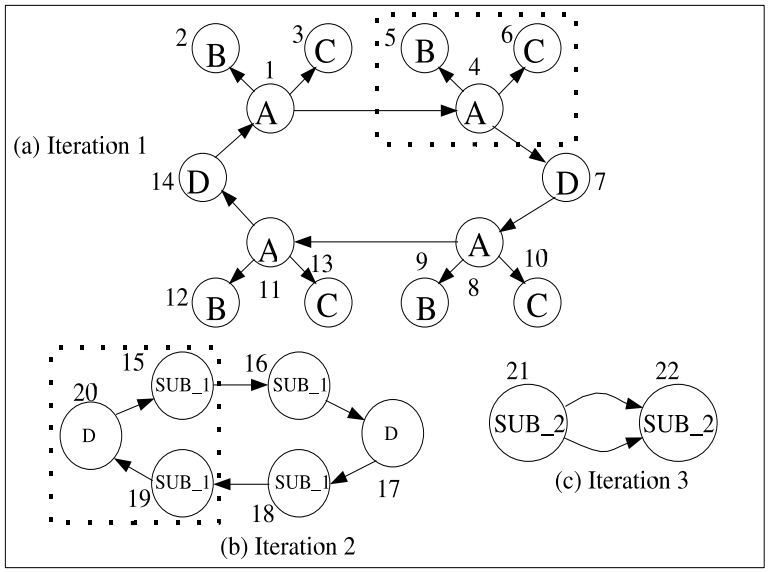

Figure 4.8 Hierarchical reduction

4.7.1 Identifying the best substructure

HDB-Subdue uses DMDL or count (given as a configuration parameter) for selecting the best substructure to compress the input graph. The substructure which has the highest DMDL value or highest count can compress the graph better than any other substructure. Two edge substructures of Fig 4.8 is shown in the Sub_Fold table 4.23. The Sub_Fold table is sorted in the descending order of DMDL value (or count in case the evaluation metric is count) and the first substructure is chosen as the best substructure to compress the graph. The instances of the best substructure are inserted into a table called BestInstances shown in Tab 4.24.

Table 4.23 Sub_Fold_2

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2	DMDL
A	B	C	AB	AC	1	2	1	3	1.50
A	D	A	AD	DA	1	2	2	3	1.09
.	.	.	.	.	.	.	.	.	.

Table 4.24 BestInstances

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	2	3	A	B	C	AB	AC	1	2	1	3
4	5	6	A	B	C	AB	AC	1	2	1	3
8	9	10	A	B	C	AB	AC	1	2	1	3
11	12	13	A	B	C	AB	AC	1	2	1	3

4.7.2 Compressing the input graph

To compress the input graph with the instances of the best substructure we need to remove the vertices and edges corresponding to the instances from the input graph. Since the input graph is defined by the vertex and edge table, we eliminate the vertices corresponding to the best substructure from the vertex table and the edges corresponding to the best substructure are removed from the edge table. The vertex and edge tables before compression are shown in Tab 4.25 and Tab 4.26 respectively. For each instance we remove, we add a new vertex SUB_i to the vertex table during the i^{th} iteration. The vertex numbers in the edges going into or out of the compressed instances need to be updated appropriately with the newly inserted vertex number as the old vertex numbers no longer exist in the vertex table. The vertex and edge tables after compression are shown in Tab 4.27 and Tab 4.28 respectively. The updated vertices and edges will participate in the next iteration until we hit the maximum number of iterations specified by the user or if there are no more subgraphs to compress.

4.8 Summary

In this chapter we introduced a correct and complete graph representation in database. We addressed the unconstrained expansion to expand multiple edges and the

Table 4.25 Vertex table before

Vertex No	Vertex Name
1	A
2	B
3	C
4	A
5	B
6	C
7	D
.	.
14	D

Table 4.26 Edge table before

Vertex 1	Vertex 2	Edge Label
1	2	AB
1	3	AC
4	5	AB
4	6	AC
4	7	AD
7	8	DA
.	.	.
11	14	AD
14	1	DA

Table 4.27 Vertex table after

Vertex No	Vertex Name
15	SUB_1
16	SUB_1
17	SUB_1
18	SUB_1
7	D
14	D

Table 4.28 Edge table after

Vertex 1	Vertex 2	Edge Label
16	7	AD
7	17	DA
11	18	AD
14	15	DA

technique to eliminate the duplicate instances produced. We also addressed how cycles are handled in the input graph and the modifications made in DMDL to take multiple edges and the new graph representation into account. Also we proposed a technique to perform Hierarchical graph reduction.

CHAPTER 5

IMPLEMENTATION DETAILS

This chapter discusses the implementation details of database minimum description length, substructure pruning, pseudo duplicate removal and hierarchical reduction. The HDB-Subdue system has been developed using Pro*C [21] precompiler and C language, using Oracle database running on a Linux based system. The SQL statements are embedded in the C code and the precompiler translates each embedded SQL statement into calls to the Oracle runtime library (SQLLIB).

5.1 Pseudo-duplicate Elimination

The unconstrained expansion in HDB-Subdue introduces pseudo duplicates as explained in the design section. To eliminate these pseudo duplicates it is necessary to sort the columns containing vertex numbers and connectivity attributes. We will be referring to the same example as in design chapter, but for explanatory purpose we would be using only the third and fourth instances in the instance table Tab 4.3 as shown in Tab 5.1

Table 5.1 Instance table - Before canonical ordering

Id	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
3	1	3	4	A	C	D	AD	CA	1	3	2	1
4	3	1	4	C	A	D	CA	AD	1	2	2	3

Since we can only sort the rows using an SQL query, we project out the instance identifier, vertex number and the column position into a relation called Unsorted. The

instance identifier, edge label and the connectivity attributes are inserted into a relation called Old_Ext. Next, we sort the Unsorted table on the instance identifier and vertex number and insert the tuples in a relation called Sorted. The queries to do the above follow.

```
INSERT into Unsorted(
      (SELECT Id,VL1,V1,1 FROM instance_n)
UNION (SELECT Id,VL2,V2,2 FROM instance_n)
          .    .   .   .
UNION (SELECT Id,VLn+1,Vn+1,n+1 FROM instance_n))

INSERT into Old_Ext(
      (SELECT Id,EL1,F1,T1 FROM instance_n)
UNION (SELECT Id,EL2,F2,T2 FROM instance_n)
          .    .   .   .
UNION (SELECT Id,ELn,Fn,Tn FROM instance_n))
```

The projected values for the instances in the previous table are shown in Tab 5.2 and Tab 5.3 and the Sorted table is shown in Tab 5.4. *2n+1* queries are required to project out the columns into rows where n is the substructure size.

Table 5.2 Unsorted

Id	V	VL	Pos
3	1	A	1
3	3	C	2
3	4	D	3
4	3	C	1
4	1	A	2
4	4	D	3

Table 5.3 Old_Ext

Id	EL	F	T
3	AD	1	3
3	CA	2	1
4	CA	1	2
4	AD	2	3

Table 5.4 Sorted

Id	V	VL	Old	New
3	1	A	1	1
3	3	C	2	2
3	4	D	3	3
4	1	A	2	1
4	3	C	1	2
4	4	D	3	3

After sorting, the column position occupied by the projected vertices changes. The connectivity attributes still point to the old position. To update the connectivity attributes with the new vertex positions, a column called new position is needed in the sorted table. Since the reconstructed instance table will have the vertices appear in the order as in sorted table, we make use of the in-built rownum attribute and a simple mod function to generate the new position. For an n edge substructure the new positions are updated using the following SQL statements. Since mod function returns 0 for the last column because its rownumber is a multiple of n+1, we use a separate query to update the correct position of last column.

```
UPDATE sorted SET rowno = rownum
UPDATE sorted SET new = mod(rowno,n+1)
UPDATE sorted SET new = n+1 WHERE new = 0
```

Since the sorting alters the original order of the vertices in the unsorted table, the connectivity attributes also need to be updated to reflect the change. To do that, we use the following query to join the Sorted and the Old_Ext table on the Id and old position attribute to obtain the new positions, and the resulting New_Ext table is shown in Tab 5.5.

```
INSERT into new_ext
 (SELECT o.Id, o.EL,s1.new,s2.new
 FROM sorted s1, sorted s2, old_ext o
 WHERE o.Id = s1.Id and s1.Id = s2.Id and
       o.F = s1.old and o.T = s2.old)
```

To obtain the canonical order of the connectivity attributes, we sort the New_Ext table on Id, F and T attributes and insert into a relation called Sorted_Ext as shown in Tab 5.6.

Table 5.5 New_Ext

Id	EL	F	T
3	AD	1	3
3	CA	2	1
4	CA	2	1
4	AD	1	3

Table 5.6 Sorted_Ext

Id	EL	F	T
3	AD	1	3
3	CA	2	1
4	AD	1	3
4	CA	2	1

To reconstruct the instance table back in canonical order, we do a *2n+1* way join of *n+1* Sorted tables and n Sorted_Ext tables. The query to do that is shown below and the reconstructed table is shown in Tab 5.7.

```
INSERT into Instance_n(
SELECT S1.Id, S1.V .. Sn+1.V, S1.VL .. Sn+1.VL, O1.EL .. On.EL,
       O1.F, O1.T .. On.F, On.T
FROM   Sorted S1 .. Sorted Sn+1, Sorted_Ext O1 .. Sorted_Ext On
WHERE  S1.Id=S2.Id .. Sn.Id=Sn+1.Id and O1.Id=O2.Id .. On.Id=On+1.Id
       and O1.rowno < O2.rowno .. On-1.rowno < On.rowno
       and S1.rowno < S2.rowno .. Sn.rowno < Sn+1.rowno)
```

Table 5.7 Instance table - After canonical ordering

Id	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
3	1	3	4	A	C	D	AD	CA	1	3	2	1
4	1	3	4	A	C	D	AD	CA	1	3	2	1

One can identify the set of pseudo duplicates of any instance by grouping by the vertex numbers and the connectivity attributes. To eliminate the pseudo duplicates of an instance we insert the Id of an instance that has maximum Id, into a relation called Pseudo_Del. In the next step we eliminate all the instances other than the ones inserted

in Pseudo_Del relation in the previous step. The instance table after eliminating the pseudo-duplicates is shown in Tab 5.8

```
INSERT into Pseudo_Del
(SELECT MAX(Id)
 FROM Instance
 GROUP BY V1 .. Vn+1, F1, T1 .. Fn, Tn HAVING count(*) > 1)

DELETE FROM Instance WHERE id NOT IN (SELECT Id FROM Pseudo_Del)
```

Table 5.8 Instance table - After Pseudo Elimination

Id	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
4	1	3	4	A	C	D	AD	CA	1	3	2	1

5.2 Substructure pruning

A user specified parameter called *beam* specifies the number of interesting substructures to retain in each iteration. Beam is applied on the Sub_Fold_n table after eliminating pseudo duplicates and calculating DMDL value for each substructure. Sub_Fold_n is ordered in the descending values of DMDL and the top, beam number of substructures are inserted into a relation called Beamdel_n. All the instances of the beam substructures in Beamdel_n are inserted into another relation called InstanceIter_n and only these instances are used for expanding to the next iteration by adding an edge. Beam is implemented using cursors in EDB-Subdue and for larger graph sizes, use of cursors pose a memory limitation.

5.2.1 Analytic Functions

In this thesis we use what are called Analytic Functions [22, 23] for implementing beam. Analytic functions have been available since Oracle 8.1.6 and are designed to address such problems as, 'Calculating a running total', 'Finding percentages within a group', 'Top-N queries', 'Compute a moving average' and many more. Most of these problems can be solved using standard PL/SQL, however the performance is often not what it should be. Analytic Functions add extensions to the SQL language that not only make these operations easier to code but also make them faster (as they are optimized by the system) than could be achieved with pure SQL or PL/SQL.

Analytic functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or ORDER BY clause.

Table 5.9 Sorted DMDL

DMDL
1.412
1.412
1.383
1.383
1.246
1.246
1.101

Table 5.10 Using Rownum

DMDL
1.412
1.412
1.383
1.383

Table 5.11 Sorted DMDL

DMDL
1.412
1.412
1.383
1.383
1.246
1.246
1.101

5.2.2 Top N with ties

There are several approaches for implementing the beam. One is to use cursors which is used in EDB-Subdue [8]. Second is to make use of the internal row number

attribute of the Sub_Fold_n relation. In this approach the Sub_Fold_n relation is sorted in the descending order of the DMDL values and a SELECT from this relation with a WHERE clause, 'WHERE rownum <= beam' would return the top beam substructures. The downside of this approach is that it does not handle ties. For example consider the sorted Sub_Fold_n relation in Tab 5.9. Assume that we are operating with a beam of 4. The above approach would return the first 4 substructures as in Tab 5.10. Notice that there is a tie between substructure 1 and 2, and substructure 3 and 4. Ideally we would want the best 4 substructures inclusive of their ties as in 5.11. In HDB-Subdue, for calculating DMDL we do not use the adjacency matrix as in Subdue, hence the probability of occurrence of ties is more. The query using the dense rank analytic function, shown below, can select the best beam substructure including all its ties.

```
INSERT INTO Beamdel_n
SELECT VL1..VLn, EL1.. EL2, F1, T1..Fn, Tn, dmdlvalue
FROM ( SELECT VL1..VLn, EL1.. EL2, F1, T1..Fn, Tn, dmdlvalue,
              DENSE_RANK() OVER (ORDER BY dmdlvalue desc ) TopN
       FROM Sub_Fold_n )
WHERE TopN <= beam
```

5.3 Database Minimum Description Length

The DMDL formula that was explained in the design chapter is calculated at run time for each substructure in the Sub_Fold_n table. This is accomplished by writing a PL/SQL function. The function is given below and comments are included for explanatory purposes but are not part of the actual function.

```
1: CREATE FUNCTION DMDLVALUE_N (from_1 IN NUMBER, to_1 IN NUMBER
   ··· from_n IN NUMBER, to_n IN NUMBER, count NUMBER, graph_vertices NUMBER,
   graph_edges NUMBER, sub_edges NUMBER) return NUMBER
```

```
2: IS /* Variable declarations go below */
3: nonZeroRows NUMBER; found NUMBER; sub_vertices NUMBER; sizeG NUMBER;
   index1 NUMBER; index2 NUMBER; dmdl_value NUMBER(10,5); sizeS NUMBER;
   sizeGS NUMBER; extarray extlist := extlist (from_1, from_2 ··· from_n);
   vertex extlist := extlist (from_1, to_1 ··· from_n, to_n);
4: BEGIN
5:    nonZeroRows := 0; /* Begin Non-zero rows calculation */
6:    for index1 IN 1..n loop
7:       found := 0;
8:       for index2 IN index1 + 1..n loop
9:          if extarray(index1) = extarray(index2) then found := 1; end if;
10:      end loop;
11:      if found = 0 then nonZeroRows := nonZeroRows + 1; end if;
12:   end loop; /* End Non-zero rows calculation */
13:   sub_vertices := 0; /* Begin Substructure vertices counting */
14:   for index1 IN 1..n loop
15:      found := 0;
16:      for index2 IN index1 + 1..n loop
17:         if vertex(index1) = vertex(index2) then found := 1; end if;
18:      end loop;
19:      if found = 0 then sub_vertices := sub_vertices + 1; end if;
20:   end loop; /* End Substructure vertices counting */
21:   sizeG := graph_vertices + graph_edges;
22:   sizeS := sub_vertices + nonZeroRows;
23:   sizeGS := ((graph_vertices - (sub_vertices * count)) + count)
24:                +(graph_edges - sub_edges * count);
25:   dmdl_value := (sizeG / (sizeS + sizeGS)); /* DMDL calculation */
26:   RETURN(dmdl_value);
27: END DMDLVALUE_N;
```

Lines 6 to 12 are used for counting the number of vertices that have an outdegree of at least one. This parameter helps to simulate the use of adjacency matrix representation as in Subdue. Lines 14 to 20 are used for counting the number of unique vertices in the substructure. In the presence of multiple edges and cycles the vertex numbers repeat in

a substructure. Since the connectivity attributes refer to the same vertex, in case that vertex repeats itself, a count on the unique connectivity attributes will give the count of unique vertices in the substructure. The PL/SQL function mentioned above is registered in the database and is then used in a SQL query that generates the Sub_Fold_n table. The query for obtaining the DMDL values for n edge substructure follows.

```
update Sub_Fold_n
SET dmdlvalue = DMDLVALUE_n(from_1, to_1, .. from_n, to_n,
                            count, graph_vertices, graph_edges, n)
```

If the maximum substructure size specified by the input parameter is n, n functions from DMDLVALUE_1 $\cdots$ DMDLVALUE_n, one for each substructure size is generated and registered with the database.

5.4 Hierarchical Reduction

To compress the graph input hierarchically, we need to identify the best substructure in each iteration and then use that substructure to compress the graph. The compressed graph then becomes the input for the subsequent iteration. In each expansion stage, we insert a user specified number (BestSubs) of best substructures having highest DMDL values from Sub_Fold_n into a relation called Sub_Fold. After expanding the substructures till the MaxSize, a substructure having highest DMDL value is picked from the Sub_Fold relation, to compress the graph for that iteration. The first step involves in selecting all the instances of the best substructure into a relation called BestInstances. We would be using the same example discussed in the design chapter. The query to do this is shown below and the relation itself is shown in Tab 5.12. Next we sequentially process the instances in the BestInstances relation one by one.

```
INSERT into BestInstances
SELECT i.V1 .. i.Vn+1, i.VL1 .. VL5, i.E1 .. i.En, i.EL1 .. ELn,
```

```
        i.F1, i.T1 .. Fn, i.Tn
FROM Sub_Fold s, Instance i
WHERE i.V1=s.V1 and .. i.Vn+1=s.Vn+1 and i.E1=s.E1 and .. i.En=s.En
       and i.F1=s.F1 and i.T1=s.T1 and .. i.Fn=s.Fn and i.Tn=s.Tn
       and s.dmdlvalue=MAX(dmdlvalue)
```

Table 5.12 BestInstances

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
1	2	3	A	B	C	AB	AC	1	2	1	3
4	5	6	A	B	C	AB	AC	1	2	1	3
8	9	10	A	B	C	AB	AC	1	2	1	3
11	12	13	A	B	C	AB	AC	1	2	1	3

To compress the input graph by a substructure, we insert one new vertex into vertex relation for every instance that is compressed. Next we update in oneedge relation, the vertex numbers of the edges that go into and out of the compressed instance. We compress the input graph by substituting one instance at a time. This is done in relations by processing the first tuple (with rownum = 1) and deleting it and processing the next tuple, then deleting it and so on. To keep track of the compressed instances and their corresponding compressing vertex, we insert each instance and its substituted vertex number and name into a relation called Compressed_n, where n is the iteration number we are operating at. The vertex, oneedge and Compressed_n tables are shown in Tab 5.13, Tab 5.14 and Tab 5.15 respectively. For every instance in the BestInstances relation, the queries shown below are executed.

```
{
  MAX_VERTEX = MAX_VERTEX + 1
```

Table 5.13 Vertex table before

VertexNo	VertexName
1	A
2	B
3	C
4	A
5	B
6	C
7	D
.	.
14	D

Table 5.14 Oneedge - Before

V1	V2	EdgeNo	EL	VL1	VL2
1	2	1	AB	A	B
1	3	2	AC	A	C
1	4	3	AA	A	A
4	5	4	AB	A	B
4	6	5	AC	A	C
4	7	6	AD	A	D
.	.	.	.	.	.
11	14	14	AD	A	D

Table 5.15 Compressed_1

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2	V	VL
1	2	3	A	B	C	AB	AC	1	2	1	3	15	SUB_1
4	5	6	A	B	C	AB	AC	1	2	1	3	16	SUB_1
8	9	10	A	B	C	AB	AC	1	2	1	3	17	SUB_1
11	12	13	A	B	C	AB	AC	1	2	1	3	18	SUB_1

```
INSERT into VertexTable(VertexNo, VertexName)
VALUES (MAX_VERTEX,'SUB_iterationNumber')

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

  INSERT into Compressed_iterationNumber
  SELECT V1 .. Vn+1, VL1 .. VL5, E1 .. En, EL1 .. ELn,
         F1, T1 .. Fn, Tn, MAX_VERTEX, 'SUB_iterationNumber'
  FROM BestInstances
  WHERE rownum = 1

  DELETE FROM BestInstances WHERE rownum = 1
 }
```

After compressing all the instances, the Compressed_n relation would contain the vertices and edges of all the compressed instances. We remove the vertices occurring in the compressed instances from the vertex table and the edges from the oneedge table. Also we join the vertex table with the oneedge table to update the edges with new vertex names. The oneedge table can then participate in the next iteration. The queries to do the above are shown below. Also the vertex table and the oneedge table after one iteration of hierarchical reduction is shown in Tab 5.16 and Tab 5.17 respectively.

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

  DELETE FROM oneedge
```

```
WHERE EdgeNo IN ((SELECT E1 FROM Compressed_iterationNumber)
           UNION (SELECT E2 FROM Compressed_iterationNumber)
                          . . . .
           UNION (SELECT En FROM Compressed_iterationNumber))
```

Table 5.16 Vertex table-After

VertexNo	VertexName
15	SUB_1
16	SUB_1
7	D
17	SUB_1
18	SUB_1
14	D

Table 5.17 Oneedge-After

V1	V2	EdgeNo	EL	VL1	VL2
15	16	1	AA	SUB_1	SUB_1
16	7	2	AD	SUB_1	D
7	17	3	DA	D	SUB_1
17	18	4	AA	SUB_1	SUB_1
18	14	5	AD	SUB_1	D

5.5 Summary

This chapter addressed the implementation issues that were discussed in the design chapter. It also presented the SQL queries to achieve the solution that was explained conceptually in the design chapter and also illustrated how the corresponding tables were updated in the database. Generalization was also given for each issue that was explained in this thesis.

CHAPTER 6

PERFORMANCE EVALUATION

This chapter gives an overview of the performance of HDB-Subdue and also compares it with that of Subdue. For graphs without multiple edges and cycles, performance comparison with EDB-Subdue is also shown. In this chapter we discuss the graph generator that generated the graphs which were used for testing, the configuration file that HDB-Subdue uses for reading the input parameters, and about the log file for recording the running times of different modules, experimental analysis for different kinds of input data sets and the observations and conclusions drawn from these experiments.

6.1 Graph Generator

This section explains the graph generator used for generating the data sets. The synthetic graph generator used for generating the input graphs for this thesis was developed by the AI Lab [24] at the University of Texas at Arlington. The graph generator accepts many parameters for constructing the graph. The parameters are listed below:

- ◇ Graph output filename
- ◇ Number of vertices in the graph
- ◇ Number of edges in the graph
- ◇ Number of unique vertex labels
- ◇ Number of unique edge labels
- ◇ Number of substructures to embed in the graph
- ◇ For each substructure

- Number of instances
- Number of vertices
- For each substructure vertex
  - ▷ The vertex label. It must be of the form v0, v1 etc..
- Number of edges
- For each substructure edge
  - ▷ The edge label. It must of the form e0, e1, etc..
  - ▷ The first vertex to which this edge is attached. An integer ranging from 0 to (number of substructure vertices - 1)
  - ▷ The second vertex to which this edge is attached. An integer ranging from 0 to (number of substructure vertices - 1)

Below is an example of an input to the graph generator.

```
T20V30E.g
20
30
10
15
2
1
2
v0
v1
1
e0
0
1
3
3
v0
v1
v2
3
```

```
e1
0
1
e1
1
2
e1
2
3
```

In the above example each parameter is specified on a separate line. The example contains 20 vertices and 30 edges. There are ten distinct vertex labels and fifteen distinct edge labels. Two different substructures are embedded in the graph. The first substructure will appear once. It contains two vertices (labeled v0 and v1). It has one edge (labeled e0), which goes between the two vertices. The second substructure is embedded three times. It contains three vertices (labeled v0, v1, and v2). It has three edges, all labeled e1, connecting the vertices in a triangle. The data set is labeled as T20V30E, which means that the graph generated from the graph generator has 20 vertices and 30 edges.

The graph generator is a main-memory algorithm and constructs graphs in main memory before writing it out to a file. The maximum graph size that we generated with the graph generator is 1.6 million vertices and 3.2 million edges. Since this is a main-memory algorithm the maximum size graph that it can generate will depend upon the available system memory.

6.2 Configuration File

A configuration file is useful for automating the process of performance evaluation. It consists of a number of parameters, which once specified, can be used for running the algorithm in an unattended mode. It can also be used for executing several data sets

with different configuration parameters. The different parameters in the configuration file are:

```
User Name # Password # Input Table Name # MaxSize # Beam # Iterations
# EvalApproach # BestSubs # Optimize # Debug # LogResults # Ties
```

User Name, Password: Needed to make a connection to the database

Input Table Name: The name of the relation containing the input graph data set

Max Size: Specifies maximum size of substructure to be discovered (in other words maximum number of edges allowed in a substructure)

Beam: Number of substructures to retain in every substructure size

Iterations: Specifies number of passes for Hierarchical reduction

EvalApproach: Substructure evaluation metric. 1 - Count, 2 - DMDL

BestSubs: Number of substructures to be retained in every substructure size, for selecting the best substructure during hierarchical reduction. If a zero is specified BestSubs will default to the value of beam.

Optimize: Specifies whether to use indexes on tables often used. 0 - Without Index, 1 - With Index

Debug: Print the SQL commands executed on STDOUT and errors on STDERR. 0 - Suppress, 1 - Print

LogResults: Writes out the running times of queries into a text file. 0 - Disable, 1 - Enable

Ties: Include ties when selecting substructures using beam. 0 - Ignore ties, 1 - Include ties

For each experiment, the values of all these parameters are specified in a single line in the order shown above and are separated by a # sign. The program will be invoked as many times as there are lines in the configuration file. An example entry in the configuration

file is shown below.

```
scott # tiger # 50V100E # 5 # 4 # 3 # 2 # 5 # 1 # 1 # 1
```

6.3 Writing Log File

Graph mining is a time-consuming process and for some data sets it may take hours to complete. To compare the performance of this approach with others, we run the Subdue, EDB-Subdue, and HDB-Subdue on the same machine. The running time of each query is captured in an array and at the completion of each dataset (at the end of every line in config file) it is written out to a log file, if the log results option is On. If the log results option is enabled a text file with <input table name>.txt is created in a logfiles directory; if the file already exists then the execution times are appended to the end of the file. In addition to the running times, the execution progress for each dataset is also written at the completion of each expansion iteration. The progress entry contains the iteration number, number of substructures and instances found in that iteration. Unlike the log entry which is written at the completion of a dataset, the progress entry is is written during the execution of the dataset and a 'tail' (in UNIX) or similar command on the <input table name>.txt will the last expansion iteration that completed. Given below is a sample content of log file with timings shown for iteration 2.

```
Iter1: Substructures 10 Instances 60 Iter2: Substructures 6 Instances 40 ..
Size1 .. Inst_2 Index_2 Pseudo_2 LabReord_2 SubFold_2 CountUpd_2 .. Total
         0.990  1.200   2.230    1.650      0.890     0.975       .. 8.132
```

Inst_n gives the time taken for extending from size n-1 substructure to size n substructure. Index_n tells the time it took to create index on the relations, Pseudo_n gives the time taken for eliminating pseudo duplicates on size n substructures, LabReord_n gives the time taken to canonically order the vertex and edge labels, SubFold_n gives

the time taken for creating, counting, and evaluating substructures from instances and CountUpd_n gives the time taken to update the correct count due to the presence of multiple edges and Total gives the time taken for all the iterations to complete.

6.4 Experimental Results

Experiments were conducted on several data sets of different sizes – from small to very large to verify the growth of computation time as the data sizes increase. For the purposes of comparison with Subdue and EDB-subdue, graphs with and without multiple edges, with and without cycles were used. All of the HDB-Subdue experiments were run 4 times and the first run (cold start) was discarded and the timings from other three runs were averaged to obtain average running times. All the experiments were performed on a Linux machine using Oracle9i Release 9.2.0.1.0. The machine was running on dual processors with 2 GBytes of memory.

6.4.1 Without cycles and multiple edges

The substructures that were embedded are shown in Fig 6.1. The number of instances embedded is proportional to the size of the dataset. We embedded roughly two times more substructures when the data set doubles. The graphs were generated by a synthetic graph generator explained in the earlier graph generator section.

Table 6.1 Parameter Settings

Parameters	Subdue	EDB-Subdue	HDB-Subdue
MaxSize	5	5	5
Beam	4	4	4
Iterations	1	1	1

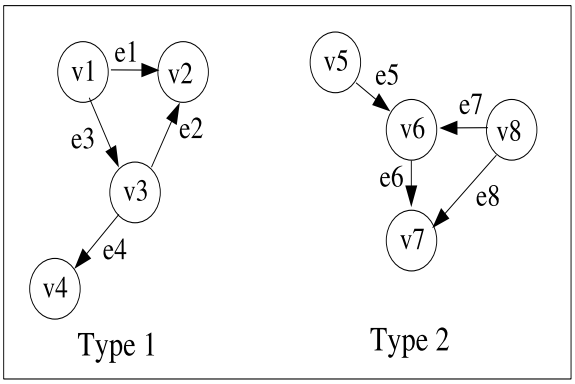

Figure 6.1 Graphs without cycles and multiple edges

The parameters used for performing this experiment is shown in Tab 6.1. Additionally we disabled the ties while using beam to limit substructures and also the substructure evaluation metric was set to DMDL. Subdue, EDB-Subdue and HDB-Subdue discovered the same substructures with all the embedded instances. The substructures discovered by Subdue, EDB-Subdue and HDB-Subdue for 1KV2KE dataset is shown below and the number of substructure instances discovered for each dataset is shown in table Tab 6.2.

```
Dataset: 1KV2KE
Substructure discovered by Subdue:
Substructure: value = 1.13157, pos instances = 60, neg instances = 0
  Graph(4v,4e):
    v 1 v1
    v 2 v2
    v 3 v3
    v 4 v4
    d 1 2 e1
    d 1 3 e2
    d 3 2 e3
    d 3 4 e4

Substructure discovered by HDB-Subdue:
Vertex1name Vertex2name Vertex3name Vertex4name Vertex5name
-           v1          v2          v3          v4
Edge1name Edge2name Edge3name Edge4name
e1        e2        e3        e4
From_1 To_1 From_2 To_2 From_3 To_3 From_4 To_4 count DMDL
```

```
2       3     2       4     4       3     4       5     60 1.13636
```

Table 6.2 Instances discovered

Dataset	Subdue	EDB-Subdue	HDB-Subdue
50V100E	4	4	4
250V500E	15	15	15
500V1000E	30	30	30
1KV2KE	60	60	60
2.5KV5KE	150	150	150
5KV10KE	300	300	300
7.5KV15KE	450	450	450
10KV20KE	600	600	600
5KV30KE	900	900	900
20KV40KE	1200	1200	1200
50KV100KE	3000	3000	3000
100KV200KE		6000	6000
200KV400KE		12000	12000
400KV800KE		24000	24000
800KV1600KE		48000	48000
1600KV3200KE		96000	96000

The performance comparison is shown in Fig 6.2. The X-axis shows the dataset and the Y-axis shows the running time in seconds. From the graph we can observe that Subdue performs well for small data sets, but slows down when the data set grows more than 2500 vertices and 5000 edges. From this point HDB-Subdue performs better than Subdue. Also we can notice that EDB-Subdue performs better than HDB-Subdue because it does not perform pseudo-elimination, canonical ordering and other functions that HDB-Subdue does.

6.4.2 Dataset with multiple edges and cycles

In this section we have discussed the experiments performed on Subdue and HDB-Subdue for data sets in which cycles and multiple edges are present. EDB-Subdue is not

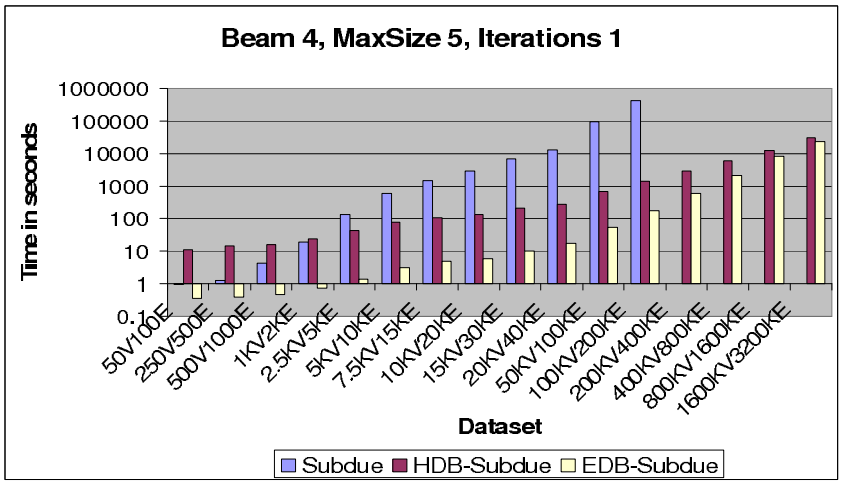

Figure 6.2 Graphs without cycles and multiple edges

compared because it does not handle multiple edges in the input. The embedded substructures are shown in Fig 6.3. The parameter settings used for running the experiments are shown in Tab 6.3.

Table 6.3 Parameter Settings

Parameters	Subdue	HDB-Subdue
MaxSize	5	5
Beam	4	4
Iterations	1	1

Subdue and HDB-Subdue discovered the same substructure and all the embedded instances. The discovered substructure for 1KV2KE dataset is shown below and the number of instances discovered for each dataset is shown in table Tab 6.4.

```
Dataset: 1KV2KE
Substructure discovered by Subdue:
Substructure: value = 1.15591, pos instances = 60, neg instances = 0
  Graph(4v,5e):
    v 1 v5
    v 2 v6
```

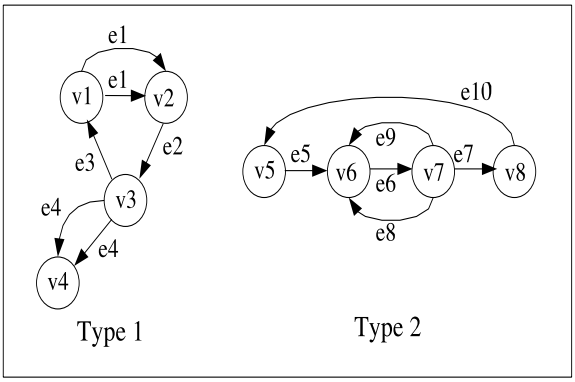

Figure 6.3 Graphs with cycles and multiple edges

```
v 3 v7
v 4 v8
d 1 2 e5
d 2 3 e6
d 3 4 e7
d 3 4 e7
d 4 2 e8

Substructure discovered by HDB-Subdue:
Vertex1name Vertex2name Vertex3name Vertex4name Vertex5name Vertex6name
-           -           v5          v6          v7          v8
Edge1name Edge2name Edge3name Edge4name Edge5name
e5        e6        e7        e7        e8
From_1 To_1 From_2 To_2 From_3 To_3 From_4 To_4 From_5 To_5 count DMDL
3      4    4      5    5      6    5      6    6      4    60    1.15964
```

From the comparison chart shown in Fig 6.4, we can see that HDB-Subdue starts to outperform Subdue when the input graph size crosses 2500 vertices and 5000 edges as before. We can also observe that, in the presence of multiple edges, a dataset of same size takes more time to complete than the dataset without multiple edges. This is due to the fact that the number of tuples generated and retained in each iteration is more in the presence of multiple edges and cycles.

Table 6.4 Instances discovered

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

6.4.3 Hierarchical Reduction

In this section we have shown the experiments conducted on Subdue and HDB-Subdue, for datasets which could be compressed at least two times to show the hierarchical reduction. The parameter settings used for running the experiments are shown in Tab 6.5. The embedded input graphs are shown in Fig 6.5.

Table 6.5 Parameter Settings

Parameters	Subdue	HDB-Subdue
MaxSize	5	5
Beam	4	4
Iterations	3	3

The substructures discovered by Subdue and HDB-Subdue in each iteration is shown below. We can see that the best substructure discovered by Subdue and HDB-

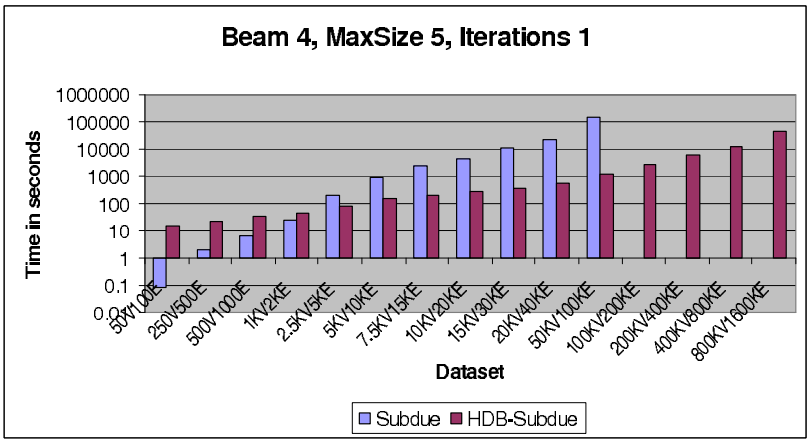

Figure 6.4 Graphs with cycles and multiple edges

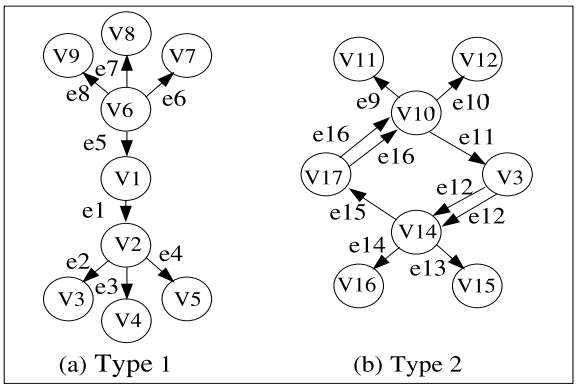

Figure 6.5 Sample graphs for Hierarchical reduction

Subdue in each iteration is not the same. This is because of the limitation in the accuracy of DMDL in HDB-Subdue. Since the DMDL formula is based on measuring the substructure size rather than measuring byte level storage space, the DMDL value tends to be the same for many substructures. Since the choice of the best substructure among the many substructures having ties in the highest DMDL value is arbitrary, the best substructure selected may not be the one selected by Subdue. We have validated the correctness of hierarchical compression of HDB-Subdue with many smaller examples and found it to be correct.

The choice of best substructure depends upon the value of BestSubs parameter also. This is because the number of best substructures selected after each expansion iteration is dependant on the value of BestSubs parameter. For all the experiments in this section, we have set the value of BestSubs to its default value which is *beam*. Hence BestSubs does not influence the choice of the best substructures for these experiments.

```
Substructure discovered by Subdue:
Iteration 1:
Substructure: value = 1.10491
                  pos instances = 40, pos examples = 1
                  neg instances = 0, neg examples = 0
  Graph(6v,5e):
    v 1 v1
    v 2 v2
    v 3 v3
    v 4 v4
    v 5 v6
    v 6 v9
    d 1 2 e1
    d 2 3 e2
    d 2 4 e3
    d 5 1 e5
    d 5 6 e8

Iteration 2:
Substructure: value = 1.06654
                  pos instances = 20, pos examples = 1
                  neg instances = 0, neg examples = 0
  Graph(6v,6e):
    v 1 v10
    v 2 v11
    v 3 v13
    v 4 v14
    v 5 v15
    v 6 v16
    d 1 2 e9
    d 1 4 e11
    d 3 4 e12
    d 3 4 e12
    d 4 5 e13
    d 4 6 e14
```

```
Iteration 3:
Substructure: value = 1.04056
                  pos instances = 40, pos examples = 1
                  neg instances = 0, neg examples = 0
  Graph(4v,3e):
    v 1 SUB_1
    v 2 v5
    v 3 v7
    v 4 v8
    d 1 2 e4
    d 1 3 e6
    d 1 4 e7

Substructures discovered by HDB-Subdue:

Iteration 1:
Vertex1name Vertex2name Vertex3name Vertex4name Vertex5name Vertex6name
v1          v2          v3          v6          v7          v8
Edge1name Edge2name Edge3name Edge4name Edge5name
e1        e2        e5        e6        e7
From_1 To_1 From_2 To_2 From_3 To_3 From_4 To_4 From_5 To_5 count DMDL
1      2    2      3    4      1    4      5    4      6    60    1.11649

Iteration 2:
Vertex1name Vertex2name Vertex3name Vertex4name
SUB_1       v4          v5          v9
Edge1name Edge2name Edge3name
e3        e4        e8
From_1 To_1 From_2 To_2 From_3 To_3 count DMDL
1      2    1      3    1      4    40    1.05522

Iteration 3:
Vertex1name Vertex2name Vertex3name Vertex4name Vertex5name Vertex6name
-           v10         v11         v12         v13         v14
Edge1name Edge2name Edge3name Edge4name Edge5name
e10       e11       e12       e12       e9
From_1 To_1 From_2 To_2 From_3 To_3 From_4 To_4 From_5 To_5 count DMDL
2      4    2      6    5      6    5      6    2      3    20    1.05411
```

From the experiments shown in Fig 6.6 we can see that HDB-Subdue starts to outperform Subdue when the input graph size crosses 2500 vertices and 5000 edges, similar to the graphs without and with cycles and multiple edges.

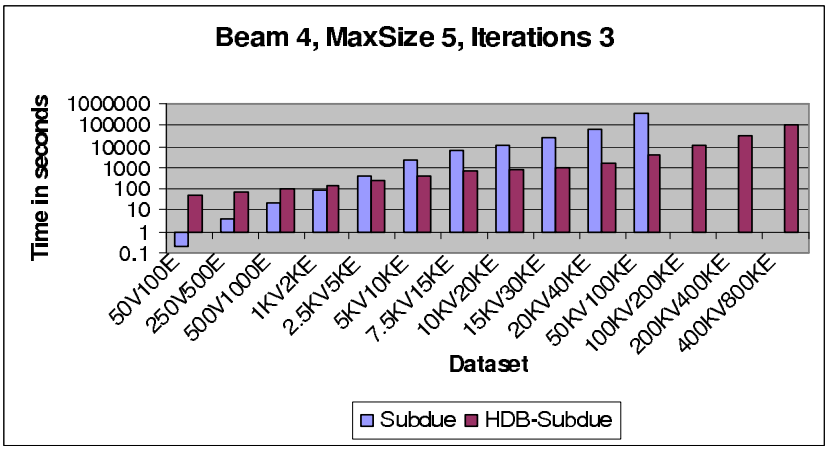

Figure 6.6 Hierarchical Reduction

6.4.4 Module Running times

The individual time taken by some of the key modules is shown in Fig 6.7. The X-axis shows the iteration numbers and the Y-axis shows the logarithm of running time in seconds. We used the multiple edges and cycles dataset shown in Fig 6.3 with 400,000 vertices and 800,000 edges (400KV800KE). The experiment was run with a beam of 4, for a maximum substructure size of 5 and for 1 iteration. From the figure we can see that the time taken for pseudo duplicate elimination dominates the total running time. Canonical label ordering takes almost 60-70% of the time taken by pseudo elimination. We can also observe that the time taken by all the modules decrease when we proceed from one iteration to the other. This is due to the fact that the number of substructure instances reduce as the size of the substructure increases (corresponds to higher iteration number).

6.5 Observations

Apart from conducting experiments on various datasets we also observed the performance of different types of queries and their alternative faster queries to do the same

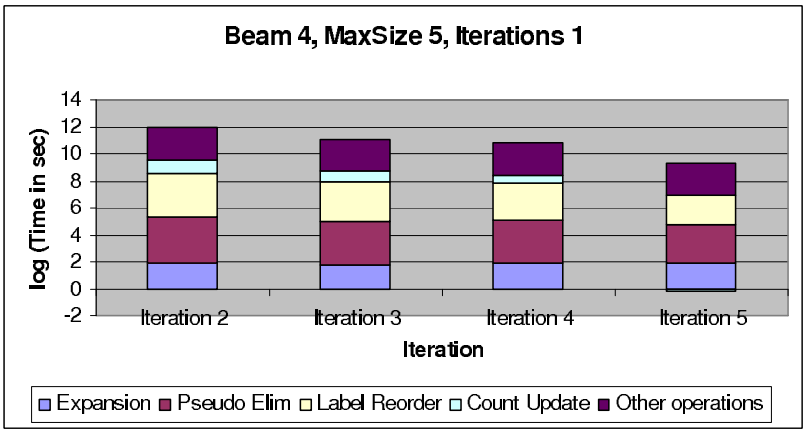

Figure 6.7 Module running times

function. We also explored the use of index on frequently used tables to improve the performance of HDB-Subdue.

6.5.1 Inplace deletes

The canonical table contains the instances and their duplicates ordered by the vertex numbers and the connectivity attributes. Each instance is identified by a unique identifier attribute called Id. To identify the duplicate instance we do a GROUP BY on the Id, vertex number and canonical attributes and select the instance with the maximum Id from each group and insert it in a table called instance_N_pseudo (The choice of max Id is arbitrary. One could choose min Id too.) To delete the duplicate instance, earlier we used an in-place delete query that deletes instances whose Id is not in the instance_N_pseudo table. This query took a longer time to complete. So we tried an alternate query that creates a canonicaltemp table by joining the canonical table with the instance_N_pseudo table on the Id attribute. Since this join produces only the instances we indented to retain, we can drop the canonical table and rename the canonicaltemp table as the canonical table. The in-place delete and join query are shown below.

```
IN-PLACE DELETE:
```

```
DELETE FROM canonical_N
WHERE Id NOT IN (SELECT Id FROM instance_N_pseudo)

ALTERNATIVE: Create a new table using a join query
INSERT INTO canonicaltemp (
 SELECT c.vertex1, .. c.vertexN+1, c.vertex1name, .. c.vertexN+1name,
        c.edge1 .. c.edgeN, c.edge1name, .. edgeN+1name, c.from_1, c.to_1,
        .. c.from_N, c.to_N, c.Id
 FROM   canonical_N c , instance_N_pseudo i
 WHERE  c.Id = i.Id)

DROP TABLE canonical_N
CREATE TABLE canonical_N AS SELECT * FROM canonicaltemp
```

The comparison of running time of HDB-Subdue using the In-place query versus creating a new table using join query is show in Tab 6.6. The table shows the results for two datasets, one with 200K vertices and 400K edges and the other with 400K vertices and 800K edges. We can observe from the table that the time taken by the pseudo duplicate elimination module in each iteration (the entries Pseudo_1, Pseudo_2, etc..) for join query is much lesser than that of the in-place delete query, corresponding to 85-90% improvement in iteration 2 and 50-60% on an average in all iterations.

6.5.2 Correlated Queries

A correlated subquery is a subquery that contains a reference to a table that appears in the outer query. During earlier phase of implementation of this thesis we used a correlated subquery to delete edges that appear only once, from the one-edge relation. The beamdel relation contains the edges that appear only once. So a delete

query identifies those substructures of one-edge that appear in beamdel table and deletes them. The query is given below.

```
DELETE FROM oneedge o
WHERE EXISTS (
      SELECT *
      FROM  beamdel b where b.vertex1name = o.vertex1name and
            b.vertex2name=o.vertex2name and b.edge1name=o.edgename)

DELETE FROM oneedge o
WHERE edgeno IN (SELECT edgeno
                 FROM beamdel)
```

The second query shown above is an alternate non-correlated query which was implemented later, to do the same function. The time taken in seconds for correlated Vs non-correlated queries with the percentage improvement is shown in Tab 6.7. From this we can conclude that it is useful to avoid correlated queries and adopt an alternate form without correlation (a join query) where possible.

6.5.3 Using Indexes

To speed up the execution of the queries, one option is to create an index on the tables that are repeatedly used. One of the tables on which an index was created is the one-edge table, because it is joined with instance_i during each expansion. We also created indexes on Sorted and the Sorted_Ext tables because we perform a 2n+1 way join during instance_i table reconstruction in pseudo-duplicate elimination step. Contrary to our expectations the performance of HDB-Subdue did not improve. We used the data set without multiple edges and cycles shown in Fig 6.1 with a beam of 4 and for one iteration.

The running times of the individual modules for 15KV30KE dataset and 50KV100KE dataset without and with indexing is shown in table Tab 6.8. From the total time, we can see that the performance dropped down by little more than 2%. Usage of index needs to be carefully analyzed in the future work.

6.6 Summary

This chapter discussed about using the graph generator to generate synthetic graphs and about the configuration and log file to execute and monitor progress of HDB-Subdue. We also compared Subdue, EDB-Subdue and HDB-Subdue for correctness and performance and observed that all of them produce the same output.

Table 6.6 Performance of In-place delete Vs Join query (Time Unit: Seconds)

	200KE400KE	200KE400KE	400KE800KE	400KE800KE
Module	In-place	New Table	In-place	New Table
CreatTab_1	4.24	13.2	5.72	3.98
Iter1_1	253.35	251.77	524.5	523.82
Inst_2	26.75	27.87	62.32	66.63
CreateIndex_2	58.3	53.84	119.06	117.18
Pseudo_2	9474.58	1103.37	36028	2229.14
LabReord_2	513.24	508.76	1015.51	1027.09
SubFold_2	5.35	5.37	10.78	12.11
CountUpd_2	3.37	6.28	12.85	13.74
SubFolInst_2	19.09	15	38.48	34.49
Inst_3	16.31	11.1	29.28	36.28
CreateIndex_3	12.71	14.26	30.63	27.01
Pseudo_3	676.7	284.51	2119.24	590.29
LabReord_3	177.1	171.83	363.21	362.72
SubFold_3	2.11	1.12	4.08	3.54
CountUpd_3	0.58	0.99	2.87	1.08
SubFolInst_3	32.71	35.66	72.97	73.71
Inst_4	18.81	17.3	42.83	42.59
CreateIndex_4	8.93	7.19	23.05	21.75
Pseudo_4	252.31	185.01	641.19	383.53
LabReord_4	59.87	57.6	118.77	116.47
SubFold_4	0.25	0.29	1.9	1.57
CountUpd_4	0.29	0.31	0.36	3.06
SubFolInst_4	37.67	36.93	77.84	76.13
Inst_5	10.05	9.73	22.48	21.32
CreateIndex_5	0.16	0.24	0.22	0.2
Pseudo_5	2.72	3.05	4.97	4.24
LabReord_5	2.42	2.24	3.22	3.32
SubFold_5	1.5	0.12	0.15	0.17
CountUpd_5	1.6	0.14	0.2	0.21
SubFolInst_5	33.56	32.78	70.45	72.08
Total	11740.65	2891.1	41508.69	5941.98

Table 6.7 Performance of Correlated queries (Time Unit: Seconds)

	15KV30KE	15KV30KE	50KV100KE	50KV100KE
Module	Non-Correlated	Correlated	Non-Correlated	Correlated
CreatTab_1	3.66	4.64	3.29	7.44
Iter1_1	19.25	77.94	57.38	694.36
Inst_2	0.95	1.06	6.82	4.89
CreateIndex_2	3.13	2.55	11.53	13.39
Pseudo_2	112.6	115.6	758.07	794.78
LabReord_2	34.6	33.06	112.74	113.22
SubFold_2	0.18	0.19	1.12	0.45
CountUpd_2	0.26	0.17	0.52	0.65
SubFolInst_2	0.59	0.81	2.47	4.64
Inst_3	0.89	1.15	5.85	5.28
CreateIndex_3	1.46	1.64	3.03	4.17
Pseudo_3	28.48	27.84	90.2	89.97
LabReord_3	12.63	15.3	52.78	53.49
SubFold_3	0.1	0.11	0.17	0.15
CountUpd_3	0.16	0.11	0.32	0.28
SubFolInst_3	0.84	1.95	8.07	7.85
Inst_4	1.51	1.36	5.04	5.16
CreateIndex_4	2.26	0.78	3.56	3
Pseudo_4	20.6	15.77	44.56	46.09
LabReord_4	3.71	3.8	13.02	13.33
SubFold_4	0.11	0.11	0.22	0.14
CountUpd_4	0.1	0.98	0.78	0.2
SubFolInst_4	1.24	1.13	8.54	9.85
Inst_5	0.92	0.77	2.16	2.16
CreateIndex_5	0.12	0.13	0.13	0.21
Pseudo_5	0.85	0.75	1.18	1.21
LabReord_5	0.48	0.54	0.8	0.85
SubFold_5	0.54	0.13	0.13	0.1
CountUpd_5	0.11	0.13	0.09	0.11
SubFolInst_5	0.96	0.97	8.43	8.14
Total	251.92	319.62	1204.41	1885.72

Table 6.8 Performance using Indexes (Time Unit: Seconds)

	15KE30KE	15KE30KE	50KE100KE	50KE100KE
Module	Without	With	Without	With
CreatTab_1	3.91	4.64	4.26	7.44
Iter1_1	70.52	77.94	670.81	694.36
Inst_2	0.86	1.06	10.84	4.89
CreateIndex_2	0	2.55	0	13.39
Pseudo_2	119.09	115.6	764.24	794.78
LabReord_2	32.61	33.06	116.35	113.22
SubFold_2	1.78	0.19	1.35	0.45
CountUpd_2	0.21	0.17	0.55	0.65
SubFolInst_2	2.6	0.81	2.25	4.64
Inst_3	1.65	1.15	6.34	5.28
CreateIndex_3	0	1.64	0	4.17
Pseudo_3	25.39	27.84	91.76	89.97
LabReord_3	13.31	15.3	49.79	53.49
SubFold_3	0.77	0.11	0.19	0.15
CountUpd_3	0.11	0.11	0.29	0.28
SubFolInst_3	1.15	1.95	6.11	7.85
Inst_4	1.83	1.36	10.39	5.16
CreateIndex_4	0	0.78	0	3
Pseudo_4	16.85	15.77	43.82	46.09
LabReord_4	6.95	3.8	16.18	13.33
SubFold_4	0.13	0.11	0.57	0.14
CountUpd_4	0.09	0.98	0.22	0.2
SubFolInst_4	1.29	1.13	9.83	9.85
Inst_5	1.94	0.77	9.2	2.16
CreateIndex_5	0	0.13	0	0.21
Pseudo_5	0.59	0.75	0.94	1.21
LabReord_5	0.47	0.54	0.89	0.85
SubFold_5	0.12	0.13	0.13	0.1
CountUpd_5	0.1	0.13	0.13	0.11
SubFolInst_5	0.99	0.97	6.39	8.14
Total	311.41	319.62	1844.1	1885.72

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis we have addressed several enhancements to the earlier graph-based data mining algorithm (EDB-Subdue), developed using relational databases. The different enhancements addressed in this thesis include, detection of all possible patterns including multiple edges using unconstrained expansion and performing hierarchical reduction. A scheme for eliminating duplicates generated during expansion was also designed and implemented. Changes were made to the substructure evaluation metric, DMDL, to account for the multiple edges and new substructure representation. The previous work had an option to select top beam substructures having the highest DMDL value. But the ties in the DMDL value was not taken into account. In this thesis we explored the use of Oracle's analytic functions to select the top beam substructures with all its ties. All the enhancements were performed using pure SQL statements to improve the performance of the algorithm. Graph mining was run on data sets that have 1600K vertices and 3200K edges. All the enhanced algorithms were validated and tested to achieve the desired scalability.

7.1 Future work

The current substructure expansion algorithm does not impose any constraint to avoid duplicate instance generation. It might be possible to design a constraint using the edge numbers to avoid duplicate instances. Then it should be possible to skip the duplicate elimination step. But still, we would need to do the label reordering step because the substructure frequency counting needs the labels to be in canonical order.

In the calculation of the substructure evaluation metric, DMDL, we use the counts of vertices and edges instead of bit representation as in MDL. Also for calculating the numerator of the DMDL formula, Value(S), we add up number of vertices and number of edges in the input graph. But in the denominator, for calculating Value(S) we add number of vertices in the substructure and non-zero rows parameter instead of the number of edges. This essentially allows us to simulate the MDL value in its trend (and not actual value) to validate the algorithm against the output produced by Subdue. In the future, DMDL could be modified to take the byte level storage space the substructure instances occupy. The storage space should be obtained at run time because all the text based columns (vertex and edge labels) are declared as VARCHAR.

In this thesis we tried to improve the performance of the queries by creating Indexes on frequently accessed tables. We created index on oneedge table because we join the InstanceIter table with oneedge table in each expansion iteration to grow substructures from smaller sizes to larger sizes. We also created indexes on Sorted table (which holds the vertices sorted on vertex numbers) and Sorted_Ext table (which holds the edge information sorted on connectivity attributes) because they are joined in *2n+1* way when we reconstruct the instance table in canonical order. However, we did not observe any improvement with the use of indexes. This perhaps need to be revisited to establish the role of indexes for this class of computation.

The current substructure discovery algorithm identifies only exact graph matches. That is, it requires all the instances of a substructure to have the same number of graph vertices and edges with matching edge and vertex labels. There are applications where a dissimilarity of few vertices can be tolerated. In our algorithm since we order the vertex and edge labels canonically, it would directly contribute to the inexact graph match, because most of the inexact graph match algorithms depend on canonical labeling. Another improvement would be to extend database graph mining to perform concept learning and

classification. Scalability and performance can be improved through partitioned and incremental approaches. The incremental approach will mine the data only on the new data that is added rather than mining the whole graph again after the addition is made.

REFERENCES

[1] T. Washio and H. Motoda, "State of the art of graph-based data mining," *SIGKDD Explor. Newsl.*, vol. 5, no. 1, pp. 59–68, 2003.

[2] P. Mishra and S. Chakravarthy, "Performance evaluation and analysis of k-way join variants for association rule mining," *BNCOD*, pp. 95–114, 2003.

[3] ——, "Performance evaluation of sql-or variants for association rule mining," *PAKDD Proceedings,Sydney*, pp. 288–298, DaWaK.

[4] D. J. Cook and L. B. Holder, "Graph-based data mining," *IEEE Intelligent Systems*, vol. 15, no. 2, pp. 32–41, 2000.

[5] J. Rissanen, *Stochastic Complexity in Statistical Inquiry Theory.* World Scientific Publishing Co., Inc., 1989.

[6] S. Chakravarthy, R. Beera, and R. Balachandran, "Database approach to graph mining," *PAKDD Proceedings, Sydney*, pp. 341–350, May 2004.

[7] R. Beera, "Relational database algorithms and their optimization for graph mining," Master's thesis, Department of Computer Science and Engineering, University of Texas at Arlington, May 2003. [Online].

[8] R. Balachandran, "Relational approach to modeling and implementing subtle aspects of graph mining," Master's thesis, Department of Computer Science and Engineering, University of Texas at Arlington, Dec 2003. [Online].

[9] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge," *Journal of Artificial Intelligence Research*, vol. 1, pp. 231–255, 1994.

[10] J. A. Gonzalez, L. B. Holder, and D. J. Cook, "Graph based concept learning," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press / The MIT Press, 2000, p. 1072.

[11] I. Jonyer, D. J. Cook, and L. B. Holder, "Discovery and evaluation of graph-based hierarchical conceptual clusters," *Journal of Machine Learning Research*, vol. 2, pp. 19–43, 2001.

[12] D. H. Fisher, "Knowledge acquisition via incremental conceptual clustering," *Mach. Learn.*, vol. 2, no. 2, pp. 139–172, 1987.

[13] K. Thompson and P. Langley, "Concept formation in structured domains," pp. 127–161, 1991.

[14] H. Bunke and G. Allermann, "Inexact graph matching for structural pattern recognition," *Pattern Recognition Letters*, vol. 1, no. 4, pp. 245–253, 1983.

[15] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 313–320.

[16] R. C. Read and D. G. Corneil, "The graph isomorph disease," *Journal of Graph Theory*, vol. 1, pp. 339–363, 1977.

[17] S. Fortin, "The graph isomorphism problem," in *Technical Report TR96-20, Department of Computing science, University of Alberta*, 1996. [Online]. Available: citeseer.ist.psu.edu/fortin96graph.html

[18] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 721.

[19] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns by prefix-projected growth," in *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 215–224.

[20] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from graphs: Mining graph data," *Mach. Learn.*, vol. 50, no. 3, pp. 321–354, 2003.

[21] S. Feuerstein and B. Pribyl, *Oracle PL/SQL Programming, 4th Edition*. O'Reilly, August 2005.

[22] S. Bellamkonda, T. Bozkaya, B. Ghosh, A. Gupta, J. Haydu, S. Subramanian, and A. Witkowski. (2000) Analytic functions in oracle 8i. [Online]. Available: www-db.stanford.edu/dbseminar/Archive/SpringY2000/speakers/agupta/paper.pdf

[23] [Online].

[24] [Online]. Available: http://ailab.uta.edu/subdue/datasets/subgen.tar.gz

BIOGRAPHICAL STATEMENT

Srihari Padmanabhan was born in Coimbatore, India. He received his B.E. degree from Coimbatore Institute of Technology, India, in May 2003. In the Fall of 2003, he started his graduate studies in Computer Science at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in December 2005.