A RELATIONAL DATABASE APPROACH FOR FREQUENT SUBGRAPH MINING

by

SUBHESH KUMAR PRADHAN

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
August 2006

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 Mr. David Levine and Dr. Lawrence Holder for serving on my defense committee.

I would like to thank Raman Adaikalavan and Aditya Tenlang for helping me throughout in times of need. Thanks are due to all ITLab friends for their help 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 Air Force (grants F30602-01-2-0570) and NSF (grants EIA-0216500)

June 26, 2006

ABSTRACT

A RELATIONAL DATABASE APPROACH FOR FREQUENT SUBGRAPH MINING

Publication No. ______

SUBHESH KUMAR PRADHAN, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Sharma Chakravarthy

Data mining aims at discovering interesting and previously unknown patterns from data sets. Further more, graph-based data mining represents a collection of techniques for mining the relational aspects of data represented as a graph. Complex relationships in data can be represented using graphs and hence graph mining is appropriate for analyzing data that is rich in structural relationships. Database mining of graphs, on the other hand, aims at directly mining graphs stored in a database using SQL queries. Several SQL-based mining algorithms have been developed successfully and their efficiency and scalability have been established. One of them is HDB-Subdue which uses SQL-based approach for mining for the *best substructures* in a graph using the MDL (Minimum Description Length) Principle. Determining frequent subgraphs is another type of graph mining for which only main memory algorithms exist currently. There are many applications in social networks, biology, computer networks, chemistry and the World Wide Web that require mining of frequent subgraphs. Also, data in most applications are directly stored in a database. Hence, there is a need for developing a SQL-based approach for frequent subgraph mining that scales to very large data sizes.

The focus of this thesis is to apply relational database techniques to support frequent subgraph mining over a set of graphs. Our primary goal is to address scalability of graph mining to very large data sets, not currently addressed by main memory approaches. Unlike the main memory counter parts, this thesis addresses the most general graph representation including multiple edges between any two vertices, and cycles. In the process of developing frequent subgraph mining over a set of graphs, the substructure representation of HDB-Subdue has been leveraged and extended. An algorithm is presented for frequent subgraph mining over a set of graphs. We also present an algorithm for pseudo duplicate elimination that is more efficient than the one used in the previous approach (HDB-Subdue).

This thesis also presents an efficient approach to infer structural relationships from relational data to facilitate graph mining (either the best subgraph or frequent subgraphs). The approach developed for the task infers the entity-relationship model for the database using the table instances along with primary and foreign key constraints and generates the instances of the graphs from the populated instances of the relations. The primary focus of this approach is the portability of the system across various relational database platforms and to minimize memory/space requirement. As part of this task the following issues were addressed: i) representation of an individual relations as a graph (template), ii) representation of multiple relations as a graph (based on foreign key constraints), iii) alternative representations for ii), and iv) an algorithm which uses the most appropriate sequence in which the relations are processed to generate graph instances to minimize memory/space requirements.

TABLE OF CONTENTS

Chapter

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the advent of automated data collection tools, our ability to generate and collect data have increased rapidly in the last few decades. This explosive growth in data has opened the possibility of extracting useful information and knowledge from the data. Data mining helps in discovering non-trivial patterns on a stored data and hence more research is being done in this field to extract useful information from large amounts of collected data. Furthermore, many algorithms are being developed for graph mining to discover interesting patterns on data having inherent structural relationship. Representation of complex relationships in data can be readily done using graphs and since graph mining makes use of structural relationship to discover interesting patterns, graph mining has emerged as an appropriate solution to mine over data that can be represented as graphs. In order to address the scalability issues of graph mining, several SQL-based mining algorithms have been developed successfully and their efficiency and scalability have been established. These SQL-based algorithms have been developed for mining best substructures in a graph. However, for determining frequent subgraphs which is another mining graphS, currently only main memory algorithm exist. There are many applications in social networks, biology, computer networks, chemistry and the World Wide Web that require mining of frequent subgraphs over very large data sets and these main memory algorithms do not scale very well for large data sets. Hence, there is a need for developing SQL-based approach for frequent subgraph mining that scales to very large data sizes. In the following paragraphs we briefly discuss different graph mining algorithms that motivated the development of a SQL-based approach for frequent subgraph mining.

Subdue [1] is one of the early graph mining algorithms that detects the best substructure using the minimum description length principle [2]. It can also mine for interesting concepts, detect anomalies, and similarities between graph structures. The minimum description length principle [2] 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.

FSG [3] is another main memory algorithm that mines over a set of graphs to discover frequent subgraphs. It uses canonical labeling to determine the subgraph isomorphism. FSG assumes the simple undirected graph representation without multiple edges (between two vertices) or cycles. Hence, FSG cannot mine over more general form of directed graphs, graphs with multiple edges, and cycles.

Main memory data mining algorithms typically face two problems with respect to scalability. Input 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. Since, graph mining as other data mining domains assumes massive data, it is natural to explore the possibility of utilizing database technology for graph mining purpose as it provides the advantage of scalability. However, applying database for graph mining is not easy. Representation of a graph, generation of larger subgraphs, checking for exact and inexact matches of subgraphs in relational database is not straight forward.

DB-Subdue [4, 5]), SQL-based version of Subdue is the first attempt at using relational database approach for graph mining. DB-Subdue has been developed for the DB2 database. It explains the mapping of substructures to tuples in the database. The goal of DB-Subdue 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 1,600,000 edges). Although DB-Subdue solved the scalability problem, it did not deal with the 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 dID not distinguish between two overlapping substructures having the same number of edges and vertices and the same number of instances. Host variables were used for exchanging data (extracted using cursors) between the database and the host programming language. In the version of DB2 used, an array of host variables cannot be declared 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 adequate.

EDB-Subdue [4, 6] 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 the 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 that vertex. 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.

HDB-Subdue [7] extends EDB-Subdue to handle multiple edges, cycles, and hierarchical reduction to deal with a general graph. HDB-Subdue uses unconstrained substructure expansion with duplicate elimination, to explore all possible expansions and expand multiple edges. HDB-Subdue introduces connectivity attributes which have complete information about the edges with vertex number invariant and connectivity information. SQL-based analytic functions were used to implement the beam. HDB-Subdue was able to discover best substructures using frequency count or DMDL values within a graph. However, HDB-Subdue did not support mining over a set of input graphs to discover frequent subgraphs. The graph representation of HDB-Subdue was not sufficient to incorporate frequent subgraph mining.

As mentioned earlier there are domains such as social network, science, business and others where graph mining can be implemented to extract interesting patterns. On the other hand, there is an increasing trend of storing data in relational database in these domains. Even though storing data as instances of relations can be easily automated, the structural relationship embedded within the data is not explicit. However, these structural relationships embedded in this data set is needed to infer similarities of patterns, how these patterns change over a period of time and so on. Hence, there is a need to infer structural relationship embedded within this form of data for the graph-based mining purpose. In other words, transformation of data from relational database to graph-based domain is required to facilitate graph-based mining on the data. Transformation of data from relational database to graph involves more practical aspect with focus on processing time, space/memory efficiency and correct interpretation of data. Since, the transformed data will serve as input to graph mining algorithms, preferred characteristics of the transformed data would be minimum memory/space requirement to represent the transformed data.

This thesis addressed two problems related to graph mining: i) To develop a new algorithm for frequent subgraph mining using an SQL-based approach and ii) To convert relaitonal database into an appropriate graph form utilizing the information embedded in the represenation (such as foreign key) into a graph on which differen forms of mining can be applied.

The primary focus of this thesis is to apply relational database techniques for frequent subgraph mining. We extended the graph representation of HDB-Subdue to incorporate mining over a set of graphs. In this thesis we present DB-FSG, a relational database approach to mine frequent subgraph over a set of graphs. DB-FSG takes support value from 0 to 100 percent and max substructure size as input and calculates the minimum support count as: *support X number of graphs/100.* Then DB-FSG mines over the set of graph and returns all the subgraphs with size less than or equal to max substructure and that has frequency(across the set of graphs) more than or equal to support count. We also present a new approach for pseudo duplicate elimination that is more efficient than the approach presented by HDB-Subdue.

In this thesis we also present an efficient system DB2GraphGen to transform relations of database to graph. DB2GraphGen is an efficient system that infers the Entity-Relationship model (ER model) [8] within a relational database using primary key and foreign key constraints and transforms the relations to a graph utilizing the ER model .The graph representation used in DB2GraphGen is derived from conceptual graph representation [9, 10]. This graph representation requires less space (less number of edges and vertices) than the conceptual graph[9, 10].

The rest of this thesis is organized as follows. CHAPTER 2 discusses the background and related work. CHAPTER 3 provides the overview of DB-FSG. CHAPTER 4 discusses the design issues for DB-FSG. CHAPTER 5 presents implementation details of all the issues that are explained in the design chapter and CHAPTER 6 provides the experiment results of the DB-FSG. CHAPTER 7 describes our approach of inferring structural relationships from a relational database i.e. DB2GraphGen system and CHAPTER 8 presents it's performance evaluation. CHAPTER 9 concludes the thesis and identifies potential future work.

CHAPTER 2

RELATED WORK

In recent years data mining has attracted a great deal of attention. Due to the abundance of data in many application domains, the research in data mining has grown rapidly. Extensive research in the field of data mining has been done to develop the challenging real-life applications [11]. In this section, we briefly describe some of the ongoing research activity in the area of graph mining. Some of the work outlined here include Subdue, FSG (Frequent SubGraph discovery), gSpan (graph-based Substructure pattern mining), DB-Subdue, EDB-Subdue and HDB-Subdue.

Many times, data collected as transactions in a relational database need to be transformed into a graph representation before graph mining algorithms can be applied. The structural relationship embedded in the data need to be inferred to mine the data. Inferring structural relationship from relational database is seemingly A new concept as large amounts of data is collected in relational database from banking, telephone , and other transactions. As graph mining is only being applied to transactional data with embedded structural relationships (in contrast to transactional data such as point of sales that do not have structural relationship) there is hardly any prior works in this field. However, there are ongoing research that utilizes the graph representation of relational database to serve bigger purpose like information retrieval or Graph-Based mining. RDB2Graph and BANKS are among those works.

2.1 FSG - Frequent Subgraphs

FSG [3] finds all connected subgraphs that appear frequently in a large database. In FSG, a subgraph is considered as a frequent subgraph when it has the frequency percentile (across the set of graph) greater than or equal to the support percentile specified by the user. FSG models objects as undirected labeled graphs where each vertex represents an entity and each edge represents relation between two entities. FSG is equivalent to association rule mining of frequent item sets. If a set of transactions is modeled as a set of graphs then FSG can discover frequent subgraphs where each vertex corresponds to an item and each edge signifies the coexistence of two items in a transaction. 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 has frequency(across the set of graph) greater than the user specified support value.

FSG discovers frequent subgraphs in two steps

1. Candidate Generation
   (a) Two frequent subgraph of size n having same core of size n-1 is combined to generate probable candidates of size n+1
   (b) Canonical labeling of the candidates are referred to avoid duplicates
   (c) Downward closure property constraint is imposed to prune false candidates
2. Frequency Counting and SubGraph pruning
   (a) Canonical labels are used to check the subgraph isomorphism
   (b) Subgraphs having frequency greater or equal to the user specified support value are retained . Only these subgraphs participate in further expansion of subgraphs

FSG represents graphs as sparse adjacency matrix. The heart of FSG is the canonical labeling of subgraphs as it is used to avoid duplicate candidates as well as to check graph isomorphism for frequency counting.

2.1.1 Canonical label

A canonical label is a unique code of a given graph [12, 13] constructed by flattening the adjacency matrix representation of a graph. Since, same graph can be represented in different sequence of vertices in adjacency matrix, more than one label can be generated from adjacency metric representation of the graph. Hence, to compute a canonical label of a graph, all the permutations of the vertices of the graph are tried to see which order of vertices gives the lexographically minimum label. To narrow down the search space, the vertices are partitioned by their degrees and labels using a technique called vertex invariants [12]. Then all possible permutations of vertices inside each partition are generated. Further more, two isomorphic graphs will have same canonical labels.

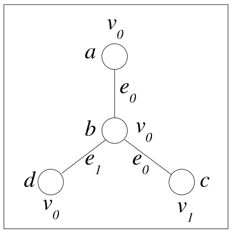


Figure 2.1 Frequent Subgraphs

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 dacbe$_1e_0e_0$, while the right one has a label of adcbe$_0e_1e_0$. By string comparison, the label of the right matrix becomes the canonical label.

2.2 gSpan - graph based Substructure pattern mining

gSpan [14] like FSG is a main memory algorithm for frequent subgraph mining. However, gSpan avoids the costly steps of FSG such as candidate generation and pruning false positives. Similar to FSG, gSpan generates unique code for each graph known as

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

DFS (Depth First Search) code. gSpan was shown to perform better than FSG because it combines the candidate generation and the isomorphism checking into a single phase. gSpan also tries to address the scalability issue of main memory. If the entire graph does not fit in main memory, graph-based data projection as in PrefixSpan [15] is applied and then gSpan is performed.

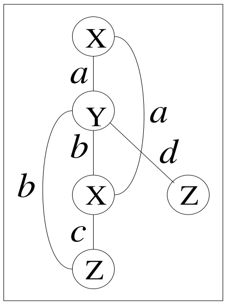


Figure 2.2 gSpan

2.2.1 DFS Coding

DFS code is generated from the DFS tree model of a graph. Since, a graph can have many DFS tree, gSpan selects the DFS tree that results into the lexographically minimum DFS code hence, DFS code will be unique to each graph. A DFS code for any DFS 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. Table 2.3 shows the corresponding DFS codes for Figure 2.2. Thus the problem of mining frequent connected subgraphs is equivalent to mining their corresponding minimum DFS codes.

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)

2.3 Subdue

Subdue is one of the earliest works in graph based data mining. It uses adjacency matrix for representing labeled graphs. The graph representation of Subdue can represent multiple edges and cycles and it can handle directed as well as undirected graphs. In Subdue, a connected subgraph (without considering vertex numbers) is known as a substructure and a set of vertices and edges that have the same vertex labels, edge labels and edge directions as that of the substructure are known as instances of the substructure. Subdue discovers substructures based on Minimum Description Length(MDL) principle [2, 16],an information theoretic metric on adjacency matrix. Once, the substructure is discovered, the substructure is used to compress the graph replacing the instances of the substructure with pointer to the newly discovered substructure.

2.3.1 Inexact Graph Match

Apart from doing exact graph match, Subdue [1] has the capability of matching two graphs, differing by the number of vertices specified by the threshold parameter, inexactly. Branch and Bound algorithm and Hill Climbing algorithms were developed by enhancing the Bunke and Allerman [17] inexact graph algorithm. These inexact graph match algorithms were shown to be in polynomial time. The worst case of these algorithm is given in equation 2.1 where v is the number of vertices, s is the bound for number of structures to be considered and x is the number of partial maps to be considered.

$$(\sum_{i=1}^{s} i * ((v-1) - (i-1))) * (v(s-1) * x). \tag{2.1}$$

2.3.2 Supervised Learning

The Subdue Concept Learner (SubdueCL) [18] is an extension to the Subdue system aimed at supervised pattern discovery. Positive and negative examples of a phenomenon are provided to SubdueCL and it searches for a pattern that compresses the positive graphs, but not the negative graphs.

2.4 DB-Subdue

DB-Subdue is a first attempt to implement database approach for graph mining. Db-Subdue mines directly on the graph represented as relations in the database. DB-Subdue evaluates the best substructure of a graph by counting the frequency of the instances of the substructure within the graph. DB-Subdue performs expansion of a graph using join operation and frequency counting is done by group by operation. It uses standard SQL statements and indexing techniques to improve the performance of the algorithm. The DBMS version of Subdue (DB-Subdue) [4, 5] was developed for the DB2 database using the C language. The experiments show that it can mine graphs with millions of vertices and edges without main memory becoming a bottleneck. DB-Subdue was able to mine data sets with 800K vertices and 1600K edges. DB-Subdue has several approaches for graph mining, such as the cursor-based approach, the UDF 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.

2.5 EDB-Subdue

EDB-Subdue [6, 4] is an extension of DB-Subdue and was developed in an effort to handle certain aspects of graph such as cycles, overlap and inexact graph match which were not considered in DB-Subdue. EDB-Subdue uses DMDL (Database Minimum Description Length) which is an database equivalent of MDL principle to evaluate best substructures. 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. 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 also explored the possibility implementing inexact graph match by using cursors and invoking the C subroutine that compares 2 subgraphs for a threshold and return a boolean value. However, EDB-Subdue has to call the C subroutine m^2 times where m is the number of subgraphs and the complexity of the inexact graph match routine is given in equation 2.1 Hence, the inexact graph match is not scalable enough to handle very large graphs.

2.6 HDB-Subdue

HDB-Subdue [7] is a modification of DB-Subdue and EDB-Subdue. In order to represent more general form of graphs including cycles and multiple edges between vertices, HDB-Subdue extends the graph representation of EDB-Subdue by introducing connectivity attributes. Since, the constrained expansion of substructures in EDB-Subdue was not able to produce all the variations of a substructure, HDB-Subdue allows unconstrained expansion of substructures. However, unconstrained expansion of substructures leads to pseudo duplicate instances of substructures. That is, same instances but expanded or enumerated in a different order. HDB-Subdue identifies the pseudo duplicates by ordering substructure instances by vertex numbers and connectivity map. Then it eliminates the pseudo duplicates. Similarly to count the frequency of instances of a substructure, it first arranges the instances by vertex labels and connectivity attributes.

HDB-Subdue allows hierarchical reduction of graphs. After best substructure is identified for an iteration, HDB-Subdue compresses the graph by replacing all the best substructures of the graph by a vertex.

2.7 RDB2Graph

RDB2Graph [19] provides an approach for the transformation of a relational database into a graph for mining purposes. It infers a conceptual graph [9, 10] from a relational database and generates the instances of the graph from the populated instances of the relations. The graph generated from RDB2Graph [19] is used as input to Subdue [1] for graph-based mining. RDB2Graph [19] employs only the Oracle database for graph representation. Hence considerable modifications are needed in order to generalize this approach and apply it to any other relational database management system. RDB2Graph does not consider constraining factors of limited resources such as memory space and processing speed; neither does it address the problem of conversion of a relational database having composite primary keys and foreign keys. Even though RDB2Graph [19] works for small and simple Oracle databases, it is not able to transform complex, large database instances.

2.8 BANKS

BANKS [20] provides keyword-based search on relational databases, together with data and schema browsing. BANKS [20] enables users to extract information in a simple manner without any knowledge of the schema or any need for writing complex queries. BANKS [20] models tuples as nodes in a graph, connected by links induced by foreign key and other relationships. Answers to a query are modeled as rooted trees connecting tuples that match individual keywords in the query. Even though BANKS [20] uses foreign key and primary key constraints for graph representation of relations, it represents whole tuples of a relation as a vertex. Hence, this representation does not allow detection of interesting patterns on the basis of individual attributes.

CHAPTER 3

OVERVIEW OF DB-FSG

In this chapter we will describe the relational database approach for frequent subgraph mining. We will discuss how a set of graphs and their subgraphs can be represented in a relational database. Then we will describe the expansion of as n- edge sub-graph to an n+1 edge sub-graph using sql queries. We will also discuss frequency counting of sub-graphs across the graphs. Finally, we will discuss the limitations of DB-FSG.

First, we present some common terminology that will be used through out this thesis. In this this, number of edges in a sub-graph corresponds to the size of the subgraph. 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. Frequency of a substructure denotes the count of the similar (isomorphic) substructures across the set of graphs. Even when there are multiple occurrences of a subgraph within a graph , the count across the graphs includes only one count for each distinct graph in the input. Note that the input to Db-FSG is a set of graphs.

3.1 Graph Representation

In order to accommodate mining over a set of graphs we extend the graph representation of HDB-Subdue [7]. Since data are stored in relational database as relations, we need to represent graphs as tuples in a relation. Normally graphs are re presented as A set of edges and vertices. DB-FSG represents graphs using two relations: i) A vertex table and ii) An edge table. The vertices of graphs are stored in a relation called the *vertex* table and the edges are stored in a relation termed the *edge* table. For the set of graphs shown in Fig 3.1 the corresponding *vertex* and *edge* tables are shown in Tables 3.1 and Tab 3.2, respectively. Graph Id(in short GID) attribute in the tables helps to identify the edges and vertices belonging to the same graph. In other words, each graph is given a unique identification that is also stored as part of the representation.

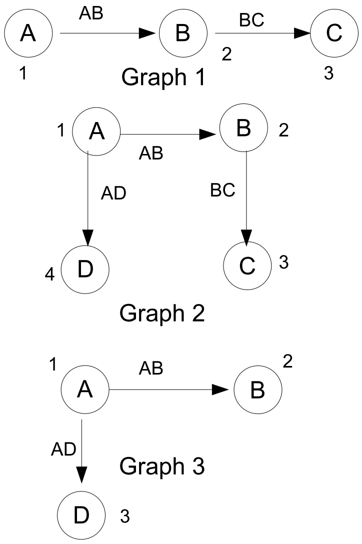


Figure 3.1 Example Graph

As the edge table does not contain information about vertex labels, tuples of edge table cannot represent substructures of size one. Hence, we create a new relation called *oneedge* by joining the vertex and the edge tables as shown in Table 3.3. The *Oneedge* table will contain all the instances of substructures of size one as tuples. For a one edge substructure, the edge direction is always from the first 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 will be using for generating higher size substructures. For each edge in the oneedge table

Table 3.1 Vertex table

Vertex No	Vertex Name	Graph Id
1	A	1
2	B	1
3	C	1
1	A	2
2	B	2
3	C	2
4	D	2
1	A	3
2	B	3
3	D	3

Table 3.2 Edge table

Vertex 1	Vertex 2	Edge Label	Graph Id
1	2	AB	1
2	3	BC	1
1	2	AB	2
1	4	AD	2
2	3	BC	2
1	2	AB	3
1	3	AD	3

we assign a unique identifier called THE edge number. We will discuss the usefulness of edge numbers later in this chapter. Since, the edges having frequency less than minimum support value will not expand to substructures that satisfy the minimum support, we remove those edges from *oneedge* (those that have frequency less than the minimum support value across the set of graphs).

3.2 Unconstrained Expansion of a Subgraph

In order to count the isomorphic substructures across graphs, we need to systematically generate subgraphs of increasing size in all the input graphs and count them. To expand a one-edge substructures to a two-edge substructures we join *oneedge* rela-

Table 3.3 Oneedge table

Vertex1	Vertex2	EdgeNo	EdgeLabel	Vertex1 Name	Vertex2 Name	GraphID
1	2	1	AB	A	B	1
2	3	2	BC	B	C	1
1	2	3	AB	A	B	2
2	3	4	BC	B	C	2
1	4	5	AD	A	D	2
1	2	6	AB	A	B	1
1	3	7	AD	B	D	1

tion with itself on matching vertices. To insure that the expansion is done within the same graph we impose a constraint that GID of both one edged substructure should be same. 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 order to avoid expansion of instances on edges that are already present (remember that our approach unlike FSG, handles multiple edges and cycles), we impose the rule that the new edge being added should not have the same edge number as the edge already present in the substructure instances. The queries used for unconstrained expansion of substructures are described in section 3.3. 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 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+3 attributes are needed to represent an n-edge substructure. 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.

Instance_2 relation for the graph in Fig 3.1 is shown in Table 3.4.

Table 3.4 Instance_2

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	F1	T1	F2	T2
1	2	3	A	B	C	1	2	AB	BC	1	1	2	2	3
1	2	3	A	B	C	3	4	AB	AC	2	1	2	2	3
1	2	4	A	B	D	3	5	AB	AD	2	1	2	1	3
1	2	3	A	B	D	6	7	AB	AD	3	1	2	1	3

Table 3.5 Sub_Fold_2

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
A	B	C	AB	BC	1	2	2	3
A	B	C	AB	AC	1	2	2	3
A	B	D	AB	AD	1	2	1	3
A	B	D	AB	AD	1	2	1	3

The edge label attribute, which is just a text string, does not give any information about the vertex it is originating or terminating from. As the name indicates, it is a label associated with the edge. The direction and the vertices associated with the edge can be obtained from the connectivity attributes Fi and Ti, which tell us that the edge i originates at Fi and terminates 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 attribute numbers from the left whose value correspond to the vertex numbers. Let us consider the third substructure in the instance_2 table. For edge AB the value of F1 is 1 indicating that the edge originates from the vertex number stored in V1 which is 1 and T1 is 2 meaning it terminates at vertex number stored in V2 which is 2. Similarly for edge AD F2 is 1 which indicates that the edge originates from vertex 1 and T2 is 3 which means that the edge terminates at vertex 3.

To calculate the frequency of a substructure we identify similar substructures across the set of graphs and count them. To insure only one instance of substructure per graph is considered while evaluating frequency, we project distinct vertex labels, edge labels, GID and connectivity attributes and store them in relation dist_n. Projection on vertex labels, edge labels and connectivity attributes in the dist_n and group by on the same attributes gives the frequency of each substructure across all the input graphs. Then we stored the substructure with frequency satisfying minimum support in relation Sub_Fold_n. Sub_Fold_2 relation is shown in Table 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, edge numbers, and graph id whereas substructures are described without vertex numbers, edge numbers, and graph id. In order to get ALL the instances corresponding to the substructures in Sub_Fold_n, we join the 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.

One of the halting conditions for the expansion process is the user specified parameter termed as MaxSize. Once the frequent substructures OF MaxSize are discovered the expansion terminates. Another halting condition for the expansion process is when no substructures are generated during expansion. This can happen if all the substructures generated have frequency less than the minimum support or if the graph size of all graphs are less than MaxSize.

Table 3.6 Instanceiter_2 with support value=20%

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	F1	T1	F2	T2
1	2	3	A	B	C	1	2	AB	BC	1	1	2	2	3
1	2	3	A	B	C	3	4	AB	AC	2	1	2	2	3
1	2	4	A	B	D	3	5	AB	AD	2	1	2	1	3
1	2	3	A	B	D	6	7	AB	AD	3	1	2	1	3

3.3 Generalization of Expansion

The InstanceIter_2 relation shown in Tab 3.6 contains all the instances of the two edge substructure that have been chosen for further expansion. A 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 all possible ways. In order to make an extension, one of the vertices in the substructure has to match a vertex in the oneedge relation. Queries used for extending a single edge substructure in all possible ways are given below. The unconstrained expansion is extremely important as it is not possible to generate subgraphs in a constrained manner. Since the input graph is arbitrary, and every substructure has to be generated and checked, it is not possible to constrain the generation process in any way. This entails that we may generate the same substructure in many ways and those duplicate substructures have to be identified and handled correctly.

```
/* 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.gid, 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
                and o.GID=s.GID)

/* 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.gid, 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
                 and o.GID=s.GID)
```

```
/* 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.gid, 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
                and o.GID=s.GID)

/* 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.gid, 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 and
                 o.GID=s.GID)

/* 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.gid,
          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 and o.GID=s.GID)

/* 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.gid,
          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 and o.GID=s.GID)

/* 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.gid,
           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 and o.GID=s.GID)

/* 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.gid,
           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 and o.GID=s.GID)
```

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. Therefore as in HDB-Subdue [7], we mark the repeating vertex number by vertex invariants 0 and the corresponding vertex label is marked as '-'. Marking of repeating vertices avoids redundant expansion on same vertex. 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.

3.4 Limitations of this approach

There are some limitations of representing a subgraph as a tuple of a relation. The approach presented in DB-FSG 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+3 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.

3.5 Summary

In this chapter we discussed the representation of a graph in a database using relations and how we prune the instances with frequency less than minimum support. Also we discussed a generalization for substructure expansion covering all possible expansion types and we also identified the limitations of representing a substructure as a tuple in relational databases.

CHAPTER 4

DESIGN ISSUES OF DB-FSG

This chapter discusses the design issues involved in relational database approach for frequent subgraph mining. First we will provide algorithm for DB-FSG. Then we will explain major of the aspects involved in DB-FSG algorithm.

4.1 DB-FSG Algorithm

DB-FSG is a sql based approach to detect frequent substructure from a set of graph that is stored in tables of relational database. As mentioned in chapter Overview of DB-FSG, the vertex information and edge information of the input graphs are assumed to be stored in vertex table and edge table respectively. The algorithm starts by creating one edge substructure instance by joining vertex table and edge table as shown in the step 1 of algorithm 1. Then the one edge instances with the frequency less than the user specified support value are pruned. Then the remaining one edge instances are expanded to two edge instances and the two edged substructure instances having frequency less than the

Algorithm 1 DB-FSG Algorithm

```
 1: Create oneedge (instance_1) table by joining vertex table and edge table
 2: Remove the edges with instance count less than support from the oneedge table
 3: for n=2 to MaxSize do
 4:    Join instance_(n-1) with oneedge table to generate instance_n
 5:    Eliminate pseudo duplicates from instance_n table
 6:    Canonically order instance_n table on vertex labels
 7:    Project distinct vertex label, edge label and gid to obtain one instance per substructure for each graph and store
       in dist_n table.
 8:    Group dist_n table by vertex label and edge label to obtain substructures and its count
 9:    Retain only the instances of substructure satisfying support and store it in instance_n table
10:    If there are no instances of substructure satisfying support then stop
11: end for
```

support value are pruned. As shown in the step 3 to 11 of algorithm 1, the expansion and pruning of sub-graphs continues till user specified max size is reached or till the subgraphs cannot be expanded any further. There are various major aspects involved in order to perform each step of the algorithm. For instance in order to handle most general form of graph (with cycles and multiple edges) new substructure instance representation is required. And in order to get all the possible sub-graphs of size n expanded from the sub-graphs of size n-1, unconstrained expansion is required as shown in step 4 of the algorithm 1 . Due to the unconstrained expansion the same substructure may be generated in many ways. Hence, pseudo duplicate elimination is required to remove such duplicates as mentioned in step 5 of the algorithm 1. Also, due to unconstrained expansion similar substructure instances may be generated in different ways and canonical ordering is performed in step 6 of the algorithm 1 to identify such substructures instances. Similarly to get the correct frequency of the substructures substructure counting and pruning is done in step 7, 8 and 9 of the algorithm 1.

The major aspects *new substructure instance representation*, *unconstrained expansion*, *pseudo duplicate elimination*, *canonical label ordering* and *frequency counting and substructure pruning* are discussed in the following sections.

4.2 Need for new substructure instance representation

HDB-Subdue [7] represents a substructure as a tuple of a relation. Along with vertex attributes and edge attributes, HDB-Subdue stores connectivity information of each edge in the substructure. Graph representation of HDB-Subdue also has the provision to represent cycles and multiple edges. The repeating vertex number of a cycle or a multiple edge is marked by '0' and the corresponding vertex label is marked by '-'. The marking of repeating vertices avoids redundant expansion on same vertex. Some of the substructure instances of size 3 in HDB-Subdue for the Fig 4.2 is shown in table 4.1. This form of representation can represent most general form of a graph including cycles and multiple edges. However, this representation is not sufficient to represent a set of graphs. For example, this representation cannot represent a set of graphs in Fig 4.2. In Subdue, the goal was to find the best substructure to compress the input–be it a single graph or a set of graphs (forest). Hence there was no need to distinguish between disjoint input graphs as the goal was to compress the entire input. In contrast, in FSG, we need to distinguish between the graphs in which the same substructure appears. Hence, we need one more attribute to denote which graph a substructure belongs to. So, we extend the graph representation of HDB-Subdue to incorporate the Graph Id (GID) as attribute. Each graph is assigned a unique GID and all the substructures belonging to same graph will have the same GID. New substructure instance representation of size two for Figure 4.2 is shown in table 4.2.

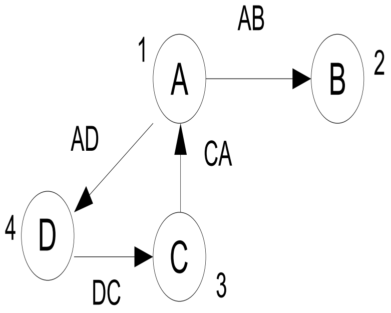


Figure 4.1 HDB-Subdue graph representation

Table 4.1 HDB-Subdue instances

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

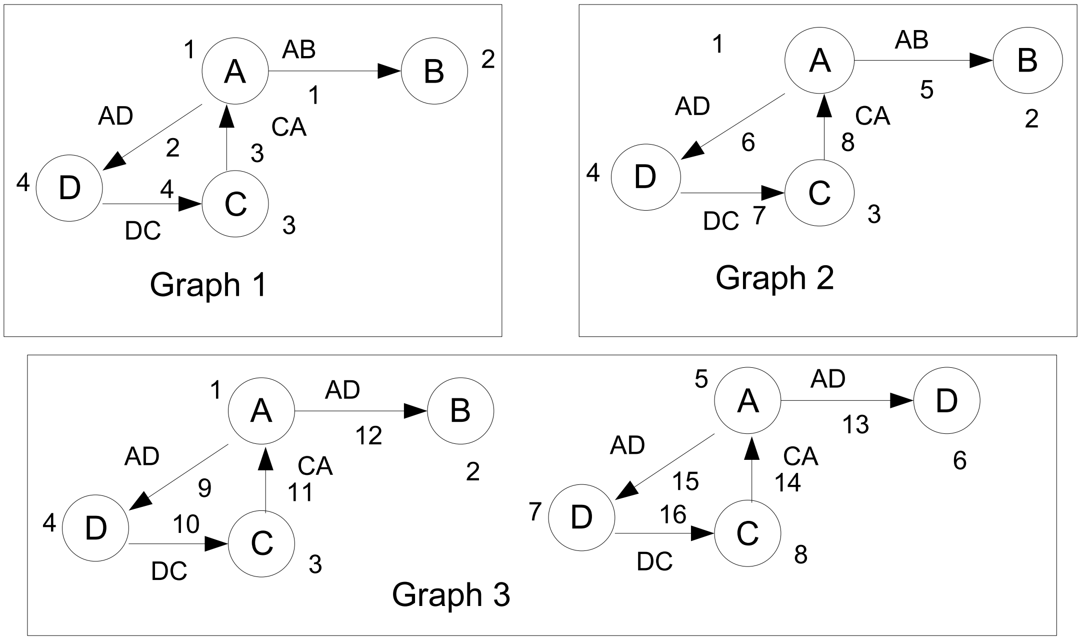


Figure 4.2 DB-FSG graph representation

Table 4.2 DB-FSG instances

V1	V2	V3	V4	VL1	VL2	VL3	VL4	EL1	EL2	EL3	GID	F1	T1	F2	T2	F3	T3
1	2	3	4	A	B	C	D	AB	AD	CA	1	1	2	3	1	1	4
1	3	4	0	A	C	D	-	AD	DC	CA	1	1	2	2	3	3	1
1	2	3	4	A	B	C	D	AB	AD	CA	2	1	2	3	1	1	4
1	3	4	0	A	C	D	-	AD	DC	CA	2	1	2	2	3	3	1
1	2	3	4	A	D	C	D	AB	AD	CA	3	1	2	3	1	1	4
1	3	4	0	A	C	D	-	AD	DC	CA	3	1	2	2	3	3	1
5	6	7	8	A	D	C	D	AB	AD	CA	3	1	2	3	1	1	4
5	6	7	0	A	C	D	-	AD	DC	CA	3	1	2	2	3	3	1

4.3 Unconstrained Expansion and Duplicate Elimination

In DB-FSG, the substructure instances are allowed to expand on any matching vertex (belonging to same graph) along the edges that does not exist in the substructure that is being expanded. This unconstrained expansion generates all possible substructures in an arbitrary graph input. However, this unconstrained expansion also allows same instance to be generated in different order (will be termed pseudo duplicates). For example in Fig 4.2, the substructure instances in the Table 4.3 represents the same substructures of graph 1. In order to identify same instances that grew in different order we can implement pseudo duplicate elimination approach PROPOSED in HDB-Subdue. In this approach, the pseudo duplicates are identified after canonically ordering vertex numbers and connectivity map of all the instances. The canonical ordering of vertex numbers and connectivity map is similar to canonical ordering of vertex labels that we will discuss in section 4.4. Canonical ordering is an expensive process which involves maintaining six intermediate tables, sorting of two intermediate tables, One 3-way join and one 2n+2-way join where n is the size of the substructure. In order to avoid the costly process of canonical ordering of vertex numbers and connectivity map, we present an alternative approach of constructing edge code that is unique to an instance. Construction of edge code is explained in subsection 4.3.1. We extend the existing representation of instances by introducing new attribute called ecode in instance_n table. This attribute will store edge code of each instance in the table. Then by comparing ecodes we can identify and remove pseudo duplicates much more efficiently.

Table 4.3 Pseudo Duplicates

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	F1	T1	F2	T2
1	3	4	A	C	D	2	3	AD	DC	1	1	3	3	2
1	4	3	A	D	C	2	3	AD	DC	1	1	2	2	3
3	4	1	C	D	A	2	3	CD	AD	1	2	1	3	1

4.3.1 Construction of Edge Code to Detect Pseudo Duplicates

Pseudo duplicates of an instance contains same edges in different order and each edge has a unique edge number. Hence, pseudo duplicates of an instance will have same edge numbers in different order. Thus, we can construct an unique edge code for each instance by concatenating Graph Id with edge numbers ordered in ascending order and separated by comma(,) . Construction of edge code is given in chapter 5 All pseudo duplicates of an instance in a graph will have same ecode.

Further more, if we have ecode for an instance of size n, constructing ecode for the instance of size n+1 (expanded from the instance of size n) requires only insertion of the newly added edge number into proper position. Hence, constructing ecode for instance of size n+1 expanded from instance size n has the time complexity of O(n).

4.3.2 Implementation of Edge Code to Detect Pseudo Duplicates

We created an Oracle function EdgeCode that takes size of the instance that is participating in expansion, edge code of the instance and the edge number of the new edge participating in the expansion. Then it returns the edge code for the expanded instances. We use the Oracle function EdgeCode in the expansion queries for instances of size 2 or more to maintain the edge code of the instances. The following queries extend a single edge substructure and also computed and stores the edge code value of the instances:

```
/* 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,EdgeCode(1,s.ecode,o.edge),
           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
                and o.GID=s.GID)


/* 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,EdgeCode(1,s.ecode,o.edge),
        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
                 and o.GID=s.GID)

/* 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,EdgeCode(1,s.ecode,o.edge),
          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
                and o.GID=s.GID)

/* 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, EdgeCode(1,s.ecode,o.edge),
           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 and
                 o.GID=s.GID)

/* 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,
          EdgeCode(1,s.ecode,o.edge),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 and o.GID=s.GID)

/* 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,
          EdgeCode(1,s.ecode,o.edge),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 and o.GID=s.GID)

/* 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,
           EdgeCode(1,s.ecode,o.edge),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 and o.GID=s.GID)

/* 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,
           EdgeCode(1,s.ecode,o.edge),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 and o.GID=s.GID)
```

4.3.3 Handling cycles using ecode

Consider the cycle of vertex 1,3 and 4 of graph 1in Fig 4.2. Even though the cycle might have many instances as shown in table 4.4, the edges in all the instances will be same. Thus, the edge code of the cycles will be same. Hence, using edge code can also detect different instances of a cycle. To avoid the redundant expansion on the repeating vertex of cycle the repeating vertex number is marked by '0' and the corresponding vertex label is marked by '-'.

Table 4.4 DB-FSG instances of cycles

V 1	V 2	V 3	V 4	VL 1	VL 2	VL 3	VL 4	EL 1	EL 2	EL 3	GID	ecode	F 1	T 1	F 2	T 2	F 3	T 3
1	3	4	1	A	C	D	1	AD	DC	CA	1	1,2,3,4	1	3	3	2	2	1
3	4	1	1	C	D	A	1	CA	AD	DC	1	1,2,3,4	1	2	2	3	3	1
4	1	3	1	D	A	C	1	DC	CA	AD	1	1,2,3,4	1	3	3	2	2	1

4.3.4 Pseudo duplicate elimination using ecode

After expansion of instances of size n-1, all the instances of size n are stored in instance_n table. Table 4.5 shows the instances of size two with additional attribute ecode. Then a group by on the ecode 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.

Table 4.5 Detecting Pseudo Duplicates using ecode

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	ecode	F1	T1	F2	T2
1	3	4	A	C	D	2	3	AD	DC	1	1,2,3	1	3	3	2
1	4	3	A	D	C	2	3	AD	DC	1	1,2,3	1	2	2	3
3	4	1	C	D	A	2	3	DC	AD	1	1,2,3	2	1	3	1

4.4 Canonical Ordering

The unconstrained expansion of substructure instances may also cause instances of similar substructure to grow in different ways. For example the table 4.6 represents the similar substructure instances for Fig 4.2. Hence, in order to get the correct frequency of similar substructures across the set of graphs, we need to identify similar substructures regardless of their growing order. This problem is addressed by canonical ordering of instances as in HDB-Subdue.

Table 4.6 Similar Substructure Instances

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

In order to identify two similar substructure instances , vertex labels and the connectivity attributes need to be used (unlike vertex numbers or edge numbers for pseudo duplicate elimination) If two instances have the same vertex labels and edge directions then they can be identified as similar (or isomorphic) instances. In SQL we can identify similar substructures only if the vertex labels and connectivity map OF each tuple is canonically ordered. Since databases do not allow rearrangement of columns (only rows by using group by and order clauses) 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. To facilitate construction of canonically ordered instance_n table we introduce an additional attribute called ID in unordered instance_n table. Each instance(tuple) in the instance_n table should have unique ID FOR which rownum is used as ID value. Table 4.7 shows the instance_2 table for Fig 4.2 before canonical ordering.

Table 4.7 Before Canonical Ordering

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

Owing to the table space constraints, canonical ordering of only the second and third instance are shown below. 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 4.8. 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 Id value from the instance table onto the unsorted table. Next we sort the table on *Id and vertex label* and insert it into a table called Sorted as shown in Tab 4.9 with its New attribute pointing to the new position of the vertex and the attribute Old pointing to the old position of the vertex.

Similarly the connectivity attributes are also transposed into a table called Old_Ext as shown in Table 4.10. 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 4.11. Next we sort the New_Ext table on Id and the attributes F (From vertex) and T (Terminating vertex).

Since, we also need ecode and GID attributes for expansion of the instances, the ecode and GID attribute were also transposed to tables called label_ecode_n and la-

Table 4.8 Unsorted

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

Table 4.9 Sorted

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

Table 4.10 Old_Ext

Id	EL	F	T
2	AD	1	2
2	DC	2	3
3	DC	2	1
3	AD	3	1

bel_GID_n. To differentiate between the GID and ecode of different instances we carry the Id value from the instance table onto the respective tables.

Now that we have the ordered vertex as well as connectivity map tables, we can do a *2n+3* way join (where n is the current substructure size) of *n+1* Sorted tables, label_GID_n table, label_ecode_n table and n Sorted_Ext tables to reconstruct the original instance in the canonical order. Table 4.13 shows the substructures after canonically ordering the vertex numbers and the connectivity attributes.

Table 4.11 New_Ext

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

Table 4.12 Sorted_Ext

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

Table 4.13 Instance table - After canonical ordering

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

4.5 Frequency Counting and Substructure Pruning

A graph may have many instance of the same substructure. For example, if we consider substructure in Fig 4.3. Then it has two instance in graph 3 of Fig 4.2. The table 4.14 shows the instances of substructure in Fig 4.3. If we count the frequency of the substructure from the instance table , it will give count of four even though the correct frequency count across the set of graph is three. Hence, in order to get the correct frequency of a substructure in the set of graph, we need to include only one instance per substructure within a graph. However, we need to preserve all the instances of a substructure that satisfies the support condition for future expansion. In order to get one instance per substructure of size n, we project *distinct* vertex labels, edge labels,connectivity map and GID and stores it into dist_n table. The dist_2 table for the set of graphs in Fig 4.2 is shown in table 4.15. Then a GROUP BY operation on vertex labels, edge labels and connectivity map in dist_n table will provide the correct frequency of each substructure. Since, the substructures having less frequency than support value will not contribute to future expansion, we can store only those substructures that satisfies the support value in sub_fold_n table. Then we can prune the instance_n table by removing instances of

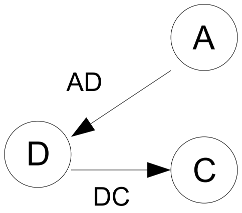


Figure 4.3 Substructure having more than one instance in a graph

substructures that are not in sub_fold_n table. Sub_fold_2 table of the Fig 4.2 is show in table 4.16.

Table 4.14 Instance table - with multiple instances of same substructure

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	GID	F1	T1	F2	T2
1	3	4	A	C	D	AD	DC	1	1	3	3	2
1	3	4	A	C	D	AD	DC	2	1	3	3	2
1	3	4	A	C	D	AD	DC	3	1	3	3	2
5	7	8	A	C	D	AD	DC	3	1	3	3	2
.	.	.	.	.	.	.	.	.	.	.	.	.

Table 4.15 Dist_n table

VL1	VL2	VL3	EL1	EL2	GID	F1	T1	F2	T2
A	C	D	AD	DC	1	1	3	3	2
A	C	D	AD	DC	2	1	3	3	2
A	C	D	AD	DC	3	1	3	3	2
.	.	.	.	.	.	.	.	.	.

Table 4.16 SubFold_2 table

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
A	C	D	AD	DC	1	3	3	2
A	C	D	AD	DC	1	3	3	2
A	C	D	AD	DC	1	3	3	2
.	.	.	.	.	.	.	.	.

4.6 Summary

In this chapter we introduced graph representation for a set of graphs. We addressed the issues raised due to unconstrained expansion by pseudo duplicate elimination and canonical ordering . We presented better alternative for pseudo duplicate elimination by using edge code. We also discussed an approach for frequency counting and substructure pruning.

CHAPTER 5

IMPLEMENTATION DETAILS of DB-FSG

This chapter discuses the pseudo duplicate removal using edge code, canonical ordering and substructure counting and pruning. The DB-FSG system has been developed using Pro*C [21] precompiler and C language, using Oracle database running on a Linux. 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 chapter. To eliminate these pseudo duplicates we can use canonical ordering on vertex numbers and connectivity map as done in HDB-Subdue [7]. However, canonical ordering is an expensive process. Hence, we explored a new approach for detecting pseudo duplicates using edge code in the design chapter. The approach implements an Oracle function EdgeCode to generate edge code for instances. The oracle function uses dynamic array of type Edge_array. Edge_Array is created as :

create or replace type EDGE_ARRAY as table of number;

The oracle function is given below with the explanatory comments.

```
1:  /* creating function to create unique edge code for each substructure */
    CREATE or REPLACE Function EdgeCode(sz IN Number,ecode IN Varchar2,
    newedge in NUMBER)
    /* takes size of substructure being expanded , edge code of the substructure and edge
    number of the edge being added */
```

```
RETURN VARCHAR2 /* returns new edge code as string */
AS /* variable declarations */
fcode Varchar2(600); /* size of the string in which new ecode will be stored.
  It can be extended up to 4000*/
fgid Varchar2(100);
temp NUMBER(38);
cnt number(10);
E_array EDGE_ARRAY := EDGE_ARRAY();
BEGIN
cnt := sz;
fgid:=SUBSTR(ecode,1,INSTR(ecode,',', 1, 1)-1);
FOR i IN 1 .. cnt-1
LOOP
  fcode:=SUBSTR(ecode,
    INSTR(ecode,',',1,i)+1,INSTR(ecode,',',1,i+1)-INSTR(ecode,',',1,i)-1);
  E_array.EXTEND;
  E_array(i):=TO_NUMBER(fcode);
END LOOP;
fcode:=SUBSTR(ecode, INSTR(ecode,',', 1, cnt)+1);
E_array.EXTEND;
E_array(cnt):=TO_NUMBER(fcode);
IF cnt=1 THEN
  temp:=E_array(1);
  IF temp¿newedge THEN
    fcode:=fgid ||',' ||newedge ||',' ||temp;
  ELSE
    fcode:=fgid ||',' ||temp ||',' ||newedge;
  END IF;
ELSE
  fcode:=fgid;
  temp:=cnt+1;
  FOR i IN 1 .. cnt
  LOOP
    IF E_array(i)¿newedge THEN
      temp:=i;
      EXIT;
```

```
36:      END IF;
37:   END LOOP;
38:   FOR i IN 1 .. temp-1
39:   LOOP
40:      fcode:=fcode||',' ||E_array(i);
41:   END LOOP;
42:   fcode:=fcode||',' ||newedge;
43:   FOR i IN temp .. cnt
44:   LOOP
45:      fcode:=fcode||',' ||E_array(i);
46:   END LOOP;
47:END IF;
48:RETURN fcode;
49:END EdgeCode;
```

Function Edge code creates edge code for each subgraphs. It takes size of the expanding sub-graph, edgecode of the sub-graph and the edge number of the edge on which the graph will expand and returns the new edge code for the expanded graph. Then it extracts the graph id from the edgecode and stores it in a string variable (done in the step 11 of the code). Then it extracts the edge numbers from the edge code and stores the edge number into an array in ascending order (done in steps 12 to 20). Then it compares the edge number from the new edge with each edge numbers in the array to find the appropriate position for the new edge number with respect to the edge numbers in the array (done in the step 20 to 37). Then it reconstructs the edge code by concatenating graph id and edge numbers less than the new edge number seperated by comma (done in steps 38 to 41). Then it appends the new edge number followed by the remaining edge numbers in the array to the newly constructed edge code (done in the step 42 to 47). The Edge code funcion returns a sring fromed by concatenation of graph id and edge numbers in ascending order seperated by comma.

In order to get the edge code of the expanded sub-graphs, the above Edge Code function is implemented in the expansion queries as shown below and as discussed in the design chapter . For the Fig 4.2 some of the pseudo duplicates are shown in table 5.1.

```
INSERT INTO instance_n (
   SELECT s.vertex1, s.vertex2,...,s.vertexn, o.vertex2,
    s.vertex1name, s.vertex2name,..., s.vertexnname,
    o.vertex2name, s.edge1,s.edge2,..., o.edge, s.edge1name,
    s.egde2name,..., o.edgename, EdgeCode(1,s.ecode,o.edge),
    s.from_1, s.to_1, s.from_2,s.to_2,..., 1, n
   FROM   InstanceIter_n s, oneedge o
   WHERE  o.vertex1=s.vertex1 and o.edge<>s.edge1 and o.edge<> s.edge2
    and...and o.vertex2<>s.vertex1
    and  o.vertex2<>s.vertex2 and o.GID=s.GID)


  INSERT INTO instance_n (
   SELECT s.vertex1, s.vertex2,...,s.vertexn, o.vertex2, s.vertex1name,
    s.vertex2name,..., s.vertexnname, o.vertex2name, s.edge1,
    s.edge2,..., o.edge, s.edge1name, s.egde2name,..., o.edgename,
    EdgeCode(1,s.ecode,o.edge),s.from_1, s.to_1,s.from_2,
    s.to_2,..., n, 1
   FROM   InstanceIter_n s, oneedge o
   WHERE  o.vertex1=s.vertex2 and o.edge<>s.edge1 and o.edge<> s.edge2
    and...and  o.vertex2<>s.vertex1 and
    o.vertex2<>s.vertex2 and o.GID=s.GID)
          .
          .
          .
```

Table 5.1 Detecting Pseudo Duplicates using ecode

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	ecode	F1	T1	F2	T2
1	3	4	A	C	D	2	3	AD	DC	1	1,2,3	1	3	3	2
1	4	3	A	D	C	2	3	AD	DC	1	1,2,3	1	2	2	3
3	4	1	C	D	A	2	3	DC	AD	1	1,2,3	2	1	3	1
1	2	4	A	B	D	1	2	AB	AD	1	1,1,2	1	2	1	3
1	4	2	A	C	D	2	1	AD	AB	1	1,1,2	1	2	1	3
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

As discussed in design chapter, all psuedo dupliates of same substructure will have same edge code.Hence performing a group by operation on edgecode and retaining an instance having maximum Id (or can be minimum Id however only one instance should be retained)will eliminate the psudoduplicates. 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 5.2

```
INSERT into Pseudo_Del
(SELECT MAX(Id)
 FROM Instance_n
 GROUP BY ecode)


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

Table 5.2 Table Instance_2 after pseudo duplicate elimination

V1	V2	V3	VL1	VL2	VL3	E1	E2	EL1	EL2	GID	ecode	F1	T1	F2	T2
1	3	4	A	C	D	2	3	AD	DC	1	1,2,3	1	3	3	2
1	2	3	A	B	C	1	2	AB	AD	1	1,1,2	1	2	1	4
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

5.2 Canonical Ordering

If we lexographically or canonically order instances of substructures, similar substructures instances will have same vertex labels and same connectivity attribute. Hence, in order to identify the similar substructure instances of size n, we need to canonically order the vertex labels and connectivity maps. Since, SQL does not allow column reordering we have to transpose the columns of the instance_n table into rows of intermediate tables, sort them, and reconstruct the canonically ordered instance_n table from the intermediate tables by performing join operations.

As an example let us consider the instance_2 table given in table 5.3 that represents instances of Fig 4.2

Table 5.3 Similar Substructure Instances

ID	V1	V2	V3	VL1	VL2	VL3	EL1	EL2	GID	ecode	F1	T1	F2	T2
2	1	4	3	A	D	C	AD	DC	2	2,2,4	1	2	2	3
3	3	4	1	C	D	A	DC	AD	3	3,2,4	2	1	3	1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Transposing of columns of the instance_n table is done by following queries.

```
INSERT into label_ecode_n(select id,ecode from instance_n)

INSERT into label_gid_n(select id, gid from instance_n)

INSERT into Unsorted_n(

      (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_n(

      (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 5.4, Tab 5.5, Tab 5.6, and Tab 5.8 and the Sorted table is shown in Tab 5.7. *2n+3* queries are required to project out the columns into rows where n is the substructure size.

Table 5.4 Label_GID_n

Id	GID
2	2
3	3

Table 5.5 Label_ecode_n

Id	ecode
2	2,2,4
3	3,2,4

Table 5.6 Unsorted

Id	V	VL	Pos
2	1	A	1
2	4	D	2
2	3	C	3
3	3	C	1
3	4	D	2
3	1	A	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 in the same

Table 5.7 Sorted

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

Table 5.8 Old_Ext

Id	EL	F	T
2	AD	1	2
2	DC	2	3
3	DC	2	1
3	AD	3	1

order as in the 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 THE 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 5.9.

```
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 5.10.

Table 5.9 New_Ext

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

Table 5.10 Sorted_Ext

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

To reconstruct the instance table back in canonical order, we do a *2n+3* way join of *n+1* Sorted tables, Label_gid_n, Label_ecode_n and n Sorted_Ext tables. The query to do that is shown below and the reconstructed table is shown in Tab 5.11.

```
INSERT into Instance_n(
SELECT S1.Id, S1.V .. Sn+1.V, S1.VL .. Sn+1.VL, O1.EL .. On.EL,
       LG.GID,LC.ecode,O1.F, O1.T .. On.F, On.T
FROM   Sorted S1 .. Sorted Sn+1, Sorted_Ext O1 .. Sorted_Ext On,
   Label_GID_n LG,Label_ecode_n,LC
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)
       and S1.id = LG.Id and S1.id=LC.Id
```

Table 5.11 Instance table - After canonical ordering

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

5.3 Substructure counting and pruning

As discussed in the design chapter, we have to make sure that only one instance of a substructure per graph is considered while counting the frequency of the substructure over the set of graph. Then we can prune the substructures that have less frequency count than user defined support value. Let us consider Tab 5.12 for frequency counting.

The following query inserts one instance of substructure per graph in dist_n table.

```
INSERT into dist_n(
SELECT DISTINCT VL1,VL2,...,EL1,EL2,...,GID,F1,T1,F2,T2,...
       from instance\_n)
```

Table 5.12 Instance table - with multiple instances of same substructure

V1	V2	V3	VL1	VL2	VL3	EL1	EL2	GID	F1	T1	F2	T2
1	3	4	A	C	D	AD	DC	1	1	3	3	2
1	3	4	A	C	D	AD	DC	2	1	3	3	2
1	3	4	A	C	D	AD	DC	3	1	3	3	2
5	7	8	A	C	D	AD	DC	3	1	3	3	2
.	.	.	.	.	.	.	.	.	.	.	.	.

Tab 5.13 shows the dist_2 table after executing the above query. The following query will insert the substructures satisfying user defined support condition sup_cond into sub_fold_n table. Tab 5.14 shows the substructure satisfying the support condition.

Table 5.13 Dist_n table

VL1	VL2	VL3	EL1	EL2	GID	F1	T1	F2	T2
A	C	D	AD	DC	1	1	3	3	2
A	C	D	AD	DC	2	1	3	3	2
A	C	D	AD	DC	3	1	3	3	2
.	.	.	.	.	.	.	.	.	.

```
INSERT into sub_fold_n(

SELECT VL1,VL2,...,EL1,EL2,...,F1,T1,F2,T2,...from dist_n

group by VL1,Vl2,...,EL1,EL2,F1,T1,F2,T2,... having count(*)>=sup_cond)
```

Table 5.14 SubFold_2 table

VL1	VL2	VL3	EL1	EL2	F1	T1	F2	T2
A	C	D	AD	DC	1	3	3	2
A	C	D	AD	DC	1	3	3	2
A	C	D	AD	DC	1	3	3	2
.	.	.	.	.	.	.	.	.

At the end of this step, the substructure satisfying support conditions are stored in sub_fold_n table. We can now store instances of the substructures into instanceiter_n table for further expansion. The following queries stores the instances of substructures satisfying support condition into the instanceiter_n table.

```
INSERT into instanceiter_n(

SELECT i.V1,i.V2,i.VL1,i.VL2,...,i.E1,i.E2,...,i.EL1,i.EL2,...,

i.F1,i.T1,i.F2,i.T2,...from instance_n i, sub_fold_n s

where i.VL1=s.VL1 and i.VL2=s.VL2,...,i.EL1=i.El1,i.El2=i.El2,...)


INSERT into instance_n(select * from instance_n)
```

5.4 Summary

This chapter addressed the implementation issues involving pseudo duplicate elimination, canonical ordering, substructure counting and pruning. SQL queries and Oracle function implemented to address these issues were also presented in this chapter. It also illustrated how the corresponding tables were updated in the database.

CHAPTER 6

PERFORMANCE EVALUATION

This chapter gives an overview of the performance of DB-FSG. We will compare canonical ordering approach of HDB-Subdue with our approach of using edge code for pseudo duplicate elimination. Then we will provide the performance of DB-FSG over a large set of graphs. We will also evaluate the processing time and intermediate space required by Naive, Db2Graph and SubdueDb2Graph algorithms of DB2GraphGen system.

6.1 Graph Generator

The graph generator used for generating the input graphs for HDB-Subdue and extended HDB-Subdue was developed by the AI Lab [22] at the University of Texas at Arlington. The graph generator accepts many parameters and constructs a graph as set of vertices and edges. The parameters accepted by graph generator 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 size graph that it can generate will depend upon the available system memory.

6.1.1 Generating a set of graphs

We modified the graph generator to generate a set of graphs for DB-FSG. The graph representation was extended to mine over the set of graphs. In the extended representation, each graph contains a unique graph id, and vertices and edges belonging to same graph possess same graph id. The parameter instance number for each substructure was replaced by the support for the substructure. User can give value from 0 to 100 percent as a support for the substructure. Graph generator will calculate the number of instances as *(support X number of graphs)/100.* Then it will embed the calculated number of instances of the substructure across the set of graphs. These modifications on the graph generator was sufficient to generate a set of graphs for DB-FSG.

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. In this section we will discuss about the input parameters for DB-FSG,extended HDB-Subdue and DB2GraphGen.

6.2.1 Input parameters for DB-FSG

The input parameters with their explanations for DB-FSG are given below.

```
User Name # Password # Input Table Name # MaxSize # Support

# Optimize # Debug # LogResults # Useecode
```

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.

*MaxSize:*Specifies maximum size of the substructure to be discovered.

Support: Minimum frequency(count) for the substructures. Substructures with frequency value less than support are pruned.

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

Useecode: Specifies whether to use canonical ordering or edge code for pseudo duplicate elimination. 1- use edge code 0-use canonical ordering.

For each experiment, the values of all these parameters are specified in a single line in the order shown above. We can also use executable script file to automate execution of DB-FSG for many inputs and different parameters. An example entry to execute extended DB-FSG is given below.

```
 -ipfile mg_20K -usrname scott -password tiger -subsize 3 -support 50
 -optimize 1 -debug 0 -log 1 -ecode 1
```

6.2.2 Input parameters for extended HDB-Subdue

The parameters and their explanations for extended HDB-Subdue are given below.

```
User Name # Password # Input Table Name # MaxSize # Beam # Iterations
```

`# EvalApproach # BestSubs # Optimize # Debug # LogResults # Ties # Useecode`

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

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

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

Useecode: Specifies whether to use canonical ordering or edge code for pseudo duplicate elimination. 1- use edge code 0-use canonical ordering.

For each experiment, the values of all these parameters are specified in a single line in the order shown above. We can also use executable script file to automate execution of extended HDB-Subdue for many inputs and different parameters. An example entry to execute extended HDB-Subdue is given below.

```
-ipfile T20V25E  -user scott -password tiger -subsize 3 -iterations 1 -beam 10
-approach 2 -bestsubs 5 -optimize 1 -debug 1 -log 1 -ties 0 -useecode 1
```

6.3 Writing Log File

Graph mining and transformation of relational database to graph is a time-consuming process and for some data sets it may take hours to complete. Further more, all the experiments should be done on same machine to analyze the performance of different approaches and algorithms. Hence, we implemented the log file to store performance of extended HDB-Subdue and DB-FSG. To compare the performance of edge code approach with canonical ordering approach of HDB-Subdue we need to run experiments on the same machine. The running time of each query is captured in an array and at the completion of a dataset it is written out to a log file, if the log results option is n. 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 for HDB-Subdue and Extended HDB-Subdue

In order to compare canonical ordering of HDB-Subdue and edge code approach of extended HDB-Subdue, graphs with and without multiple edges, with and without cycles were used. Experiments were conducted on several data sets of different sizes from 20 vertices 25 edges to 50K vertices 100K edges to verify the efficiency of edge code approach over canonical ordering. All of the 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 of the experiments were performed on a Linux machine using Oracle10G. The machine was running on dual processors with 2 GBytes of memory. For the verification of correctness of edge code approach, both approaches were experimented on the graph of size 20 vertices and 25 edges with the substructure shown in Fig 6.1. Then we experimented on larger graphs with same parameters. The parameters set for both approach are shown in Tab 6.1. Edge Code approach (extended HDB-Subdue) was able to detect same substructures and same number of instances as canonical ordering(HDB-Subdue). Total processing time of both approaches are shown in Tab 6.2. The experiment results showed that the edge code approach was much more efficient than the canonical ordering of HDB-Subdue.

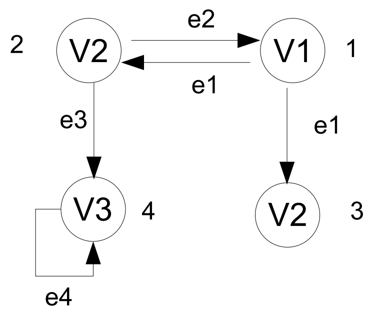


Figure 6.1 Example Graph

Table 6.1 Parameter Settings

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

Table 6.2 Performance of HDB-Subdue and Extended HDB-Subdue

Graph size	HDB-Subdue(in sec)	Extended HDB-Subdue(in sec)
T500V1000E	27.87	12.76
T1KV2KE	27.93	15.62
T5KV10KE	184.1	57.4
T15KV30KE	1251.03	323.35
T50KV100KE	14676.83	2681.59

6.5 Experiment Results for DB-FSG

The purpose of the experiments conducted for DB-FSG was to analyze the performance of the algorithm for different sized dataset and for various support value. The experiment was conducted in a Linux machine of Distributed and Parallel Computing Cluster at UTA (DPCC@UTA) using Oracle10G. The machine was running on dual processors with 2.4 G Hz cpu speed and 2 GBytes of memory. All of the experiments were conducted 4 times and the first run (cold start)was discarded and the timings from other three runs were averaged to obtain average running times.

The first set of experiments were conducted to analyze the performance of DB-FSG in varying datasets. We performed experiments on the datasets containing graphs without cycles and multiple edges, graphs with cycles and graphs with multiple edges. All graphs in the dataset have 40 edges and 40 vertices. The datasets varied from 50K of graphs (2 million vertices and 2 million edges in the dataset) to 300K of graphs (12 million vertices and 12 million edges in the dataset). The frequent substructures embedded in the dataset had support value of 3% and 4%. Tab 6.3 shows the parameters for the set of experiments.

Table 6.3 Parameters for DB-FSG

Substructure size	Support	Use-ecode
5	1	1

Tab 6.4 and Fig 6.2 gives the processing time required by DB-FSG on dataset containing graphs without cycles and multiple edges. Tab 6.5 and Fig 6.3 shows the performance of the DB-FSG on the dataset with cycles. Tab 6.6 and Fig 6.4 shows the performance of DB-FSG on dataset with multiple edges. The experiments results showed that the processing time of algorithm increases linearly as the size of the dataset grows. The number of substructures instances discovered in dataset containing graph with cycles was less than the graphs without cycles and graphs with multiple edges. Hence, the procesing time of the datasets containing cycles was less than the graphs without cycles and graphs with multiedges.

Table 6.4 Performance of DB-FSG on simple graphs

# Graphs	Total # Vertices	Total # Edges	Processing Time in sec
50K	2 Million	2 Million	391.23
100K	4 Million	4 Million	1510.40
150K	6 Million	6 Million	2572.61
200K	8 Milion	8 Million	3680.08
250K	10 Million	10 Million	4663.78
200K	12 Million	12 Million	5692.28

Then we conducted experiments to analyze the performance of the algorithm for varying support value. We performed experiments on the datasets containing graphs

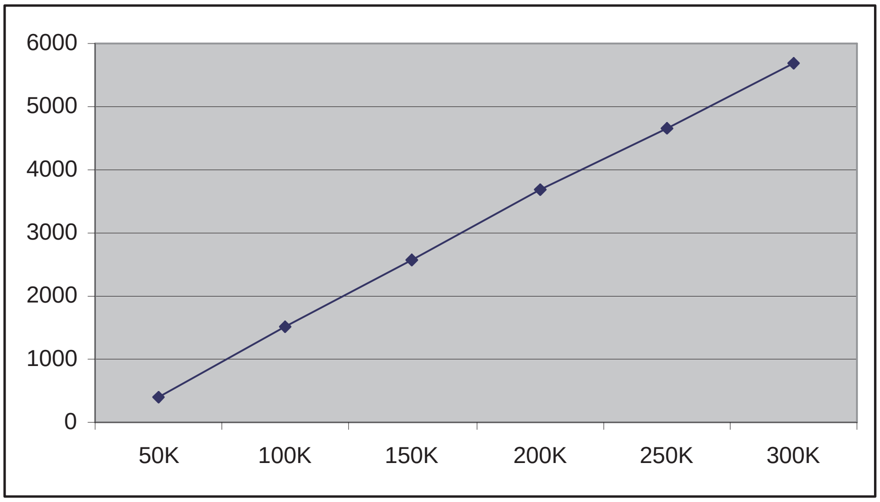


Figure 6.2 Performance of DB-FSG on simple graphs

Table 6.5 Performance of DB-FSG on graphs with cycles

# Graphs	Total # Vertices	Total # Edges	Processing Time in sec
50K	2 Million	2 Million	431.74
100K	4 Milion	4 Million	1516.365
150K	6 Million	6 Million	2313.04
200K	8 Milion	8 Million	3233.4
250K	10 Million	10 Million	4387.89
200K	12 Million	12 Million	5297.8

without cycles and multiple edges. The graphs in the dataset have 30 to 50 edges and 30 to 50 vertices. The datasets varied from 100K of graphs (approximately 2 million vertices and 2 million edges in the dataset) to 300K of graphs (approximately 12 million vertices and 12 million edges in the dataset). The frequent substructures embedded in the dataset had support value of 1%, 3%, 5% and 7%. We varied the support value from 1% to 7 % while keeping the max size as 5 in order to evaluate the performance of the DB-FSG on those datasets.

Table 6.7, Table 6.8, Table 6.9 and Figure 6.5 shows the relation of support value with the processing time. The experiment results showed that the processing time de-

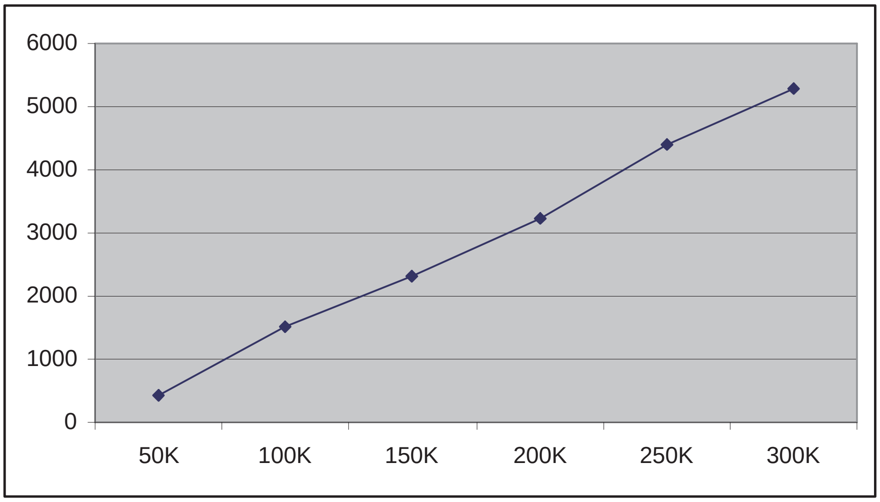


Figure 6.3 Performance of DB-FSG on graphs with cycles

Table 6.6 Performance of DB-FSG on graphs with multiple edges

# Graphs	Total # Vertices	Total # Edges	Processing Time in sec
50K	2 Million	2 Million	560.80
100K	4 Milion	4 Million	1735.80
150K	6 Million	6 Million	2639.49
200K	8 Milion	8 Million	3535.39
250K	10 Million	10 Million	4590.78
200K	12 Million	12 Million	5604.93

creased as the suppport value increased. For greater support value more substructures will be pruned in earlier iterations of the algorithm hence, less processing time is required. The number of substructure instances retained for 7% of support value will be less than the number of instances retained for support value of 1% in each expansion iteration. Hence, the processing time for each steps DB-FSG like substructure expansion, psuedo duplicate elimination, canonical ordering and substructure counting and pruning will require less time for user defined support value of 7% than for 1%.

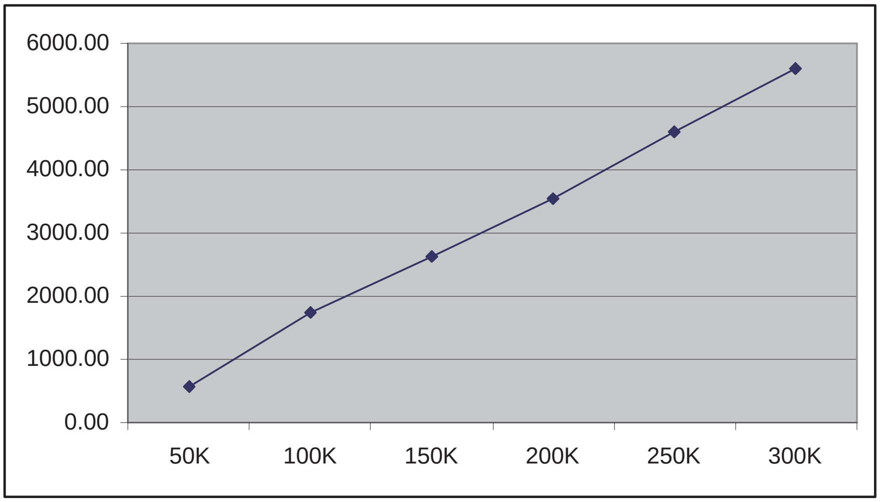


Figure 6.4 Performance of DB-FSG on graphs with multiple edges

Table 6.7 Performance of DB-FSG for 100K Graphs

Support	Total time in sec
1%	1892.89
3%	1679.05
5%	1516.74
7%	1064.64

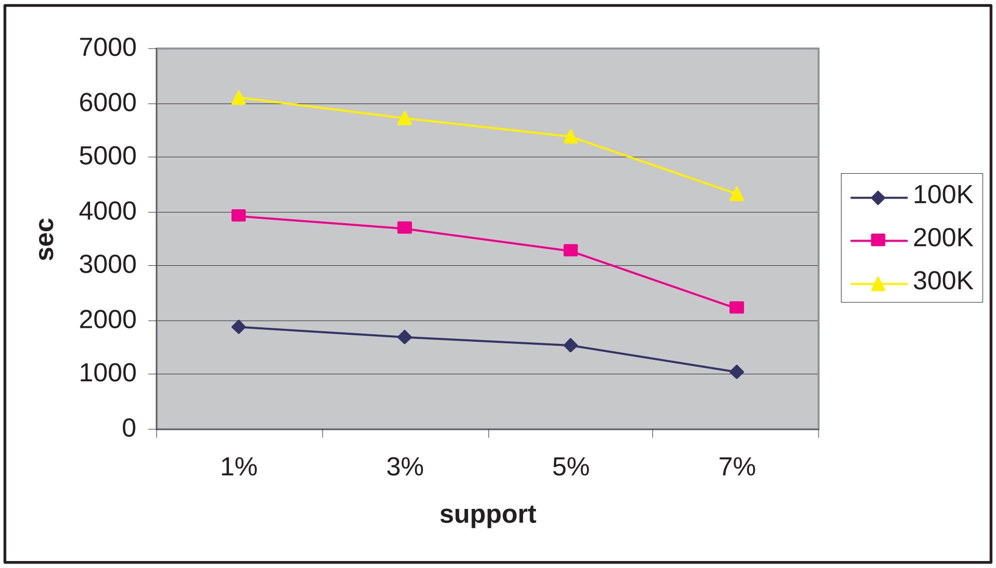


Figure 6.5 Performance of DB-FSG for Different Support Value

Table 6.8 Performance of DB-FSG for 200K Graphs

Support	Total time in sec
1%	3912.11
3%	3697.68
5%	3280.12
7%	2224.03

Table 6.9 Performance of DB-FSG for 300K Graphs

Support	Total time in sec
1%	6088.9
3%	5722.07
5%	5374.15
7%	4323.82

CHAPTER 7
DB2GraphGen

7.1 Overview

In this chapter we will describe DB2GraphGen system that transforms data from relational database to graph-based domain. The focus of the system is to attain maximum portability across different relational database management systems(RDBMS). Even though many RDBMS follows SQL standards there are variations in metadata schema and data definition syntax. So, approaches developed for one RDBMS may not work on others. RDB2Graph [19] is an example of such approach. It is developed for RDBMS Oracle hence, considerable modifications are needed in order to apply this approach to any other relational database management system. Therefore, portability of DB2GraphGen across various RDBMS is among our primary goals. In order to achieve portability, we confined the data definition and manipulation queries to basic SQL standard. We also modularized the system into database platform dependent modules and platform independent modules. Modularization of the system will be discussed in section 7.2.

The resultant graph of the transformation is used by graph mining algorithms to detect interesting patterns. Therefore graph representation of data from relational database is important as the size of the input for mining may increase significantly (e.g., can double) based on the representation chosen. This will have a critical impact on the processing time and main memory requirements. Also, choice of a representation that captures the semantics of the transactions is important. Otherwise, it may be difficult to interpret the results of mining. We will present a comparative study of various graph representations in section 7.3.

Since databases may contain huge amount of data, the transformation process might utilize huge amount of memory or space to store intermediate data during transformation. This might effect the scalability of the system. Hence, another focus for the system was to minimize the maximum memory/space utilization. In order to minimize the space/memory utilization during transformation we present various algorithms in section 7.4.

7.2 Architecture of DB2GraphGen

Different relational database platforms (Oracle, DB2, SQLServer, MySQL) use different metadata representations. So, it is not possible to use a generalized query to extract the schema information such as the list of tables, list of attributes, foreign key constraints and primary key constraints. On the other hand most of the relational databases follow SQL standards for data manipulation and data definition . Thus, we can exploit the portability of generalized queries for data definition and manipulation. Hence, we modularize DB2GraphGen to minimize system development in order to incorporate new relational database management systems (DBMS) as shown in the Fig 7.1

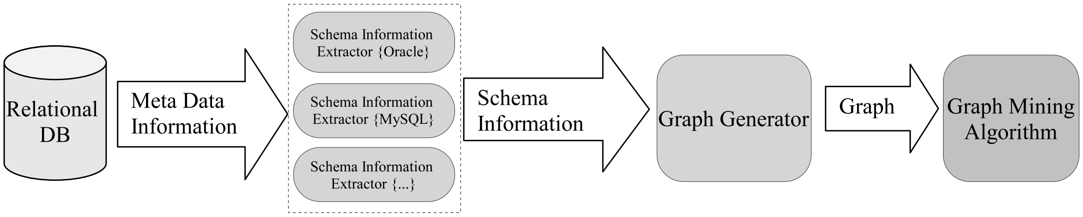

Relational DB
Meta Data Information
Schema Information Extractor {Oracle}
Schema Information Extractor {MySQL}
Schema Information Extractor {...}
Schema Information
Graph Generator
Graph
Graph Mining Algorithm

Figure 7.1 Architecture of DB2GraphGen

7.2.1 Schema Information Extractor:

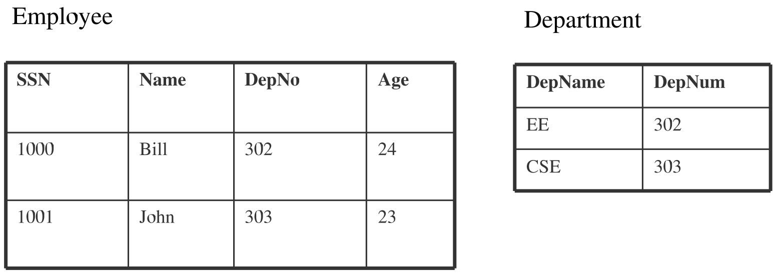
This module extracts the entity relationships such as the list of relations (tables), list of attributes, list of primary and foreign key attributes associated with each relation along with the size of those attributes and number of tuples in each relation. This information is stored into data structures, which act as inputs to the *Graph Generator* module. As this information is dependent on the metadata of the database, we need to develop platform dependent Schema Information Extractors. As shown in the Fig 7.1, we have currently developed a Schema Information Extractor for Oracle and MySQL.

7.2.2 Graph Generator:

Graph Generator gets the necessary schema input from the Schema Information Extractor and transforms these database relations into a graph. Since all the information will be provided to this component in form of data structures, this component will be platform-independent. The database to graph conversion for Graph Generator is done using SQL92 standard queries. We have developed three algorithms for graph conversion, which will be explained in following sections.

7.3 Graph Representation of Relational Database

Relations in a relational database can be represented as a graph in a number of ways. Any graph representation used for Relational Database representation should preserve all the relational information represented by the database. In this section we will analyze different graph representations for relational databases using two small relations: Employee (SSN is primary key and DepNo references Department) and Department (DepNum is primary key) as shown in Fig 7.2.



Employee

SSN	Name	DepNo	Age
1000	Bill	302	24
1001	John	303	23

Department

DepName	DepNum
EE	302
CSE	303

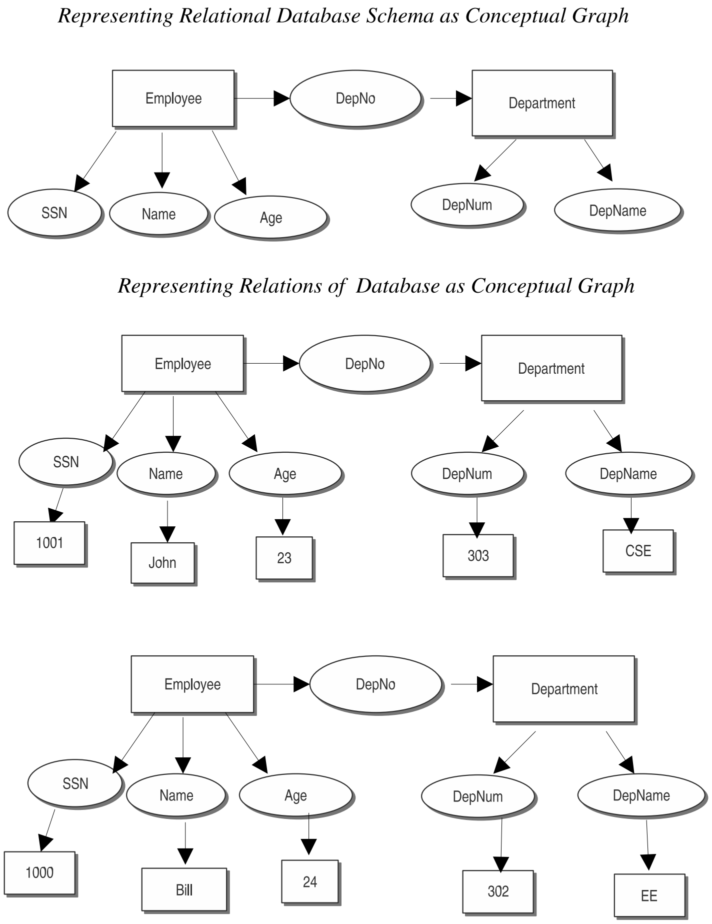
Figure 7.2 Tables from Relational Database

7.3.1 Conceptual Graph

As shown in Fig 7.3, in Conceptual Graph representation, relations are represented as rectangular node, attributes as oval nodes and attribute values as rectangular nodes. Representing relations as a conceptual graph [9, 10] retains all the relational information in the database but this representation has some ambiguity. Even though the foreign key attribute node shows relationship between two tables, it does not reveal the attribute on which the tables are related. Also, since the edges do not have any weights or labels associated with them, the significance of these edges is rendered inconsequential with regards to graph mining algorithms.

7.3.2 EntitytoAttributeGraph

EntitytoAttributeGraph is a modified Conceptual Graph. In this representation, attribute names are represented as edge labels instead of nodes and the common attribute between relations is shown by a directed edge from each relation to the common attribute. In this representation, the relational database schema and dataset is represented as shown in Fig 7.4.



Representing Relational Database Schema as Conceptual Graph

Representing Relations of Database as Conceptual Graph

Figure 7.3 Representing Relational Database as Conceptual Graph

7.3.2.1 Relation of Relations (Tables) Graph (RR-graph)

RR-graph provides relation reference information of the database schema. RR-Graph is a directed graph where nodes represent relations (tables) and a directed edge from node R1 to node R2 implies that relation R1 is referenced by relation R2. Fig 7.5 shows how the relationship between relations can be represented by an RR-Graph. RR-graphs and EntitytoAttribute representations provide a clear picture of primary key-foreign key relationship between relations than a Conceptual Graph alone.

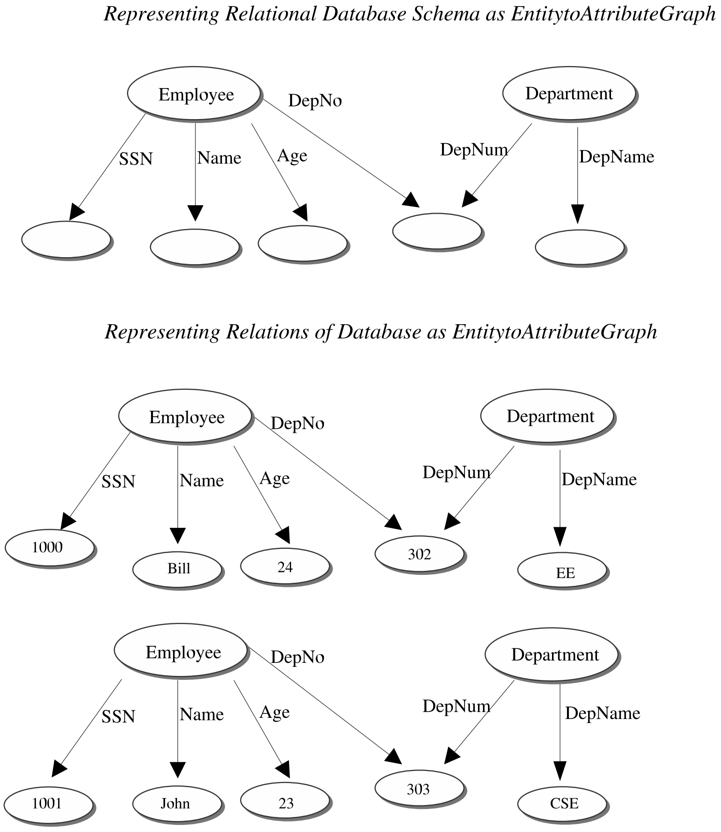
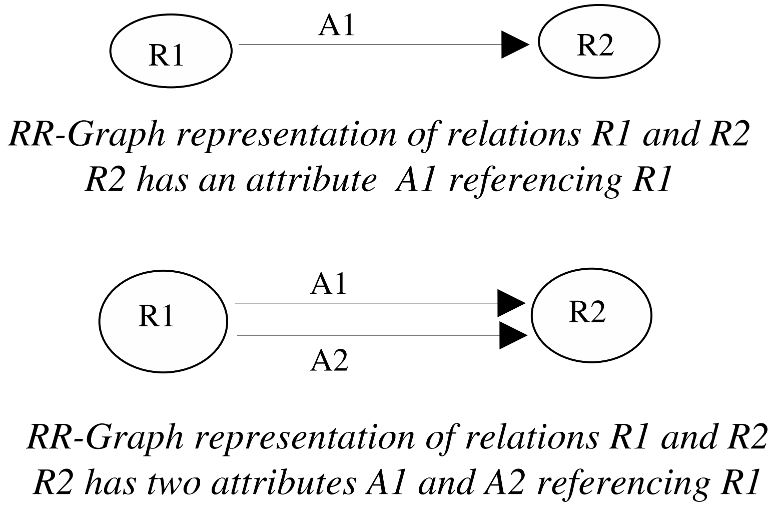


Figure 7.4 Representing Relational Database as EntitytoAttribute Graph

7.3.3 Conceptual Graph VS EntitytoAttributeGraph

The number of vertices and edges in a graph determines the space required for a graph. Since the graph representation of a relational database will be used as an input to graph mining algorithms, the processing time and memory requirement for graph mining is dependent on the number of edges and vertices required (space required) for the graph representation of the relational database. So, we compare the number of edges and vertices required by Conceptual Graph and EntitytoAttributeGraph for representing a relation. Let n be the number of tuples and m be the number of attributes in a relation. In the Conceptual Graph, for each tuple, we need one node for table name, m nodes for



RR-Graph representation of relations R1 and R2
R2 has an attribute A1 referencing R1

RR-Graph representation of relations R1 and R2
R2 has two attributes A1 and A2 referencing R1

Figure 7.5 RR-Graph

attribute names, m nodes for the attribute values and $2m$ edges to show the relation between each node.

$$\textit{Number of nodes required for representing the table into graph} = 2\ X\ m\ X\ n + n$$

$$\textit{Number of edges required for representing the table into graph} = 2\ X\ m\ X\ n$$

In EntitytoAttributeGraph, we need one node for table name for each tuple of the table , m nodes for attribute values and m edges representing attribute names.

$$\textit{Number of nodes required for representing the table into graph} = m\ X\ n + n$$

$$\textit{Number of edges required for representing the table into graph} = m\ X\ n$$

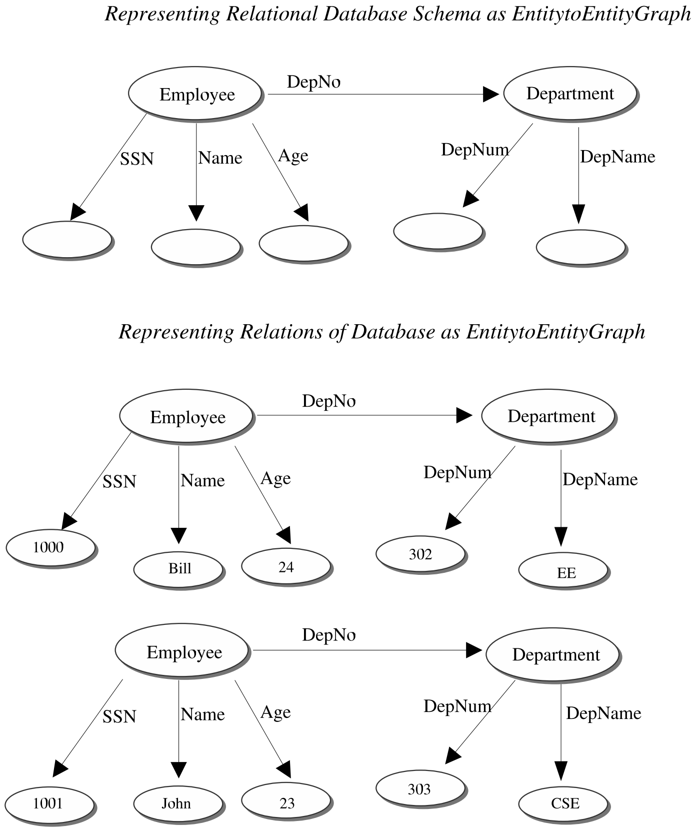
In comparison to Conceptual Graph, EntitytoAttributeGraph requires approximately half the number of vertices and edges to represent a relation (50% reduction). Hence, graph mining algorithms will process EntitytoAttributeGraph much faster than the Conceptual Graph.

7.3.4 Graph Representation for Subdue

As mentioned earlier, a relational database can be represented as a graph in various ways. However, an important criteria for the graph representation is the ability of the graph mining algorithm like Subdue [1] to process the representation. Any representation failing to fulfill this criteria is not considered for graph representation. For example a relational database can be represented using the approach used by BANKS [20]. In BANKS [20], each tuple from all the tables is represented as an individual node. Whenever two tuples are connected, the nodes will have table names as labels for the edges. Since Subdue [1] cannot process such a representation, the approach used by BANKS was not considered. Also, considering an entire tuple as a node does not allow the detection of interesting patterns with respect to individual attributes of a relation. Even though Subdue can process EntitytoAttribute graphs, we modify the representation to maximize the effectiveness of Subdue. The new approach, denoted as *EntitytoEntityGraph* (Fig 7.6), uses the foreign key of a relation as a directed edge from the referencing relation to the referenced relation. DB2GraphGen is capable of transforming the relational databases to graph based domains using either EntityToAttribute or EntitytoEntity graph representations.

7.4 Algorithms for Transforming Relations to Graph

In this section, we discuss the Naive algorithm, Db2Graph algorithm and SubdueDb2Graph algorithm implemented for the Graph Generator module to transform the relations into a graph. The space requirements for these algorithms are dictated by the space required to maintain the intermediate data (lookup). Lookup contains information required to connect sub-graphs representing tuples of different relations. The structure of a lookup is dependent on the algorithm implemented. For example, Naive algorithm maintains primary key values, primary key node numbers and foreign key values as lookup


Representing Relational Database Schema as EntitytoEntityGraph

Representing Relations of Database as EntitytoEntityGraph

Figure 7.6 Representing Relational Database as EntitytoEntity Graph

for a relation whereas Db2Graph algorithm maintains primary key node numbers and primary key values. We analyze these algorithms on the basis of space requirement for maintaining lookup or intermediate data. These intermediate data or lookup can be kept in main memory, flat files or in relations of the database. Since we are focusing on large databases, we have used database relations to store the intermediate data(lookup) for the benefit of scalability and simplicity.

Naive algorithm, similar to the RDB2Graph [19] algorithm, processes relations in random order. The DB2Graph algorithm determines the sequence of relations to be processed such that the lookup space requirement is minimized to generate the EntitytoAttributeGraph representation. SubdueDb2Graph, a modification of the DB2Graph algorithm, provides the database to graph transformation in the form of EntitytoEntityGraph representation.

7.4.1 Naive Algorithm

Algorithm 2 Naive Algorithm

```
1: Take a relation(table) Rc from the database.
2: for all tuple (Tc) in the relation do
3:     Create a node for relation name (will be known as table name node)
4:     Create nodes for all the attribute values in the tuple except for foreign key attributes
5:     Connect it with table name node (directed edge from table name node to attribute
       value node) with edge label as attribute name.
6: end for
7: Repeat step 1 for all the relations in the database.
8: Take a relation Rc from the database.
9: for all tuple (Tc) in the relation Rc do
10:    Let the relation referenced by Rc be Rp
11:    Connect the table name node of Tc with primary key value node of the tuple (Tp)
       of the Rp that has same value as foreign key value of Tc. (directed graph from
       table name node of Tc to primary key value node of Tp).
12: end for
13: Repeat step 4 for all relations in database
```

As shown in Algorithm 2, Naive Algorithm transforms a relational database to a graph in two phases. The first phase transforms all relations into graphs (steps 1-3). In this, each row of every relation of the database is represented as a connected subgraph. Instances of subgraphs during this step can be directly sent to output. However, in order to perform this phase we need schema information like relation name, primary key, foreign key and other attributes of every relation. The second phase is to connect the subgraphs on the basis of primary key and foreign key attributes of the relations (done in step 4). For this, we have to store primary key vertices (nodes) and foreign key values of every tuple of each relation in the database as lookups. In the Naive algorithm, since relations are processed randomly, it does not keep track of the relations that have been processed and relations that are yet to be processed. Due to the absence of this information, there is no way to determine the utility of lookups of processed relations during the transformation. Hence, Naive algorithm requires lookups of every relation till the transformation is complete.

7.4.2 DB2Graph Algorithm

7.4.2.1 Minimizing Space Requirement for Lookup:

As seen in the previous subsection, Naive algorithm requires large number of lookups since it maintains the attribute values of all relations regardless of their utility for the transformation of the relational database to the final graph. If we determine the sequence of relations to be transformed and maintain the record of the connectivity for these relations (on the basis of primary and foreign keys), then we can easily identify those relations for whom lookups are not required. We merge the two phases of the Naive Algorithm such that while transforming a relation into a graph, we also connect the subgraphs to the ones that has been processed earlier. With this approach, we maintain the information of the relations that have been processed earlier to detect and delete those lookups that will not be required in the future. Hence, the space requirement is considerably reduced as compared to the Naive approach.

Consider the RR-Graph in Fig 7.7. In the figure, number of tuples (rows) for each vertex (relations) is provided above the vertex where 1K is equivalent to 1000 tuples. For simplicity we assume that the primary key of each relation is made up of only one attribute (i.e., primary key is not a composite key), the size of the all primary and foreign keys are same and the space required to store the corresponding vertex number of the primary or foreign key value is negligible. In the figure, the edge labels (referencing

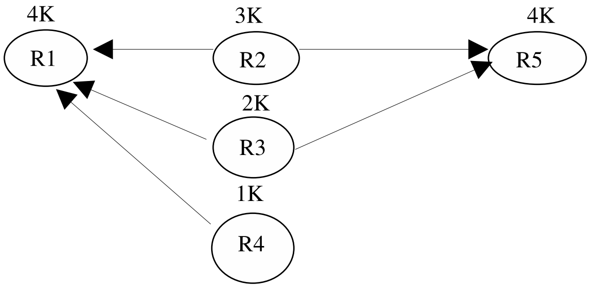


Figure 7.7 RR-Graph

attributes) are omitted since we are only calculating the space requirement. For this graph, the lookup size using the Naive approach will be

primary key values size = (4K+4K+3K+2K+1K) X size of primary key

= 14K X size of primary key

foreign key value size = (4K X 3+4K X 2) X size of foreign key

= 20K X size of foreign key

Total lookup space = 34K X size of primary key

However, if we transform the relations into the sequences R1, R2, R3, R4 and R5, then we do not have to maintain the lookups for relation R1 and R5. Also, after processing R1 we can delete the lookups of R4. The lookup space requirement for this sequence will be:

After processing R2,R3,R4

lookup requirement = (1K+2K+3K) X size of primary key

= 6K X size of primary key

After processing R1

lookup requirement = (2K+3K) X size of primary key = 5K X size of primary key

Therefore, the maximum space requirement = 6K X size of primary key.

7.4.2.2 Determining the Sequence of Relations for Transformation:

As shown in the above example, transforming the relational database on a particular sequence of relations require less lookup space than processing the relations in random order. Also, some sequences require less lookup space than others. Hence, determining the sequence of relations that will result in minimum lookup space requirement is another problem to be addressed. One approach to select the order of relations is to use the tuple size.

Lemma: Relation having minimum tuple size should be selected first

However, the tuple size alone does not provide an accurate estimation of lookup space required for a relation. Along with tuple size, the number of primary and foreign key attributes also affect the lookup space of a relation. For the case of RR-Graph, the number of edges incident (in-degree) and going out from the node (out-degree) also affect the space requirement for lookup. Also, once the primary key values of a relation in a lookup are stored, they can be referenced by any number of relations. So, the space required to maintain the lookup, denoted as the weight of a node, is calculated as:

Weight of a node=(minimum (outdegree,number of primary key)+indegree) X tuplesize

Since, tuple size alone does not provides the correct estimation of looukup space, sequence of relations determined by considering minimum tuple size may not always lead to the minimum lookup space requirement. Further more, weight assigned to relations in RR-Graph provides more accurate estimation of lookup space for a relation than tuple size alone. Hence, using minimum weight strategy to determine the sequence of relations will result to less lookup space than tuple size alone.

Lemma: The nodes having minimum weight should be selected first.

7.4.2.3 Dynamic Update of Weight:

After a relation is selected and its lookup is stored, the weight of nodes connected with the selected node will be affected. Since, primary and foreign key values of the selected relation are already stored in the lookup, we do not have to store the key values for the relations referencing the selected relation. As a result of this, updating the weight of the relations after each relation transformation along with minimum weight strategy will provide a more accurate lookup estimation. To strengthen the above argument we present a small example where minimum weight strategy results into less lookup space than minimum tuple size strategy.

7.4.2.4 Example of Tuple Size vs Weight

Consider the RR-Graph in figure 7.8. If we select the relations considering just the tuple size (taking relations having less number of tuples), the order of relations will be R2, R3 and R1. The memory requirement for maintaining lookup per iteration is shown in table 7.1.

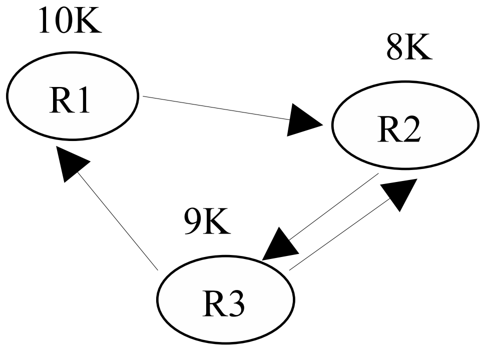


Figure 7.8 RR-Graph

So, maximum memory required is 33K for this approach.

The table 7.2, table 7.3 and table 7.4 shows the memory requirement per iteration when

Table 7.1 Memory Requirement

Relation transformed	lookup size	Remarks
R2	24K	We need to keep lookup of primary key and foreign keys of R2
R3	24K + 9K = 33K	We need to keep lookup of primary key of R1
R1	33K	Lookup required for R1 is already stored in memory

we consider the weight (taking relations having less weight first) and use ”’Dynamic Update of Weight after Selection of a Relation”’.

Weight of R1=(1+1) X 10K=20K

Weight of R2=(1+2) X 8K=24K

Weight of R3=(1+1) X 9K=18K

Table 7.2 Memory Requirement

Relation transformed	loookup size	Remarks
R3	18k	Look up is required for primary key and foreign key of R3

Since, Relation R3 already has lookup for R2 and R1 we dont need to consider the edges from R3.

Weight of R2=(0+1) X 8K=8K

Weight of R1=(1+0) X 10K=10K

Weight of R1=(0+0) X 9K=0

Table 7.3 Memory Requirement

Relation transformed	loookup size	Remarks
R2	18K + 8k = 26K	Look up is required only for foreign key from R2 referencing R1.

Table 7.4 Memory Requirement

Relation transformed	loookup size	Remarks
R1	26K	we don't need lookup of R1

From table 7.2, table 7.3, table 7.4 we can calculate the maximum memory required as 26K when we use minimum weight and dynamic update of the weight strategy.

Lemma: The nodes having minimum weight should be selected first.

7.4.2.5 Weight Table:

In order to dynamically update the weight of relations after transformation of a relation, we maintain a weight table consisting of the in-degree, out-degree, tuplesize, status and weight of all relations. Whenever a relation is transformed into a graph, the in-degree and out-degree of all the relations referencing the transformed relation will be decreased by the number of foreign keys of the relation referencing and referenced by the transformed relation respectively. The table is then updated by recalculating the weights.

7.4.2.6 Evaluating Utility of Lookup using Weight Table:

While transforming a relation, we can check the out-degree of the relation. If the out-degree of the relation is zero then we do not need lookups for the relation. When all the relations adjacent to a earlier processed relation are transformed, lookups of such relation cease to be useful and the out-degree of the relation are set to zero. Accordingly, we delete the lookups for the transformed relations having an out-degree updated to zero.

7.4.2.7 DB2Graph Pseudo code:

The DB2Graph algorithm maintains a weight table and the RR-Graphs of relations (steps 1 and 2 of Algorithm 3) to select the relation with minimum weight (step 3) and for the dynamic update of weight (steps 46-48 of the Algorithm 3). The weight table is also used for evaluating the utility of lookup for the relations (steps 49 to 53). DB2Graph algorithm transforms the relational database into a graph according to the EntitytoAttributeGraph representation (steps 4 to 45 of the Algorithm 3). DB2Graph algorithm can handle composite primary (steps 23 to 45) and foreign keys (steps 8 to 22). It also handles the special case of the same attribute being part of primary and foreign key (step 39). The lookup of a tuple of a relation for DB2GraphGen contains all the primary key vertices(primary key values and vertex number of corresponding primary keys). Since, DB2Graph algorithm uses dynamic update of weights and determines the sequence of relations to be transformed (i.e., selecting minimum weight), we can conclude that DB2Graph requires less lookup space than the Naive algorithm.

7.4.3 SubdueDB2Graph Algorithm

SubdueDb2Graph transforms the relational database to graph in the form of EntitytoEntityGraph representation. SubdueDb2Graph, similar to DB2Graph determines the sequence of relation using dynamic update of weight in order to minimize the lookup space requirement. The lookup structure of a relation in SubdueDb2Graph consist of relationname vertex(tablename node) and primary key values for the tuples of the relation. While creating a subgraph for a tuple of a relation, SubdueDB2Graph algorithm checks lookup to see if the corresponding tablename node exists the lookup. If the correspond-

Algorithm 3 Db2Graph Algorithm

```
1: Maintain a RR-Graph of the relational database.
2: Maintain weight table consisting indegree,outdegree,tuplesize and weight of each relations of database.
3: Take a relation say Rc(table) with minimum weight from the weight table.
4: for all tuple (Tc) in the relation do
5:   Create a node for relation name (will be known as table name node).
6:   Create nodes for all the attribute values in the tuple except for foreign key attributes and primary key attribute.
7:   Connect the nodes with table name node (directed edge from relation name node to attribute value node) with
     edge label as attribute name.
8:   for all foreign key attribute fk in Rc do
9:     Find the relation Rp referenced by fk
10:    if lookup of Rp exists and if node fk exists in the lookup then
11:      Connect the table name node with the fk node from the lookup
12:    end if
13:    if lookup of Rp exists and if fk node does not exists in the lookup then
14:      Create a node for fk value connect it with table name node
15:      Insert the fk node in the lookup of Rp
16:    end if
17:    if lookup of Rp does not exist then
18:      Create a lookup for Rp
19:      Create a node for fk value and connect it with table name node
20:      Insert the fk node in the lookup of Rp
21:    end if
22:  end for
23:  for all primary key pk attribute in Rc do
24:    if pk is not also a foreign key in Rc then
25:      if lookup of Rc doesn't exist then
26:        Create a node for pk and connect it to table name node
27:        if lookup of Rc is required then
28:          Create a lookup for Rc
29:          Add pk node in the lookup
30:        end if
31:      else if node pk doesnot exist inlookup of Rc then
32:        Create a node for pk value and connect it to table name node
33:        if lookup of Rc is required then
34:          Add pk node in the lookup
35:        end if
36:      else if node pk is in the lookup of Rc then
37:        Connect the table name node with node fk from lookup
38:      end if
39:    else if pk is also a foreign key in Rc and if lookup of Rc is required then
40:      Find the relation Rp referenced by pk
41:      Find the node nv corrosponding to pk values in the lookup of Rp
42:      Add node nv as node pk in lookup of Rc
43:    end if
44:  end for
45: end for
46: In the weight table, decrease in-degree value of the relations having edges from Rc by the number of edges incident
    on the relation from Rc
47: In the weight table, decrease out-degree value of the relations having edges to Rc by the number of edges incident on
    Rc from the relation
48: Update the weight of the weight table
49: for all relations in weight table do
50:   if the relation is selected and the outdegree is 0 then
51:     remove the lookup of the relation
52:   end if
53: end for
54: Repeat step 3 till there are any relations left to be transformed in the database
```

ing tablename node is in the lookup, the node will be root of the subgraph(of a tuple) else a new root node i.e. tablename node for the subgraph(of a tuple) is created(step 5-9 of Algorithm 4). Then the attribute nodes for the tuple are created and connected to the root node(steps 10 to 45 of the Algorithm 4). The SubdueDB2Graph algorithm maintains a weight table and the RR-Graphs of relations (steps 1 and 2 of Algorithm 4) to select the relation with minimum weight (step 3) and for the dynamic update of weight (steps 46-48 of the Algorithm 4). The weight table is also used for evaluating the utility of lookup for the relations (steps 49 to 53). SubdueDB2Graph algorithm can handle composite primary (steps 27 to 45) and foreign keys (steps 12 to 26). It also handles the special case of the same attribute being part of primary and foreign key (step 40-41).

Algorithm 4 SubdueDb2Graph Algorithm

```
 1: Maintain a RR-Graph of the relational database
 2: Maintain weight table consisting indegree, outdegree, tuplesize and weight of each relations of database.
 3: Take a relation say Rc(table) with minimum weight from the weight table.
 4: for all For all tuple (Tc) in the relation do
 5:   if lookup of Rc exist and table name node corrosponding to primary key values of Tc exist then
 6:     Find the relation name(table name node) node rn corrosponding to primary key values of Tc
 7:   else
 8:     Create a node rn for relation name (will be known as table name node)
 9:   end if
10:   Create nodes for all the attribute values in the tuple except for foreign key attributes and primary key attribute
11:   Connect the nodes with rn (directed edge from relation name node to attribute value node) with edge label as
      attribute name.
12:   for each foreign key attribute fk in Rc do
13:     Find the relation referenced Rp by fk
14:     if lookup of Rp exists and if table name node tn with primary key values corrosponding to foreign key values of
        Tc exists in the lookup then
15:       Connect rn with the tn from the lookup
16:     end if
17:     if lookup of Rp exists and if table name node tn with primary key values corrosponding to Tc does not exists in
        the lookup then
18:       Create a table name node tn with primary key values corrosponding to foreign key values of Tc and connect
          tn with rn
19:       Insert tn in the lookup of Rp
20:     end if
21:     if lookup of Rp does not exist then
22:       Create a lookup for Rp
23:       Create a table name node tn with primary keys corrosponding to foreign keys of Tc and connect rn with tn
24:       Insert the tn node in the lookup of Rp
25:     end if
26:   end for
27:   for each primary key pk attribute in Rc do
28:     if outdegree of the relation in RR-Graph is greater than 0 then
29:       if pk is not also a foreign key in Rc then
30:         if lookup of Rc does not exist then
31:           Create a lookup for Rc
32:           Create a node for pk and connect it to rn
33:           Add rn with primary key values of TC in the lookup of Rc
34:         else if tablename node corrosponding to primary key values of Tc does not exist in lookup for Rc then
35:           Create a node for pk value and connect it to rn
36:           Add rn in lookup with primary key values of Tc
37:         else
38:           Create a node for pk and connect it with rn
39:         end if
40:       else if pk is also a foreign key in Rc then
41:         Add rn with primary key values of TC in the lookup of Rc
42:       end if
43:     end if
44:   end for
45: end for
46: In the weight table, decrease in-degree value of the relations having edges from Rc by the number of edges incident
    on the relation from Rc
47: In the weight table, decrease out-degree value of the relations having edges to Rc by the number of edges incident on
    Rc from the relation
48: Update the weight of the weight table
49: for all relations in weight table do
50:   if the relation is selected and the outdegree is 0 then
51:     Remove the lookup of the relation
52:   end if
53: end for
54: Repeat step 3 for all the relations in the database
```

CHAPTER 8

PERFORMANCE EVALUATION

This chapter gives an overview of the performance of Db2GraphGen. We will evaluate the intermediate space required by Naive, Db2Graph and SubdueDb2Graph algorithms of DB2GraphGen system. We will also compare the processing time required by these three algorithms.

8.0.4 Input parameters for Db2GraphGen

In this section we explain the input parameters for Db2GraphGen. Db2GraphGen requires Schema Information Extractor module for each RDBMS. We have developed Schema Information Extractor for MySQL and Oracle. The input parameters allows user to choose database platform and algorithm type. The input parameters and their explanations are given below.

```
#dbplatform  #algotype
```

dbmode: Allows user to select database platform 0-Oracle 1-MYSQL

graph type: Allows user to select graph representation type 0-DB2GraphGen algorithm, 1-SubdueDb2GraphGen algorithm, 2-Naive algorithm

8.1 Writing Log File

Transformation of relational database to graph is a time-consuming process and for some database it may take hours to complete. Further more, all the experiments should be done on same machine to analyze the performance of different approaches and algorithms. Hence, we implemented the log file to store performance of DB2GraphGen. The format of log file of Db2GraphGen with explanation is shown below.

```
Algorithm applied
 Relation transformed ----> Space required
   X1-----> N1 bytes
   X2-----> N2 bytes
 Total time:: T1
```

The Algorithm applied tells which algorithm was implemented, relation transformed provides the name of the relation transformed and space required provide the information about overall lookup space utilized after the relation was transformed. Total time provide the total time required to complete the transformation.

8.2 Experimental Results for DB2GraphGen

This section gives an overview of lookup space requirement and processing time for the Naive,DB2Graph and SubdueDb2Graph algorithm. Experiments were conducted on databases varying in size. Database relations were used to store the intermediate data (lookup data). Total lookup space used by lookups for each algorithm was measured after transformation of each relation in database. Also, time taken by each algorithm to perform transformation was measured. All of the 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 of the experiments were performed on a Linux machine using Oracle10G. The machine was running on dual processors with 2 GBytes of memory. Tab 8.1 provides the total lookup space utilized by Naive, Db2Graph and SubdueDb2Graph algorithms after transformation of each relation for the database with 14800 tuples and 120 attributes. Tab 8.2 provides the total lookup space utilized by these algorithms after transformation of each relation for the database with 19381 tuples and 120 attributes and Tab 8.3 provides the total lookup space utilization for the database with 41754 tuples ans 120 attributes. Tab 8.4 and Fig 8.1 shows maximum lookup utilized and Tab 8.5 and Fig 8.2 gives the processing time of these algorithms for the databases. In order to avoid duplicate lookups, DB2Graph algorithm has to refer lookup relations (tables) for every primary key and foreign key for each tuple of every relation. Further more, SubdueDb2Graph has to refer lookup to avoid duplicate tablename node for each tuple of each relation. Since, we are using database relations to maintain lookup, referring to lookup requires accessing lookup relations from database. Hence, processing relations with more number of foreign key is comparatively slower than processing relations with less number of foreign keys in SubdueDB2Graph and DB2Graph algorithm. In database with 41734 tuples, size of relations having more number of foreign keys was bigger(number of tuples in relation with 5 foreign key was 14,942 ,with 3 foreign key was 7,540 and with 2 foreign key was 8,008). Hence, SubdueDb2Graph algorithm required more processing time than naive algorithm in the database. Similarly difference in processing time of Naive algorithm and RDB2GraphGen algorithm was less for the database.

Table 8.1 Lookup Space for Database with 14800 tuples and 120 attributes

No of Relations Transformed	DB2Graph(KB)	SubdueDb2Graph(KB)	Naive(KB)
1	0.00	0.00	0.00
2	0.00	0.00	614.88
3	0.00	0.00	657.60
4	0.00	0.00	660.88
5	0.00	0.00	666.04
6	0.00	0.00	672.71
7	0.00	0.00	672.71
8	0.00	0.00	962.97
9	0.00	0.00	962.97
10	0.00	0.00	3770.78
11	1.23	1.23	3770.78
12	2.70	2.70	3771.02
13	5.98	5.98	3771.02
14	10.90	10.90	3771.02
15	10.90	10.96	3776.64
16	16.52	16.58	3776.64
17	23.20	23.26	3776.64
18	27.89	27.95	3776.64
19	80.57	80.63	3981.82
20	155.04	155.10	3986.80
21	186.33	186.39	4000.91
22	179.65	179.71	4002.14

Table 8.2 Lookup Space for Database with 19381 tuples and 120 attributes

No of Relations Transformed	DB2Graph(KB)	SubdueDb2Graph(KB)	Naive(KB)
1	0.00	0.00	0
2	0.00	0.00	547.85
3	0.00	0.00	957.33
4	0.00	0.00	960.61
5	0.00	0.00	965.77
6	0.00	0.00	972.45
7	0.00	0.00	972.45
8	0.00	0.00	1193.74
9	0.00	0.00	1193.74
10	0.00	0.00	4606.86
11	2.11	2.11	4606.86
12	3.57	3.57	4607.09
13	6.86	6.86	4607.09
14	12.48	12.48	4607.09
15	19.16	19.16	4612.72
16	78.63	78.63	4612.72
17	78.63	78.69	4612.72
18	135.12	135.18	4612.72
19	195.47	195.53	4819.83
20	195.47	195.53	4879.36
21	223.48	223.54	5048.03
22	216.80	216.86	5050.14

Table 8.3 Lookup Space for Database with 41754 tuples and 120 attributes

No of Relations Transformed	DB2Graph(KB)	SubdueDb2Graph(KB)	Naive(KB)
1	0.00	0.00	0.00
2	0.00	0.00	1619.92
3	0.00	0.00	1619.92
4	0.00	0.00	1623.20
5	0.00	0.00	1623.20
6	0.00	0.00	1629.88
7	0.00	0.00	1629.88
8	0.00	0.00	2906.48
9	0.00	0.00	2906.48
10	0.00	0.00	7575.86
11	0.00	0.00	7575.86
12	0.00	0.00	7576.09
13	0.00	0.00	7576.09
14	5.63	5.63	7576.09
15	11.84	11.84	7581.72
16	18.52	18.52	7581.72
17	81.04	81.04	7581.72
18	429.26	429.26	7581.72
19	429.26	429.32	7810.74
20	777.42	777.48	8159.02
21	808.71	808.77	9145.82
22	802.03	802.09	9152.03

Table 8.4 Lookup Space for Db2Graph, Naive and SubdueDb2Graph

Database tuple size	Attribute size	Naive	DB2Graph	SubdueDb2Graph
14800	120	4002 KB	186 KB	186 KB
19381	120	5050 KB	223 KB	223 KB
41754	120	9152 KB	808.71 KB	808.76 KB

Table 8.5 Processing Time for Db2Graph, Naive and SubdueDb2Graph

Database tuple size	Attribute size	Naive	DB2Graph	SubdueDb2Graph
14800	120	21.76 min	12.7 min	12.6 min
19381	120	30.48 min	17.15 min	17min
41754	120	78.41 min	75.61 min	93.42min

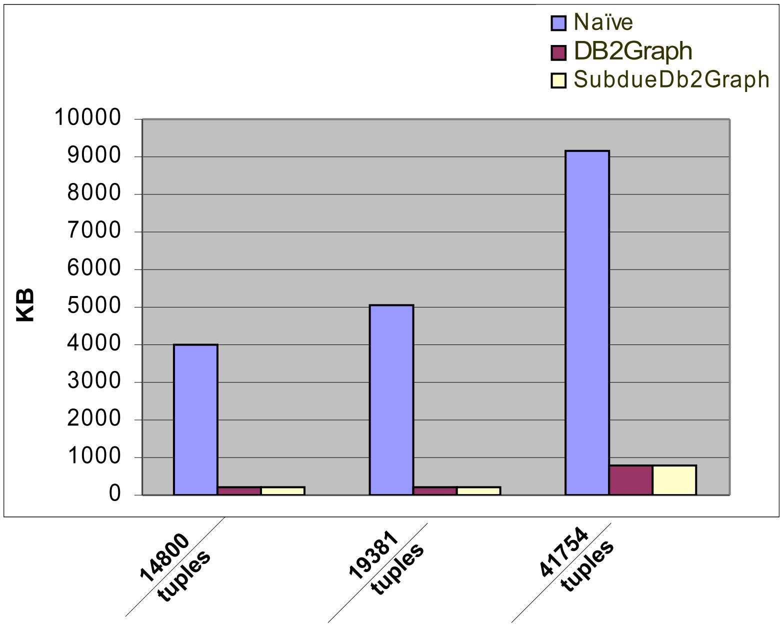


Figure 8.1 Lookup Space for Db2Graph, Naive and SubdueDb2Graph

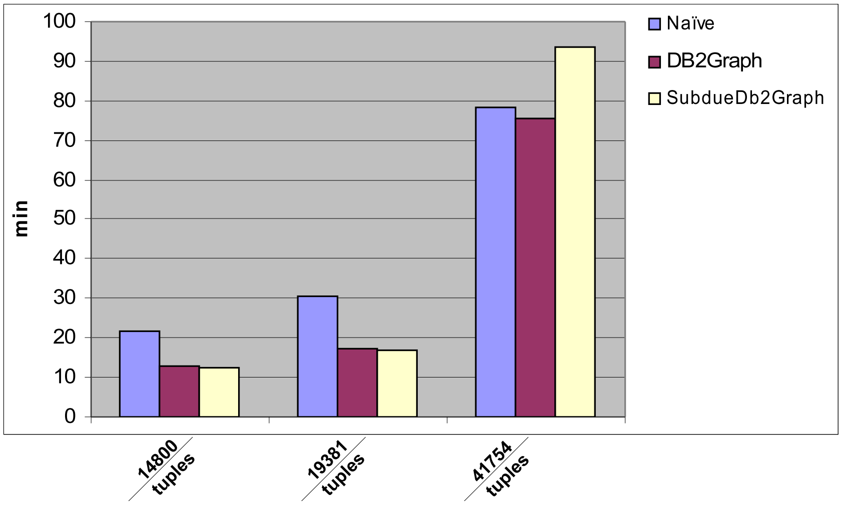


Figure 8.2 Processing Time for Db2Graph, Naive and SubdueDb2Graph

CHAPTER 9

CONCLUSION AND FUTURE WORK

In this thesis we have introduced relational database approach for frequent subgraph mining over a set of graphs. In order to accommodate a set of graphs for mining, we have extended the graph representation of HDB-Subdue. The graph representation proposed in this thesis can represent the most general form of graph including graphs with cycles and multiple edges between two vertices. This thesis also presents an alternative approach to detect duplicate substructures which is efficient than the pseudo duplicate elimination approach of HDB-Subdue. The major steps for frequent subgraph mining is i) Candidate Generation: In this step, all the probable frequent subgraphs are generated and ii) Frequency Counting and substructure pruning : In this step frequency of subgraphs are counted and subgraphs having frequency less than user specified value are pruned. Our approach also implements these steps for frequent subgraph mining. We implemented the unconstrained expansion of substructure instances to insure that all sub-graphs are considered during candidate generation. We joined tuples of instance_n table and oneedge table having same graph id to generate n+1 sized candidates(substructure instances). In order to avoid expansion of an instance on the edges that is already present in the instance, we introduced a new attribute called edge number. An edge number is unique to an edge. Hence, by imposing constraint that the new edge number should not be equivalent to any edge numbers in the substructure instance, we can avoid the expansion of instances on edges that is already present in the instance. Due to the unconstrained expansion, pseudo duplicates (same substructure instances in different order) were being generated. We used edge code to identify and eliminate the pseudo duplicate instances.

Edge code is a string formed by concatenating graph id and edge numbers ordered in ascending order. Hence, all pseudo duplicates of same instances will have same edge code. Thus, by retaining only one instance from the instances having same edge code, we eliminated the pseudo duplicates. Due to the unconstrained expansion there was the probability of similar substructure instances existing in different order. Hence, we performed canonical ordering on the vertex labels to identify such instances. In order to insure only one instance per substructure is considered for each graph, we projected distinct vertex labels, edge labels, connectivity map and graph id and stored them in dist_n table. Then we performed group by functions on dist_n table on attributes vertex labels, edge labels and connectivity attributes to get the substructures count of each substructure. Only the instances of the substructures with frequency satisfying the user specified support value were retained for further expansion.

This thesis also presents an efficient approach to infer structural relationship from relational database to facilitate graph mining. The approach developed for the task identifies the Entity-Relationship model [8] for the relational database and transforms an instance of a relational database into a graph using the inferred Entity-Relationship model [8]. In order to achieve maximum portability and platform independence, we modularize the approach. As part of the solution, new graph representations for relational representation have been presented. Several algorithms (DB2GraphGen and SubDueDB2GraphGen) have been discussed and developed to transform A relational database into a graph. These algorithms require less space as compared to the earlier approach.

The algorithms proposed in this thesis were tested for correctness and scalability. Graph mining algorithms were tested on data with and without cycles and with and without multiple edges on small to very large sized data sets. Similarly, the approach to infer structural relationship from relation database was tested on databases with 4 relations,14 tuples and 12 attributes to 22 relations, 41734 tuples and 120 attributes.

9.1 Future work

The frequent subgraph mining algorithm presented in this thesis 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 we implement the canonical ordering and connectivity map to detect exact graph match. Same concept might aid to perform the inexact graph match. Performance of the algorithm presented in this thesis need to be improved by optimizing the canonical ordering which is taking the bulk of the time. Incremental approach to FSG is another direction that can be pursued. 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. In the approach presented for inferring structural relationship from relational database, the transformation of data from relational databases to graphs is done by considering one tuple of a relation at a time. Hence, for each tuple, we need to access the lookup for the respective primary and foreign keys associated with it. Also, the current implementation addresses the scalability issue by using database relations for lookup at the expense of execution time. The next enhancement would be to modify the algorithms to reduce the cardinality of lookup access as well as utilizing main memory along with database relations for maintaining the lookup.

REFERENCES

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

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

[3] 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.

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

[5] 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]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Ramji.pdf

[6] 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]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Ramanathan.pdf

[7] S.Padmanabhan, "Hdb-subdue, a relational database approach to graph mining and hierarchial reduction," Master's thesis, Department of Computer Science and Engineering,University of Texas at Arlington: Arlington, 2004.

[8] R.Elmasri and S.B.Navathe, *Fundamentals of Database Systems*, 2nd ed. Benjamin Cummings, 1994.

[9] J.F.Sowa, "Conceptual graphs summary," in *Conceptual Structures: Current Research and Practice*, L.L.Gerholz, T.E.Nagle, J.A.Nagle, and P.W.Eklund, Eds. Ellis Horwood, 1992, pp. 3–51.

[10] S.Polovina and J.Heaton, "An introduction to conceptual graphs," *AI Expert*, vol. 7(5), pp. 36–43, 1992.

[11] 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.

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

[13] 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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] S.Palod, “Transformation of relational database domain into graphs based domain for graph based data mining,” Master’s thesis, Department of Computer Science and Engineering,University of Texas at Arlington: Arlington, 2004.

[20] G.Bhalotia, A.Hulgeri, C.Nakhe, S.Chakrabarti, and S.Sudarshan, “Keyword searching and browsing in databases using banks,” *ICDE ’02*, p. 431, 2002.

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

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

BIOGRAPHICAL STATEMENT

Subhesh K. Pradhan was born in Kathmandu, Nepal. He received his B.E. degree from Kathmandu University in April 2001. In the Spring of 2004, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington, in August 2006.