RELATIONAL APPROACH TO MODELING

AND IMPLEMENTING SUBTLE ASPECTS

OF GRAPH MINING

The members of the Committee approve the master's
thesis of Ramanathan Balachandran

Sharma Chakravarthy Supervising Professor	___________________________________
Lawrence Holder	___________________________________
Jean Gao	___________________________________

RELATIONAL APPROACH TO MODELING AND IMPLEMENTING SUBTLE ASPECTS OF GRAPH MINING

by

RAMANATHAN BALACHANDRAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2003

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me ample guidance and support through the course of this research.

I would like to thank Dr. Lawrence Holder and Dr. Jean Gao for serving on my committee.

I am grateful to Uday Koorella, Raman Adaikkalavan, Sridhar Reddy Varakala, Himavalli Kona, and Manu Aery for their invaluable help and advice during the implementation of this work. I would like to thank all my friends in the ITLAB for their support and encouragement.

I would like to acknowledge the support by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-0543), and the NSF (grants IIS-012370 and IIS-0097517) for this research work.

I would also like to thank my family members for their endless love and constant support throughout my academic career.

Nov 19, 2003

ABSTRACT

RELATIONAL APPROACH TO MODELING AND IMPLEMENTING SUBTLE ASPECTS OF GRAPH MINING

Publication No.____

Ramanathan Balachandran, M.S.

The University of Texas at Arlington, 2003

Supervising Professor: Sharma Chakravarthy

Data mining aims at discovering important and previously unknown patterns from datasets. Database mining performs mining directly on data stored in Data Base Management Systems. Complex relationships in data can be represented properly using graphs. As a result, graph mining can be used to mine data that have structural components.

The focus of this thesis is to support all aspects of graph mining by enhancing the algorithms previously developed (DB-Subdue) for graph mining using relational DBMS. The enhancements addressed in this thesis include: handling of cycles in the input, handling of overlapping substructures and their effect on compression, development of an MDL (minimum description length) equivalent for the relational approach, and inclusion of inexact graph matching. For some of the above, multiple approaches have been developed and tested. Extensive performance evaluation has been conducted to evaluate the extended algorithms and compare them with the main memory counterpart. Scalability has been addressed by exploring graphs of different sizes and their computation requirements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Databases have been used to store enormous amounts of data. It is important to extract useful knowledge from the stored data to aid in decision-making. Database mining tasks help in discovering patterns that are difficult to find manually. Database mining has been a topic of research for quite some time [1-6]. Most of the work in database mining has concentrated on discovering association rules from transactional data represented as (binary) relations. In contrast, the work on mining (from a file) spans a wide spectrum – from classification and clustering to mining structural data that represents relationships. The ability to mine over graphs is important, as graphs are capable of representing complex relationships. Graph mining is different from conventional mining approaches such as association rules and clustering. Graph mining uses the natural structure of the application domain and mines directly over that structure (unlike others where the problem has to be mapped to transactions or other representations). Graphs can be used to represent structural relationships in many domains (web topology, protein structures, chemical compounds, relationship of related actions for a concept such as fraud or money laundering, etc). Subdue [7] is a mining approach that works on graph representation.

Subdue identifies concepts describing interesting and repetitive substructures within the structural data. Subdue uses the principle of minimum description length [8] (or MDL) to evaluate the substructures. The MDL principle has been used for decision tree induction [9], pattern discovery in bio-sequences [10], image processing [11-13], concept learning from relational data [14], and learning models of non-homogeneous engineering domains [15]. The minimum description length principle, described by Rissanen [16], states that the best theory to describe a set of data is a theory that minimizes the description length of the whole data set. It also uses the concept of inexact graph match [17] to improve substructure discovery. Main memory data mining algorithms typically face two problems with respect to scalability. Graphs could be larger than the main memory available and hence cannot be loaded into main memory or the algorithm could be computationally expensive and the computation space required is more than the available main memory. Although Subdue provides us with a tool for mining interesting and repetitive substructures within the data, it is limited by the fact that it is a main memory algorithm. The algorithm 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. The adjacency matrix is computed for the graph, the substructure and also the compressed graph (compressed by the substructure). This adjacency matrix computation is done for each substructure and this contributes substantially to the time taken by the subdue algorithm for substructure discovery.

In summary, the major drawback of Subdue main memory algorithm is its scalability to large problems. This has motivated the development of graph mining algorithms (specifically Subdue class of algorithms) using SQL and stored procedures using Relational database management systems (RDBMSs). Representation of a graph, and its manipulation – generation of larger subgraphs, checking for exact and inexact matches of subgraphs – are not straightforward in SQL. The input (which is a graph) has to be represented using relations and operations have to use joins (or other relational operations) for mining repetitive substructures. At the same time they need to avoid manipulations that are known to be inefficient (e.g., correlated subqueries, cursors on large relations, in-place inserts and deletes from a relation). The DBMS version of Subdue (DB-Subdue) [18] was developed for the DB2 database and uses C, as the Subdue code was written in C. It explains the mapping of substructures to tuples in the database. Further, it elaborates on a suite of algorithms that have been carefully designed to achieve functionality as well as scalability. The experiments show that it can mine graphs with millions of vertices and edges with sub-linear growth in the computation time. DB-Subdue has several approaches for graph mining, such as the cursor-based approach, the UDF based approach and the enhanced cursor-based approach. The enhanced cursor-based approach, which was the last approach implemented, proved that the DB-Subdue scales better for larger graphs as compared to the main memory version. The crossover point is for an input graph with 500 vertices and 1000 edges. It uses pure SQL statements and indexing techniques to improve the performance of the algorithm. DB-Subdue was able to mine data sets with 800K vertices and 1600K edges.

DB-Subdue was a first attempt to support graph mining using an RDBMS. The goal was to demonstrate the feasibility and scalability of the approach. As a result, some of the subtle aspects of graph input (e.g., presence of cycles, presence of overlap) were not considered. Also, a very simplistic version of the MDL principle, based only on subgraph frequency, was used. And inexact (or isomorphic) matches were not considered. The details of DB-Subdue can be found in [18].

As a result, although DB-Subdue solved the scalability problem it did not address all aspects of graph mining as compared to Subdue. First and foremost, there was no way to distinguish two substructures having the same number of edges and vertices and the same number of instances. The *frequency* heuristic of DB-Subdue would rank both substructures equally. Therefore, the frequency heuristic does not always find the best substructure among same edge length substructures as it is based only upon frequency. Hence, it may not detect the best substructure in the whole graph, as the frequency can be same for different edge length substructures. The DBMS system always detected overlapping instances among substructures, which need to be differentiated for some applications. Without this differentiation, the frequency of the substructures will always be high which in turn affects the MDL used for identifying the best substructure.

One of the subtle and important aspects of graph mining algorithms is the handling of cycles in the input. DB-Subdue detects cycles properly but keeps expanding them indefinitely. So, the same vertices in the smaller substructure are repeated in the higher substructure. The algorithm 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. DB2 does not support declaring an array of host variables and hence the generalization for the algorithm could not be achieved. Separate host variables had to be declared for each pass.

The DB-Subdue algorithm handles only exact matches between subgraphs. In most of the real-life applications, such as discovering chemical compounds or other domains, there are very few scenarios in which exact graph matches are suitable. There are more subgraphs in the graph that differ from each other by a small extent. Therefore for practical purposes two graphs that differ by a small extent should be considered as match with each other.

1.1 Focus of this thesis

First and foremost, DB-Subdue was 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 the DB2 database. This thesis extends the DB-Subdue to address those aspects of graph mining that were not considered in DB-Subdue. This thesis proposes a method to solve the cycles and the overlap problem among the substructures. The main focus is to carefully analyze the substructures to detect these patterns among the data. The thesis also addresses a new technique for evaluating the substructures, which is both scalable and able to distinguish the best substructure among substructures of equal edge length and frequency. This gives an edge over both the *MDL principle* of Subdue as it is scalable and the *frequency* heuristic of the DB-Subdue as it is more accurate. This thesis also gives a viable solution for performing inexact graph match between two subgraphs apart from the exact graph match present in DB-Subdue.

The rest of this thesis is organized as follows. CHAPTER 2 discusses the background and related work in the field of graph-based data mining and the various approaches used for mining structural data. CHAPTER 3 describes the scalable approach developed in the DB-Subdue algorithm. CHAPTER 4 discusses the design issues for the various enhancements that have been added to DB-Subdue. CHAPTER 5 presents implementation details of all the issues that are explained in the design chapter. CHAPTER 6 presents performance evaluation including comparison with the Subdue algorithm. 0 concludes the thesis with emphasis on future work.

CHAPTER 2

RELATED WORK

A significant amount of work has been done in the field of graph-based data mining. This chapter mainly discusses two systems that support graph-based data mining. The first is the *Subdue knowledge discovery* system [7], which is a main memory algorithm. The other system is called the Frequent Subgraphs approach [19]. This chapter gives an insight into both of these systems. It briefly explains the concepts and other parameters associated with the two algorithms.

2.1 Subdue (Main memory algorithm)

The Subdue algorithm is the first of the two systems that will be discussed in the following sections. It is the first algorithm developed for substructure discovery in structural data.

2.1.1 Structural Data Representation

The substructure discovery system represents data as a labeled graph. Objects in the data are represented either as vertices or as small subgraphs and the relationships between them are represented as edges. A substructure is a connected subgraph within a graph. For the graph shown in Figure 2.1, the sample input for Subdue is shown in Table 2.1. An instance of the substructure in the graph is a set of vertices and edges that have the same vertex labels, edge labels and edge directions as that of the substructure. An edge can be either directed or undirected.

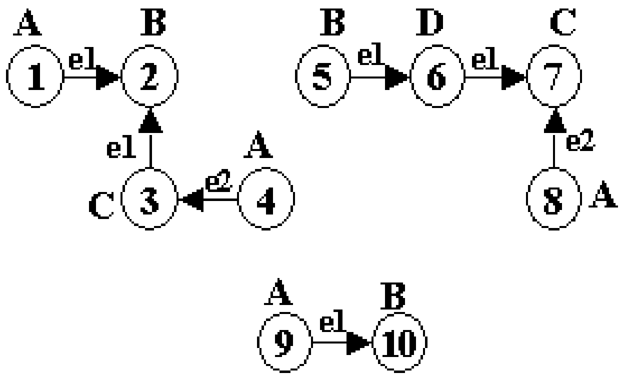
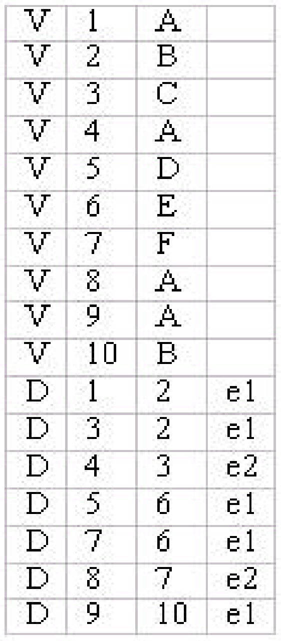


Figure 2.1 Input Graph

Table 2.1 Input File
for Subdue



V	1	A	
V	2	B	
V	3	C	
V	4	A	
V	5	D	
V	6	E	
V	7	F	
V	8	A	
V	9	A	
V	10	B	
D	1	2	e1
D	3	2	e1
D	4	3	e2
D	5	6	e1
D	7	6	e1
D	8	7	e2
D	9	10	e1

2.1.2 Parameters for Control Flow

The input to the discovery process is taken from the file as shown in Table 2.1 and the graph is constructed using these values. A number of parameters control the working of the algorithm. They are briefly described below:

1. Limit: This parameter specifies the number of different substructures to be considered in each iteration. The default value is (Number of Vertices + Number of edges)/ 2.
2. Iterations: This parameter specifies the number of iterations to be made over the input graph. The best substructure from the previous iteration is taken to compress the graph for the next iteration. The default value is one, i.e., no compression.
3. Threshold: This is a parameter that provides a similarity measure for the inexact graph match. Threshold specifies how different one instance of a substructure can be from the other instance. The instances match if matchcost(sub,inst) <= size(inst) * threshold. The default value is 0.0, which means that the graphs match exactly. A vary large value of Threshold may not be meaningful as it will match two dissimilar graphs. Currently, Subdue supports threshold values up to 0.3.
4. Nsubs: This argument specifies the maximum number of the best substructures returned.
5. Prune: If this parameter is specified, Subdue will discard child substructures that have value lesser than their parent substructure. This will reduce the search space to a great extent.
6. Beam: This argument specifies the maximum number of substructures to be kept in the substructure list. The default value is 4.

7. Overlap: This parameter guides the algorithm to consider overlap in the instances of the substructures. Two or more instances of a substructure are said to overlap if they have a common substructure. Overlap plays a significant role in calculating the compression value because with overlap we have to maintain extra information. During graph compression an OVERLAP_<iteration> edge is added between each pair of overlapping instances, and external edges to shared vertices are duplicated to all instances sharing the vertex.
8. Size: This parameter is used to limit the size of the substructures that are considered. Size refers to the number of vertices in the substructure. A minimum and maximum value is specified that determines the range of the size parameter.
9. Output: This parameter controls the screen output of Subdue. The various values are
    1) Print the best substructure in that iteration.
    2) Prints the best n substructures, where n is the number specified in the nsubs parameter.
    3) Print the best n substructures, and intermediate substructures as they are discovered.
    4) Print the best n substructures along with their instances and intermediate substructures as they are discovered.
    5) Only for Supervised Subdue: prints the substructures found in the negative graph along with the output printed by – 4 option.

2.1.3 Compression

The first compression scheme is called minimum description length (MDL). MDL, described by Rinssanen [16], states that the best theory to describe a set of data is a theory that minimizes the description length of the whole data set. There are various applications in which the MDL principle has been used. Some of them are decision tree induction, image processing etc. The principle is based on the number of bits needed to represent the substructure. The best substructure is the one that minimizes DL(S) + DL(G|S), where S is the discovered substructure, G is the input graph, DL(S) is the number of bits required to encode the required substructure, and DL(G|S) is the number of bits required to encode the input graph G after it has been compressed using substructure S. DL(G) represents the number of bits required to represent the input graph G. The final MDL value is defined as

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

The compression value is defined as

Compression = 1 / MDL

The second compression scheme is based only on Size. This theory uses simple and more efficient method but it is less accurate as compared to the MDL metric. The value of a substructure S in graph G is

Size (G) / (Size (S) + Size (G|S))

Here,

Size (G) = Number of vertices (G) + Number of edges (G)

Size (S) = Number of vertices (S) + Number of edges (S)

$$\textit{Size}(G|S) = (\textit{Number of Vertices}(G) - i*\textit{Number of Vertices}(S) + i) + (\textit{Number of Edges}(G) - i*\textit{Number of edges}(S))$$

where, G is the input graph, S is the substructure and G|S is the input graph after it has been compressed by the substructure and i is the number of instances of the substructure.

2.1.4 Inexact graph Match

Though exact graph match comparison discovers interesting substructures, most of the substructures in the graph may be slight variations of another substructure. In order to detect these, the algorithm developed by Bunke and Allerman [17] is used where each distortion is assigned a cost. A distortion is defined as the addition, deletion or substitution of vertices or edges. The two graphs are said to be isomorphic as long as the cost difference falls within the user specified threshold. This algorithm is an exponential algorithm as it compares each vertex with every other vertex in the graph. The branch and bound approach used by Subdue makes the algorithm computationally bound as the number of mappings considered are quite less when compared to all the possible mappings.

2.1.5 Subdue Algorithm

The Subdue [7] algorithm is presented below

```
1) Subdue(Graph, BeamWidth, MaxBest, MaxSubSize, Limit )
2) ParentList     = { }
3) ChildList = { }
4) BestList  = { }
5) ProcessedSubs  = 0
6) Create a substructure from each unique vertex label and
   its single-vertex instances; insert the resulting
   substructures in ParentList
7) while ProcessedSubs <= Limit and ParentList is not empty
   do
```

```
 8) while ParentList is not empty do
 9)          Parent = RemoveHead( ParentList)
10)          Extend each instance of Parent in all possible
             ways
11)          Group the extended instances into Child
             substructures
12)     for each Child do
13)          if SizeOf( Child ) <= MaxSubSize then
14)          Evaluate the Child
15)               Insert Child in ChildList in order by
                  value
16)          if Length( ChildList ) > BeamWidth then
                  Destroy the substructure at the end of
                  ChildList
17)     ProcessedSubs = ProcessedSubs + 1
18)     Insert Parent in BestList in order by value
19)     if Length( BestList ) > MaxBest then
             Destroy the substructure at the end of BestList
20) Switch ParentList and ChildList
21) return BestList
```

2.1.6 Flow of the Algorithm

The algorithm starts with the initialization of Parent List, Child List and Best List to empty sets. The Parent List consists of substructures to be expanded. The child list contains the substructures that are expanded. The Best list contains the best substructures found so far. The ProcessedSubs gives the total number of substructures processes so far. All the unique vertex labels in the graph are inserted into the Parent list. The second while loop is the major part of the algorithm. Each substructure is taken from the Parent list and is expanded in all possible ways either by adding a vertex and an edge or an edge if both vertices are already present. The first instance of the new substructure becomes the new child substructure. All the other child instances that were expanded in the same way become instances of the child substructure. Each child substructure is evaluated using the MDL heuristic and inserted in the child list in the decreasing order of MDL value. The beam value is enforced on all the three lists. This removes all substructures above the beam value; as these will not be participate in future expansions. The Parent list is now swapped with the child list, which is then considered for extensions.

The input graph is compressed by replacing each instance of the best substructure by a single node. The resulting input graph is then used for the next iteration to find more interesting substructures. This process continues until the number of iterations specified by the user is reached, or the algorithm fails to find a substructure that compresses the graph.

There are many different halting conditions for the algorithm that are determined by the parameters given by the user. The Limit parameter plays a major role, as it limits the total number of substructures processed so far. The prune parameter discards all child substructures having value lesser than their parents, also acts as a halting condition when there are no child substructures left in the child list after pruning. Another means of halting the algorithm is by using the size parameter. For example, if the maximum value of size is set to 4, then no substructures with vertices greater than 4 will be discovered.

2.2 Frequent Subgraphs

Discovery of subgraphs or substructures forms the essential part of most Graph-based Data mining algorithms. Substructure discovery can be used to form association rules, extract repetitive patterns and for classification. One such algorithm is Frequent Subgraphs (FSG) [19]. FSG is used to discover subgraphs that occur frequently in a data set of graphs. FSG is used to find all the frequent substructures that occur in a graph database. It is designed on the lines of Apriori algorithm. The key features of this algorithm are the optimizations associated with it.

2.2.1 Candidate Generation

Candidate generation is a process of selecting substructures that are frequent in the graph. FSG starts by noting all the frequent 1-edge and the 2-edge subgraphs in the graph database. It finds the frequent k edge substructures and then generates the candidates for the frequent substructures of size k+1. Two frequent k size substructures are joined to produce the k+1 size candidate substructures. For two k size frequent substructures to be joined, they must have the same 'core', which means they must have a common k-1 size subgraph. Thus, during each pass the best k size frequent subgraphs are generated. FSG does not need the adjacency list in the candidate generation phase as the candidates are generated using the existing frequent substructures. The problem with this type of candidate generation is that some of the generated candidates may not be present in graph at all. FSG uses 'count', which indicates the number of graphs in which a particular substructure exists, as the heuristic to judge the relative values of subgraphs and to decide if the candidates are frequent or not. An example of two four-edge substructures is shown in Figure 2.2 and Figure 2.3.

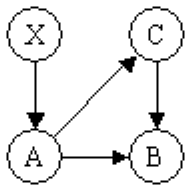


Figure 2.2 Example four edge substructure

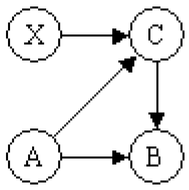


Figure 2.3 Example four edge substructure

These substructures are extended to form two candidate five edge substructures, which are shown in Figure 2.4 and Figure 2.5.

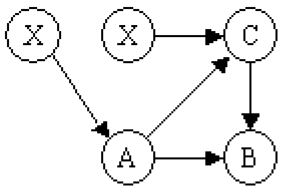


Figure 2.4 Five edge substructure extension

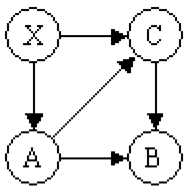


Figure 2.5 Another five edge substructure extension

FSG prunes the candidate substructures using the 'downward closure property', which allows a k+1 size substructure to be a candidate only if all its k subgraphs are frequent substructures.

2.2.2 Graph Isomorphism

Graph isomorphism plays a key role in discovering frequent substructures. FSG uses canonical labeling for isomorphism. Canonical labeling [20, 21] assigns a unique code for each substructure and two substructures have the same canonical code only if the substructures are isomorphic. Canonical labeling is an easier and faster way of finding the isomorphic substructures as it does not have to refer to the input graph for computing the canonical labels, but it suffers from the fact that canonical labeling cannot be used for graphs that have multiple edges between the vertices. The concept of inexact graph match in which graphs might match within a threshold value cannot be implemented by canonical labeling.

CHAPTER 3

OVERVIEW OF DB-SUBDUE

This chapter discusses DB-Subdue [18]. This is the first database version of Subdue (main memory algorithm). The algorithm has several approaches but this thesis will only discuss the last approach, which is called the enhanced cursor-based approach.

3.1 Graph Representation and Generation of Substructures

This section describes how graphs are represented in a database. Since databases only have relations, we need to convert the graph into tuples in a relation. The vertices in the graph are inserted into a relation called Vertices and the edges are inserted into a relation called Edges. The input is read from a delimited ASCII file and loaded into the relations. For the graph shown in Figure 2.1, the corresponding vertices and the edges relations are shown in Table 3.1 and Table 3.2. The joined_base relation will consist of all substructures of size one size representing the number of edges. The new relation joined_base is created because the edges relation does not contain information about the vertex labels. So the edges relation and the vertices relation are joined to get the joined_base relation.

Table 3.1 Tuples in the edges relation

Vertex1	Vertex2	EdgeName
1	2	e1
3	2	e1
4	3	e2
5	6	e1
6	7	e1
8	7	e2
9	10	e1

Table 3.2 Tuples in the vertices relation

VertexNo	VertexLabel
1	A
2	B
3	C
4	A
5	B
6	D
7	C
8	A
9	A
10	B

For a single edge substructure, the edge direction is always from the first vertex to the second vertex. For a higher edge substructure, we need to know the directionality of the edges between the vertices of the substructure. In case of a two-edge substructure, the third vertex can either be expanded from the first vertex or the second vertex. Therefore a new attribute called the extension attribute (extn) has been introduced for each expansion of the substructure. For example, if the 5 $\rightarrow$ 6 substructure in Figure 2.1 is extended to get the 5 $\rightarrow$ 6 $\rightarrow$ 7substructure then the extension attribute will have value 2, indicating that the third vertex is extended from the second vertex. The edge direction can be either into the vertex or away from the vertex. This is represented by a negative value in the extension attribute. For example, if 1 $\rightarrow$ 2substructure in Figure 2.1 is expanded to 1 $\rightarrow$ 2 $\leftarrow$ 3 then the extension attribute will have value –2, indicating that the third vertex is expanded from the second vertex and the edge is going towards the second vertex. In general, if the extension i (attribute ext) is –j, then the edge i+1 is from vertex i+2 to j. For a substructure of size n, we need n-1 extension attributes.

Table 3.3 joined_base relation

vertex1	vertex2	vertex1name	vertex2name	edgename
1	2	A	B	e1
3	2	C	B	e1
4	3	A	C	e2
5	6	B	D	e1
6	7	D	C	e1
8	7	A	C	e2
9	10	A	B	e1

3.2 DB-Subdue Algorithm

The algorithm used for DB-Subdue is shown below:

```
1)   DB-Subdue (input file, size)
2)   Load vertices into vertices table;
3)   Load edges into edges table;
4)   join vertices and edges table to create and populate
     joined_1 table
5)   i = 2
6)   WHILE(i<size)
7)        Compute   joined_i table (substructures of size i)
          from beam_joined_i-1, joined_base
8)        Create frequent_i table using group by clause
9)        DECLARE   Cursor c1 on frequent_i order by count
10)       DECLARE   Cursor c2 on Frequent_i order by count
11)       WHILE (c1.count < beam)
12)            FETCH     c1   into g1
13)            WHILE(c2)
14)                 FETCH     c2   into g2
15)                 If (!Isomorphic (g1, g2) =0)
16)                      Insert c1 into frequent_beam_i
17)       Insert into beam_joined_i
```

```
          from frequent_beam_i,joined_I
18)  i++
```

3.3 Algorithm Flow

The algorithm starts with initializing the vertices and the edges table. The joined_base table is then constructed by joining the vertices and edges table. The joined_base table will be used for future substructure expansion. The joined_1 table is a copy of the joined_base table. The frequent_1 table is created to keep track of substructures of size 1 and their counts. Here, size refers to the number of edges as opposed to Subdue main memory algorithm in which size refers to the number of vertices. Projecting the vertex labels and edge labels attributes on the joined_1 table and then grouping them on the same attributes, produces the count for the frequent_1 table as shown in Table 3.4. Therefore the frequent_1 table does not have vertex numbers. Group by clause is used so that all the exact instances of the substructure are grouped as one tuple with their count updated.

Table 3.4 frequent_1 relation

vertex1name	Vertex2name	edge1	count1
A	B	e1	2
A	C	e2	2
B	D	e1	1
C	B	e1	1
D	C	e1	1

The pruning of substructures with a single instance corresponds to deleting the tuples with count value 1 from the frequent_1 relation. Since these substructures do not contribute to larger repeated substructures, they have to be deleted from the joined_base relation as well, so that further expansions using that relation do not produce substructures with single instances (there may be 1 instance substructures produced later when repeated substructures of length i do not grow to repeated substructures of length i + 1). The updated frequent_1 relation is shown in Table 3.5 and the resultant joined_base relation is shown in Table 3.6.

Table 3.5 Updated frequent_1 relation

vertex1name	vertex2name	edge1	count1
A	B	e1	2
A	C	e2	2

Table 3.6 Updated joined_base relation

vertex1	vertex2	vertex1name	vertex2name	edgename
1	2	A	B	e1
4	3	A	C	e2
8	7	A	C	e2
9	10	A	B	e1

The substructures are then sorted in decreasing order of the count attribute and the beam number of substructures is inserted into another table called the frequent_beam_1 table. Only the substructures present in this table are expanded to larger substructures. Since all instances of the substructure are not present in the frequent_beam_1 table, this table is joined with the joined_1 table to construct the joined_beam_1 table. The joined_beam_1 is in turn joined with the joined_base table to generate the two edge substructures.

3.4 Extending to higher-edge substructures

The joined_beam_1 relation contains all the instances of the single edge substructure. Each single edge substructure can be expanded to a two-edge substructure on any of the two vertices in the edge. In general, an n edge substructure can be expanded on n + 1 vertices in the substructure. All possible single edge substructures are listed in the joined_base relation. So by making a join with the joined_base relation we can always extend a given substructure by one edge. In order to make an extension, one of the vertices in the substructure has to match a vertex in the joined_base relation. The following query extends a single edge substructure in all possible ways,

```
Insert into joined_2(vertex1,vertex2,vertex3,
                    vertex1name,vertex2name,vertex3name,
                    edge1name,edge2name,ext1)
                   (Select    j1.vertex1,j1.vertex2,j2.vertex2
                              ,j1.vertex1name,j1.vertex2name,
                              j2.vertex2name,j1.edge1name,
                              j2.edge1name,1
                    From      joined_1 j1, joined_base j2
                    Where     j1.vertex1 = j2.vertex1 and
                              j1.vertex2 != j2.vertex2
                    Union
                    Select
                    j1.vertex1,j1.vertex2,j2.vertex2,
                              j1.vertex1name,j1.vertex2name,
                              j2.vertex2name,j1.edge1name,
                              j2.edge1name,2
                    From      joined_1 j1, joined_base j2
                    Where          j1.vertex2 = j2.vertex1
                    Union
                    select    j.vertex1,  j.vertex2,
                              j1.vertex1,  j.vertex1name,
                              j.vertex2name, j1.vertex1name,
                              j.edge1,  j1.edge1,-2
                    From      joined_1 j, joined_base j1
                    Where     j.vertex2 = j1.vertex2  and
                              j.vertex1 != j1.vertex1 )
```

In the above there are 3 queries (and 2 unions). Since there are 2 nodes (in a one-edge substructure), the first query corresponds to the positive extension of the first node, and the other two correspond to the positive and negative extensions of the second node. In general, for a substructure with n nodes, there will be (n-1)*2 +1 queries and (n-1)*2 unions. The reason for not performing the negative extension on the first node is that it will be covered by the positive expansion of the node whose edge is coming into it. Also, every node is a starting point for expanding it to a larger size substructure. Hence the incoming edges of the starting node need not be considered.

The rest of the steps are the same as the single edge substructure expansion, except for the addition of a new extension attribute. The n-1 edge substructures are stored in the joined_beam_n-1 relation. In order to expand to n edge substructures, an edge is added to the existing n-1 edge substructure. Therefore, the joined_beam_n-1 relation is joined with the joined_base relation to add an edge to the substructure. The frequent_n relation is in turn generated from the joined_n relation. The frequent_beam_n relation contains the beam tuples. The joined_beam_n is then generated which has only the instances of the beam tuples. There main halting condition for the algorithm is the user-specified parameter – the max size (limit). Once the algorithm discovers all the substructures of the max size the program terminates.

3.5 Limitations of the algorithm

There are a number of limitations of the database approach. A few of them are enumerated below:

3.5.1 Number of columns

The approach presented in DB-Subdue uses a tuple to represent a subgraph. This means that the number of attributes in the relation will grow as the size of the substructure increases. This may eventually place a limit on the size of the maximum substructure that can be detected, as there is a limit to the number of columns a relation can have in a Relational database. It is 500 for the DB2 database system. joined_n relation would need 4n+2 attributes for describing an n edge substructure. The vertex names would need n+1 attributes, the vertex numbers would need n+1 attributes, the edge would need n attributes and the extensions would need n-1 attributes and a count attribute. So the algorithm could discover substructures of size 124 at the most. It may vary for other commercial databases.

3.5.2 Cursors

Cursors are used for implementing the beam. The host variables are declared for exchanging data between the database and the host programming language using the cursors. DB2 (in contrast to Oracle) does not support declaring an array of host variables and hence, a generalization for the algorithm cannot be achieved. Separate host variables have to be declared for each pass. This is an implementation inconvenience rather than the limitation of the approach.

CHAPTER 4

DESIGN ISSUES

This chapter gives a detailed explanation of the design issues involved in modeling certain aspects of graph mining. Some of the aspects that will be discussed in the chapter are the evolution of a heuristic to distinguish same signature substructures, handling cycles and overlap in graphs effectively, and graph isomorphism.

4.1 Database Minimum Description Length (DMDL)

Database Minimum description length principle (DMDL) is a heuristic based on the minimum description length principle (MDL) [8]. This section explains the importance of the DMDL heuristic and the advantages of the same over the frequency heuristic of DB-Subdue and the MDL heuristic of Subdue.

4.1.1 Need for a new heuristic

The primary motivation to develop a new heuristic was to bring the metrics characteristics to the DBMS environment and strike a balance between accuracy and run-time computational cost. Although the MDL principle is very accurate it takes a long time to evaluate substructures in large datasets. On the other hand, DB-Subdue's frequency heuristic scales well for large datasets, but the accuracy is not good as compared to the MDL principle. DB-Subdue cannot distinguish between substructures that have the same number of vertices and edges and having the same frequency of occurrence. These attributes form the **signature** of a substructure. This motivated the need for a heuristic that could distinguish between same signature substructures and yet achieve scalability. Database Minimum description Length (DMDL) principle proposes to achieve both these criteria.

4.1.2 Adjacency Matrix

The adjacency matrix in the MDL principle plays a vital role in distinguishing between two substructures with the same signature. The DMDL principle partly uses the adjacency matrix to differentiate same signature substructures. For example, in the graph shown in Figure 4.1, there are two substructures, each appearing twice in the graph.

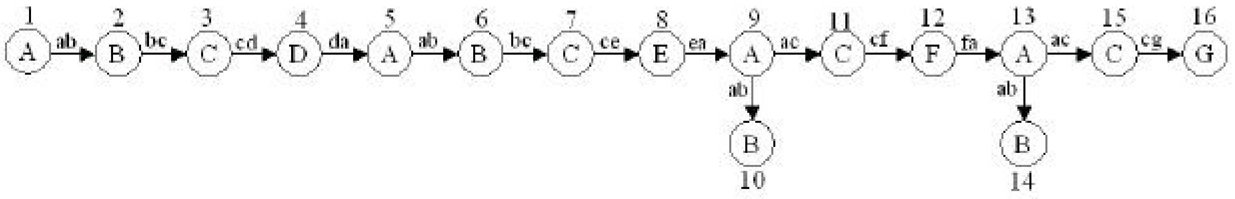


Figure 4.1 Input Graph Example

Figure 4.2 (graph1) and Figure 4.3 (graph2) shows two same-signature substructures and the explanation regarding the importance of the adjacency matrix in distinguishing these substructures follows the figures.

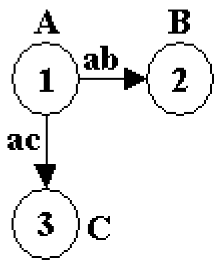


Figure 4.2 Sample Graph1

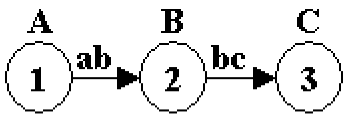


Figure 4.3 Sample Graph2

The corresponding adjacency matrices for Figure 4.2 and Figure 4.3 are shown in Figure 4.4 (matrix1) and Figure 4.5 (matrix2).

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

Figure 4.4 Adjacency Matrix for Figure 4.2

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

Figure 4.5 Adjacency Matrix for Figure 4.3

The adjacency matrices for the two graphs are different even though they have the same number of vertices and same number of edges. According to the MDL principle, the number of bits needed to encode *matrix1* is less than the number of bits required to encode *matrix2*. This is because that in *matrix1* there is only one row in which there are 1s. But in *matrix2*, both the first and second rows have 1s. As a result, two rows have to be represented in the second case whereas only one row has to be represented in the first case. For the input graph of Figure 4.1, if graph compression is performed using the first substructure the resultant MDL value is 1.14539 and if the graph is compressed using the second substructure the MDL value obtained is 1.12849. Therefore, the MDL principle ranks *graph1* higher than *graph2*. The MDL formula is shown below:

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

In the above formula, the less the number of bits needed to represent DL (S) + DL (G|S), the better is the substructure. The DMDL principle strives to take into account this representation difference as the main reason for distinguishing same signature substructures.

4.1.3 DMDL Computation

The DMDL value is calculated using a formula that helps us achieve the goal of differentiating same signature substructures.

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

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

```
Value(G) = graph_vertices + graph_edges
Value(S) = sub_vertices + uniquesub_edges
Value(G|S) = (graph_vertices – sub_vertices * count + count) +
             (graph_edges – sub_edges * count)
```

Value(G) in the above formula represents the value of the entire graph. It is calculated as the sum of vertices and edges in the graph. Value(S) is calculated as the sum of substructure vertices and the unique substructure edges. Value(G|S) represents the compressed graph (replacing all the instances of the substructure in the graph). The parameter uniquesub_edges is calculated as the number of unique extensions in the substructure. This is because extensions are the only way we can determine how vertices are connected within a substructure.

So we need to simulate the effect of discriminating between substructures using an adjacency matrix. The number of rows that contain 1's in the adjacency matrix is taken into account to compute the number of bits required to represent the substructure in MDL. So computing the number of vertices that have a minimum outdegree of 1 corresponds to the number of rows with 1's in the adjacency matrix. In DB-Subdue, the outdegree of a vertex can be deduced only from the extensions of the substructure. Therefore computing the unique number of extensions for a substructure, will give the number of vertices having a minimum outdegree of 1.

For the graph in Figure 4.2 the extensions are 1, 1, as the second vertex is extended from the first vertex and the third vertex is also extended from the first vertex. Therefore the uniquesub_edges value is 1. The value 1 indicates that only one row of 1's are present in the corresponding adjacency matrix. For the graph shown in Figure 4.3, the extensions are 1, 2 as the second vertex is extended from the first vertex and the third vertex is extended from the second vertex. The uniquesub_edges value is 2 indicating that there are two rows of 1's present in the corresponding adjacency matrix. Therefore, without computing the adjacency matrix the same effect is obtained in the DMDL value. But there are some limitations that are explained later in the section. The DMDL value for the graph of Figure 4.2 is 1.1481 and the value for graph of Figure 4.3 is 1.1071. Hence, the first substructure is a better substructure than the second substructure as it has a higher value. In general, for higher edge substructures having the same signature, substructures with vertices having a higher out-degree are better substructures without taking into consideration the outdegree in the compressed graph.

4.1.4 Limitations of DMDL

The DMDL heuristic takes into account only the representation of the input graph and the representation of the substructure. It does not take into account the connection of the substructure to the rest of the input graph, which is also taken into account by the MDL metric. In the MDL principle the adjacency matrix is computed for the compressed graph. This governs the value of DL(G|S) in the MDL formula. But in the DMDL formula, the equivalent is value(G|S), and is computed using a simple formula, which does not take into account the compression that might change if there is more than one edge going out of the same vertex in the compressed graph. Thus the DMDL value is less consistent when compared to the MDL formula.

4.2 Detecting Cycles in Substructures

Detecting cycles is a crucial aspect when dealing with graph-mining algorithms. This section describes the problem not handles by DB-Subdue, its effects on substructure discovery, and the proposed solution.

4.2.1 Problem Definition

The main problem with the DB-Subdue algorithm is the fact that it loops within a cycle after detecting the same. The graph of Figure 4.6 will be used for explanation purposes. The graph has a self-loop, a two-edge cycle and a four-edge cycle.

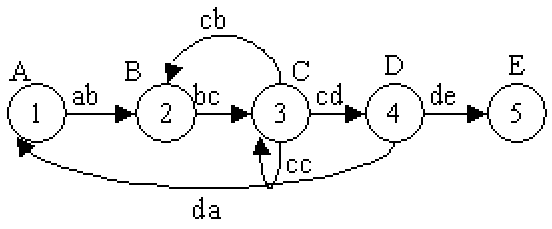
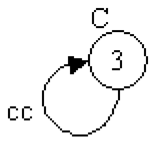


Figure 4.6 Graph with cycles

4.2.2 <u>Simple Cycle</u>

In the graph shown in Figure 4.6 there is a single edge cycle as shown in Figure 4.7



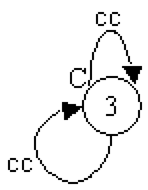


Figure 4.7 Simple Cycle Figure 4.8 Simple Cycle Extension

In DB-Subdue, the substructure in Figure 4.7 will be extended to the substructure shown in Figure 4.8. This is a wrong substructure since the substructure does not exist in the input graph. Therefore this extension must be stopped; otherwise wrong substructures will be reported. Detection of these extended cycles may also affect the best substructure that is discovered. Cycles are detected by checking if the vertex number of any vertex in the new substructure formed is already present in the substructure. The conditions to prevent cycles are formalized as follows:

Given a cycle,

$V_1, V_2 \ldots V_j, V_{j+1} \ldots V_{j+k}, V_j$

In the above subgraph, there is a cycle as the vertex Vj appears twice. We need to eliminate expansion from the second occurrence of $V_{j,}$ as it is the repetition of a vertex already present in the substructure. This is done by making the second occurrence of V_j as $V_j + 0.1$. This prevents expansion from the second occurrence of V_j as its new value is not present in the input graph. The addition of 0.1 is an arbitrary choice; instead any other decimal can be added from 0 to 1. The computation will not be affected in any manner, as the main objective is to substitute the repeated vertex with a vertex not present in the input graph. For example, in Figure 4.7, for the extension from $3 \rightarrow 3$, vertex 3 repeats again and the presence of a cycle can be deduced. As the cycle is now detected, extension from the second occurrence of vertex 3 should be prevented. This is achieved by adding the value 0.1 to the vertex 3. In the example, the new substructure will be $3 \rightarrow 3.1$. This prevents the extension from the second occurrence of vertex 3, as 3.1 will not match any vertex in the input graph, as it is assumed that vertex numbers are always integers in the input graph.

4.2.3 Multiple Cycles

Consider the example of multiple cycles shown in Figure 4.6 and Figure 4.9. In DB-Subdue, the substructure in Figure 4.9 will be extended to form repetitive cycles that are not present in the input graph. In the substructure there are multiple cycles from the same vertex as shown in the substructure $2 \rightarrow 3 \rightarrow 3 \rightarrow 2$. In this substructure both the cycles are formed from the vertex 3. The cycle $2 \rightarrow 3 \rightarrow 2$ is also contained within the cycle $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ to form nested cycles.

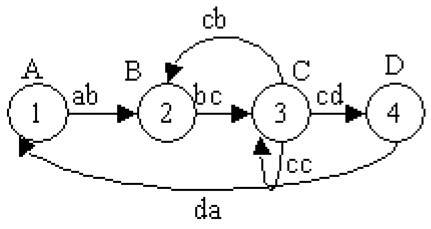


Figure 4.9 Multiple Cycles

Therefore the extension has to be stopped as we saw in a simple cycle. Cycles are again detected by checking if the vertex is already present in the substructure. This is illustrated below:

Given a cycle,

$V_1, V_2 \ldots V_i, V_{i+1} \ldots V_j, V_{j+1} \ldots V_{j+k}, V_j \ldots V_{i+k}, V_i$

In the above cycle, we need to eliminate expansion from the second occurrence of V_i and V_j, as these vertices are already present in the substructure. For example, in Figure 4.9, the extensions are $2 \rightarrow 3 \rightarrow 2$. As the vertex 2 repeats, we detect that there is a cycle. Therefore, a two-edge cycle can be effectively detected. Therefore 0.1 is added to the second occurrence of vertex '2' and the new substructure is $2 \rightarrow 3 \rightarrow 2.1$. This prevents extension from the second occurrence of '2'. Similarly 0.1 is added to the second occurrence of vertex 3 to prevent expansion from that vertex. Thus the same condition is valid for both nested and multiple cycles.

In general, each time a new vertex is added, it is checked to see if that vertex is already present in the substructure. If so, then 0.1 is added to that vertex to prevent future expansions from that vertex. This prevents extension from the last vertex to the cycle that has already been formed.

4.2.4 Limitation of the approach

The main limitation of the approach is that the approach does not handle multiple edges in the same edge direction between two vertices. For example, if there is an edge ‘a’ between vertices ‘A’ and ‘B’ and there is another edge ‘b’ between ‘A’ and ‘B’ then only one of the edges will be taken into account for the two-edge substructure generation containing both the edges.

4.3 Overlap in substructures

This section defines overlap among substructures. It explains the current status of DB-Subdue regarding overlap and the solution to the problem.

4.3.1 Problem Definition

Two or more instances of a substructure are said to overlap if they have a common substructure between them. Figure 4.10 shows an example of an overlap between two instances of a substructure. There are three instances of the substructure AB (vertex labels ‘A’ and ‘B’ and edge label ‘ab’) in the graph. As it can be seen, the vertex 1 is common for two of the three instances. This means that the two instances are said to overlap. The count of the instances of the substructure has to be counted as three if overlap is considered. But in some application domains overlap should not be taken into account. Considering the overlap returns a higher number of instances of the substructure, which tends to inflate their MDL value.

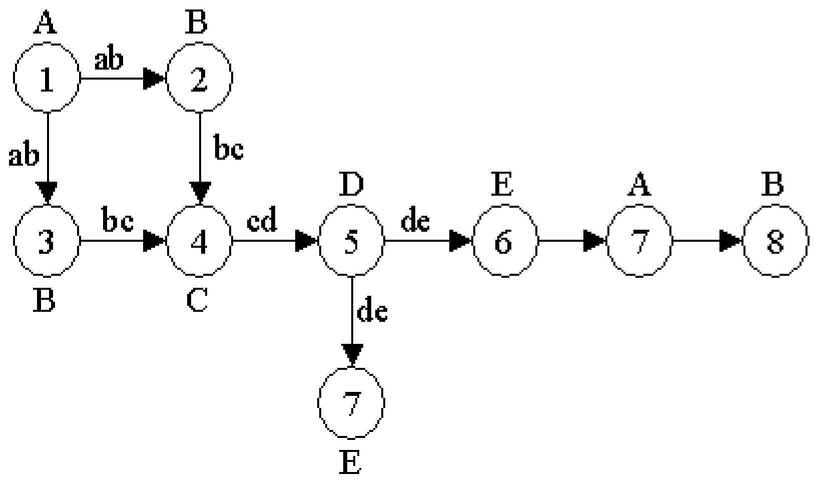


Figure 4.10 Overlap Example

In this thesis we have developed a method by which the overlapping instances of the substructure are counted only once, so that the MDL value can be reduced and the substructures are ranked in proper order.

4.3.2 Single Vertex Overlap

Figure 4.11 shows an example in which there is overlap on one vertex.

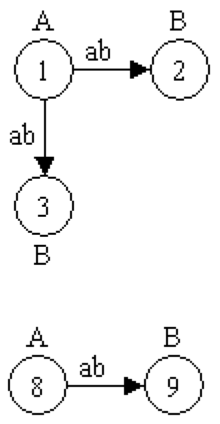


Figure 4.11 Overlap on first vertex

It can be seen that two of the three instances of substructure ‘AB’ overlap. This means that there is an overlap on vertex 1. So we need to count the number of instances of ‘AB’ as two rather than three. This is achieved by checking which overlapping instances of the substructure have a common first vertex and only the overlapping instance that has the greatest second vertex value is included in calculating the count. Taking the greatest vertex value is an arbitrary choice; instead, we can include the instance that has the least vertex value for calculating the count. The result will be the same, because the aim is to reduce the count of the substructure. This will not affect the value of the substructure, as the DMDL value does not take into account how the substructure is connected to the rest of the graph. In the example, the substructure 1 $\rightarrow$ 2 is not accounted for counting as 2 is lesser than three. Therefore the count of the substructure ‘AB’ is two rather than three. But there could be an overlap on the second vertex too, apart from the overlap on the first vertex. This is illustrated in Figure 4.12.

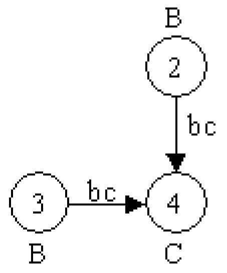


Figure 4.12 Overlap on second vertex

In this case the overlap is on the second vertex. In the example, the substructure 2 $\rightarrow$ 4 is not counted as ‘2’ is less than ‘3’. Therefore the number of instances of ‘BC’ is counted as one rather than two.

4.3.3 Substructure overlap

We will first explain how overlap is avoided in a two-vertex substructure and then generalize how the overlap is avoided for a higher-vertex overlap. An example of a two-edge substructure overlap is shown in Figure 4.13.

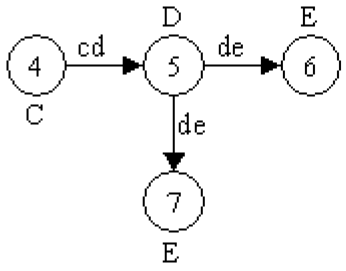


Figure 4.13 Two edge Substructure Overlap

In the above example there are two instances of the substructure ‘CDE’ that are overlapping and the overlapping substructure among them is ‘CD’. Since both the first and second vertex is overlapping, only one of the overlapping instances of the substructure must be considered for counting the number of instances of the substructure ‘CD’. This problem is solved by only accounting one instance; the instance with the greatest third vertex value while the count is computed. In the example, the substructure ‘CDE’ will have count of one rather than two, as there is substructure ‘CD’ overlapping.

In general, for a higher-edge substructure, if there is an overlap on any of the vertices other than the last vertex, then only the overlapping instance that has the greatest last vertex value is included for computing the count. In case of an overlap on the last vertex, each vertex is checked starting from the vertex previous to the last one till the first vertex and the non-overlapping vertex is determined. Only the overlapping instance that has the greatest non-overlapping vertex value is included for calculating the count. Therefore when overlap is avoided then the count of the substructure as computed as follows:

No. of instances of the substructure = No. of non-overlapping instances + one overlapping instance

4.4 Graph Isomorphism

One of the most important parameters for evaluating a substructure is the frequency of occurrence of the given substructure in the input graph. This requires the graph isomorphism test, which indicates if two graphs are identical. In most practical applications substructures that are formed differ by a small margin. Therefore, apart from performing the exact graph match we also need to perform an inexact graph match between two graphs. This section describes the inexact graph match and explains how it is handled as a part of this thesis.

4.4.1 Inexact graph match

The graph isomorphism algorithm is an inexact graph match algorithm, which also subsumes exact graph matches. Given two graphs it outputs the minimum edit distance (cumulative cost of graph changes required to transform the first graph into a graph isomorphic to the second) of the graph pair. The entire range of graph changes is the addition or deletion of a vertex or an edge, changing the direction of an edge, or changing an edge or vertex label. Since there is no known polynomial-time graph isomorphism algorithm Subdue uses an optimal search algorithm instead of an exhaustive search.

The algorithm starts with the ordering of all the vertices of the first graph by its total degree (in-degree + out-degree). It then maps this vertex to each vertex in the second graph and also maps to *null* value (indicating that the vertex should be deleted) and the cost incurred for each mapping is recorded. All the partial mappings generated along with the match costs are maintained in a queue. Each mapping in the queue is removed and the next vertex is mapped to each of remaining vertices in the second graph and the resulting mappings are stored in the queue. This procedure continues until all the vertices of the first graph are mapped to zero or more vertices in the second graph or the user specified threshold value is reached. The best mapping in the queue is the one with the least match cost.

4.4.2 Example of Inexact graph match

The inexact graph match concept explained in the above section is illustrated with an example. In this example all the edge labels are assumed to be the same, although the actual Subdue algorithm considers edge labels.

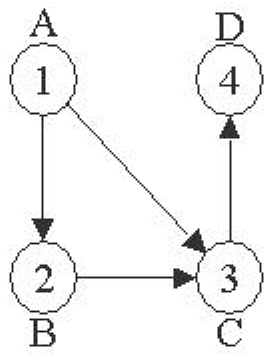


Figure 4.14 Example Graph1

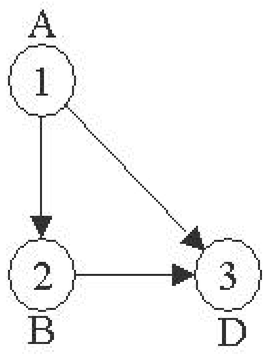


Figure 4.15 Example Graph2

The graphs shown in Figure 4.14 and Figure 4.15 are used as examples to explain the algorithm. There are several mappings of graph1 to graph2. Two of them are shown below:

Mapping 1: $1 \rightarrow 1$, $2 \rightarrow 2$, $4 \rightarrow 3$, $3 \rightarrow$ null

The operations that are performed for transforming the first graph to the second graph are shown below:

1) Delete Edge 1 → 3
2) Delete Edge 2 → 3
3) Delete Edge 3 → 4
4) Delete vertex 3
5) Add Edge 1 → 4
6) Add Edge 2 → 4

Mapping 2: 1 → 1, 2 → 2, 3 → 3, 4 → null

The operations that are performed for transforming the first graph to the second graph using the second mapping is as follows:

1) Delete Edge 3 → 4
2) Delete vertex 4
3) Rename vertex 3 to D

If the cost is assumed to be the same for all the operations that are performed then the second mapping would be the least cost method to match the first and the second graph as the total cost is 3 as compared to a match cost of 6 for the first mapping. But if the cost assigned to each operation is different and renaming a vertex is assigned a cost of 5 rather than 1 then the match cost associated with the second mapping will be 7 and the first mapping would be better as it has a lesser match cost value.

4.4.3 Importance of substructures with unit count

In the DB-Subdue algorithm only exact graph matches were computed. Therefore one-edge substructures that have a count of one will continue to have a count of one when it is extended to a higher-edge substructure. Therefore we could delete them from the input graph, as they would not participate in future extensions. But these substructures that have a count of one can still inexactly match with another substructure and increase their count. This might even affect the best substructure discovered. This is illustrated with an example shown in the Figure 4.16. In the figure, the substructure BD has a count of 1 and the substructure AB has a count of 2. The substructure BD could extend and form a two-edge substructure as shown in Figure 4.17. The substructure still has a count of 1. Now if there is a two-edge substructure as shown in Figure 4.18 with a count of 2, it can inexactly match with the substructure in Figure 4.17 (the two substructures differ only by a vertex label) and its count could be increased to 3.

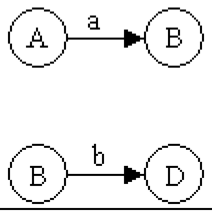
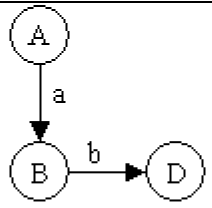
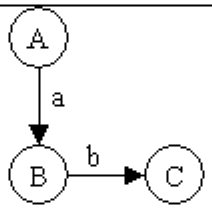
		

Figure 4.16 One edge substructures	Figure 4.17 Two edge substructure with count 1	Figure 4.18 Two edge substructure with count 2

The substructure in Figure 4.17 will not be formed if the substructure ‘BD’ was deleted from the input graph. This would have also made the count of ‘ABC’ remain as 2 rather than 3, as in this case, and thereby restrict the chance of this substructure from becoming the best substructure.

4.4.4 Database Algorithm

The first main difference that has to be made to the DB-Subdue algorithm is that we have to stop deleting substructures that have a count of one from the joined_base table. This is because substructures that have a count of one can inexactly match with another substructure and increase the frequency of that substructure. We then perform an inexact graph match by using cursors to get the tuples corresponding to the two substructures. This inexact graph match is performed on the frequent table that has substructures on which the exact graph match has already been performed and the counts updated. So, essentially without performing an inexact graph match on the whole table, the exact graph match is performed to reduce the number of tuples in the frequent table. The two substructures are then constructed to form strings. The strings are passed to the isomorphism test subroutine. The isomorphism_test subroutine is part of the Subdue algorithm. The match cost is then compared with the formula that is a function of the threshold value and the substructure vertices and edges.

```
Matchcost <= threshold * ((sub_vertices + sub_edges)/ 2)
```

If the matchcost in the above comparison is satisfied then the count of the first substructure is increased by the sum of the counts of the two substructures.

4.4.5 Need for Ordering

This section describes the importance of ordering of the substructures in the database graph isomorphism algorithm. The need for ordering is explained using the example shown in Figure 4.19. The tuples in the current algorithm are arranged in the descending order of the counts of the substructure. As it can be seen from the figure, the tuple representing the substructure ‘ABC’ and the tuple representing the substructure ‘ABD’ is very far apart due to their counts

E	F	G	6
A	B	C	4
D	E	F	3
G	H	I	2
L	M	N	2
A	B	D	1
C	D	E	1

Figure 4.19 Need for ordering

Therefore all the substructures after ‘ABC’ have to be compared to check for an inexact graph match for ‘ABC’. This could be minimized if the substructure ‘ABD’ was brought close to ‘ABC’, so that with fewer comparisons the inexact graph match would be found and the count could be updated accordingly. Therefore we need an algorithm that could bring potential isomorphic substructures close to each other.

4.4.6 Alternate approach

The algorithm for the inexact graph match that was explained in the previous approach is a naïve algorithm. Since most of the substructures were far apart from each other in the table, each tuple had to be inexactly graph-matched with every other tuple in the table. This is an n^2 approach because each tuple is compared with every other tuple in the table. The cursor operations take a long time as cursors are used to fetch each tuple from the table for comparison. An alternate approach is proposed in this section for graph isomorphism. This method is based on the canonical labeling of frequent subgraphs. In this approach a unique canonical label is formed for each substructure during the generation of the substructure. The canonical label is essentially a string of the following form:

```
G,Ctag
```

In the above expression G represents the group in which the substructure is present. Tag represents the substructure vertices and edges in lexicographic order and C represents the cost for transforming each substructure to the lexicographic order. For example, if there is a substructure B $\rightarrow$ A $\rightarrow$ C then the canonical label for this would look like *1,2ABC* where 1 represents that it belongs to the first group as the letter 'A' is assigned the value 1. Each unique label in the graph is assigned a value. In the label, 2 represents the cost for swapping A and B and the remaining terms correspond to all the vertices in the substructure that are arranged in lexicographic order. This ensures that the potential substructures that will match inexactly come close together. But this ordering cannot guarantee correctness in all cases, as the substructure is not compared with all the other substructures for inexactness. The other things that have to be taken care of while forming the canonical label are the edge labels in the substructure and the edge directions. If all the information is properly used while forming the canonical label, then the comparison between substructures in the same group will ensure correctness to a greater extent. Thus, even though the alternate approach reduces the computation time as fewer cursors need be compared, the correctness is not guaranteed as compared to the first approach.

4.5 Summary

This chapter gave an insight into the cycle's problem present in DB-Subdue and how it was solved. A viable solution was also given for evaluating the substructures using DMDL principle. The later part of the chapter discussed the overlap problem in graph mining and the need to solve the problem. The generalization was also mentioned for each problem. The database version of the inexact graph match algorithm was explained and an alternate approach was also proposed for better performance.

CHAPTER 5

IMPLEMENTATION DETAILS

This chapter discusses the implementation details of database minimum description length, how the cycles are handled, overlap related issues and graph isomorphism. It also gives a detailed explanation of how the mapping of substructures to tuples is carried out along with the change of conditions in joins.

5.1 Implementation of Database Minimum description length

This section describes what were the problems and how the database minimum description length was finally implemented. It includes the use of functions in PL/SQL [22].

5.1.1 Function in PL/SQL

5.1.1.1 PL/SQL Function Syntax

A function is a module that returns a value. Unlike a standalone function, which is a standalone executable statement, a call to a function can only be a part of an executable statement. The structure of a function is the same as that of a procedure except that the function also has a return clause. The general format of a function follows:

```
CREATE FUNCTION function-name [ (parameter [, parameter …] ) ]
     RETURN return_datatype
IS
     [declaration statements]
BEGIN
     executable statements
     [ EXCEPTION
          exception handler statements ]
```

```
END [ name ];
```

where each component is used in the following ways:

name

The name of the function comes directly after the keyword FUNCTION.

parameters

An optional list of parameters that we can define to pass information into as well as send information out of the function, back to the calling program.

return_datatype

The datatype of the value returned by the function.

declaration statements

These statements are used to declare local variables used in that function. If there are no local variables then the section is left empty.

executable statements

These statements are executed when the function is called. There must be atleast one executable statement between the BEGIN and END statements.

exception handler statements

If there are no exceptions that are handled explicitly then the EXCEPTION keyword can be left out. This statement is optional.

5.1.1.2 PL/SQL Release 2.1

PL/SQL is a procedural language extension to SQL, so we can issue native calls to SQL statements such as SELECT, INSERT, and UPDATE from within our PL/SQL programs. Until release 2.1 of PL/SQL however, we were not able to place our own PL/SQL functions in a SQL statement. This restriction often resulted in cumbersome SQL statements and redundant implementation of business rules. For example, if the total compensation has to be calculated, then the computation is straightforward:

```
Total Compensation  = salary + bonus
```

The SQL statement would include this formula:

```
SELECT employee_name, salary + bonus
   FROM employee;
```

while in the Oracle forms application (front-end) would employ the following PL/SQL code:

```
:employee.total_comp := :employee.salary + :employee.bonus;
```

In this case, the calculation is very simple, but the fact remains that if we need to change the total compensation formula for any reason (different kinds of bonuses, for example), we would then have to change all of the hard coded calculations both in SQL statements and in the front-end application components. A far better approach is to create a function that returns the total compensation:

```
CREATE FUNCTION total_comp
     (salary_in IN employee.salary%TYPE, bonus_in IN
employee.bonus%TYPE)
RETURN NUMBER
IS
BEGIN
     RETURN salary_in + bonus_in
END;
```

Then the code could be replaced as follows:

```
     SELECT  employee_name, total_comp (salary, bonus)
         FROM employee
```

Until the release of 2.1 of PL/SQL the above SELECT statement raised the following error:

ORA-00919: invalid function

because there was no mechanism for SQL to resolve references to programmer-defined functions stored in the database.

5.1.2 Algorithm

The formula that was explained in the design chapter is calculated at run time for each substructure (represented as tuple) in the table. This is accomplished by writing a PL/SQL function. The pseudo code for the algorithm is as follows:

```
1)   mdlvalue(extensions,  count, graph_vertices, graph_edges,
              sub_vertices)
2)   uniquesub_edges = Count of unique extensions from the
                       list
3)   Value (G) = graph_vertices + graph_edges
4)   Value (S) = sub_vertices + uniquesub_edges
5)   Value (G|S) = (graph_vertices - sub_vertices * count
                    + count) +
                    (graph_edges - sub_edges * count)
6)   DMDL = Value (G) / (Value (S) + Value (G|S))
7)   return DMDL
```

This PL/SQL function is registered in the database. It is then used in the SQL query when the frequent substructures table is generated. The query for obtaining this is as follows:

```
Insert    into frequent_2 (vertex1name, vertex2name,
          vertex3name, edge1, edge2, ext1, ext2, value)
     Select
               vertex1name, vertex2name, vertex3name, edge1,
               edge2, ext1,ext2,
               mdlvalue(ext1,ext2, count(*), graph_vertices,
                         graph_edges, sub_vertices)
     From     joined_2
```

```
Group by vertex1name,vertex2name,vertex3name,
         edge1, edge2
```

As it can be seen from the query the PL/SQL function, which was generated, is used in the query and the value will be calculated for each tuple automatically in a very efficient manner. This PL/SQL function is generated for each substructure. For two-edge substructures the function is called mdlvalue_2. In general for higher-edge substructures mdlvalue_n function is generated.

5.2 Implementation of handling Cycles effectively

This section describes about how the cycles problem is handled at the implementation level. This section will use the same examples shown in CHAPTER 4 and show the tuple representation in the form of tables. It first goes on to show the simple cycle followed by multiple cycles, and then the generalization for a higher-edge cycle.

5.2.1 Simple cycle

In DB-Subdue the tuple corresponding to the substructure in Figure 4.7 is shown in Table 5.1

Table 5.1 Tuple in joined_beam_1 table for Figure 4.7

vertex1	vertex2	edge1	vertex1name	vertex2name
3	3	cc	C	C

The joined_beam_1 table was chosen instead of the joined_1 table because only these tuples will be extended to the next pass. vertex2 is updated in the joined_beam_1 table and is set to a new value if it matches with vertex1. The SQL query for doing this is as follows:

```
exec sql update joined_beam_1
```

```
set    vertex2 = vertex2 + 0.1
where  vertex1 = vertex2;
```

The updated tuple in the joined_beam_1 table is shown in Table 5.2 and joined_base table is shown in Table 5.3.

Table 5.2 Updated tuple in joined_beam_1 table

vertex1	vertex2	edge1	vertex1name	vertex2name
3	3.1	cc	C	C

Table 5.3 Tuples in Joined_base table

vertex1	vertex2	edge1	vertex1name	vertex2name
3	3	cc	C	C
2	3	bc	B	C
3	2	cb	C	B

Since the vertex2 is made decimal in the joined_beam_1 table, it does not match with any tuples in the joined_base table for extending to a two-edge substructure. This prevents extension from vertex2.

5.2.2 Multiple cycles

This section describes the implementation for a multiple cycle substructure and then generalization for a higher-edge cycle. The updated tuple in the joined_beam_6 table corresponding to the substructure in Figure 4.9 is shown in Table 5.4. Since vertices '1', '2' and '3' are already present in the substructure, it represents a cycle; therefore 0.1 is added to all these vertex numbers.

Table 5.4 Updated tuple in Joined_beam_2 table

1	2	3	3.1	2.1	4	1.1	ab	bc	cc	cb	cd	da	A	B	C	C	B	D	A	2	3	3	3	4

Making the vertex decimal prevents extension from the second occurrence of vertex '3', vertex '2' and vertex '1'. This is because decimal vertex numbers do not match with any of the vertices in the joined_base table shown in Table 5.3.

In general for any higher edge cycle the same principle can be applied to prevent cycles from looping within the same substructure. The general query for updating the vertex negative is as follows:

```
exec sql update joined_beam_N
         set    vertexN+1 = vertexN+1 + 0.1
         where vertexN+1 = vertex1 or
               vertexN+1 = vertex2 or
               …
               vertexN+1 = vertexN-1;
```

The addition of 0.1 here is only an arbitrary choice; any number between 0 and 1 can be added to the vertex number, as the objective is to make the vertex number decimal, so that extension is prevented from that vertex. The attributes representing the vertex numbers in the table are made *float* so that vertex numbers can be represented as decimal.

5.3 Implementation to avoid overlap in substructures

This section explains how the overlap is avoided in substructures so that application domains that do not need overlap can specify the usage. It first explains a single vertex overlap followed by a two-vertex overlap and then goes on to explain how the implementation is done for a substructure overlap.

5.3.1 Single Vertex overlap

The joined_1 table corresponding to the substructure in Figure 4.11 is shown in Table 5.5.

Table 5.5 Tuples in joined_1 table

vertex1	vertex2	edge1	vertex1name	vertex2name	overlap
1	2	ab	A	B	0
1	3	ab	A	B	0
7	8	ab	A	B	0

As it can be seen from the table there are two tuples that have the same vertexname, edgename and extension and they also have the same vertex1 value. This signifies overlap and one of the substructures should not be accounted for computing the count. The overlap column has been added to the table to keep track of which substructure has an overlap. The sql query for performing this is as follows:

```
exec sql update joined_1
     set overlap = 1
     where (vertex1,vertex2) in
     (select j1.vertex1, j1.vertex2
      from   joined_1 j1, joined_1 j2
      where  j1.vertex1name = j2.vertex1name and
             j1.vertex2name = j2.vertex2name and
             j1.edge1 = j2.edge1 and
             j1.vertex1 = j2.vertex1 and
             j1.vertex2 < j2.vertex2)
```

The above query will update the overlap column to 1 for all the tuples that have the overlap on the first vertex except the tuple that has the greatest second vertex value. In the example, the query will update the overlap column to one for the first tuple as this tuple satisfies the where condition. The updated tuples in Joined_1 table is shown below:

Table 5.6 Tuples in joined_1 table for graph in Figure 4.11

vertex1	vertex2	edge1	vertex1name	vertex2name	overlap
1	2	ab	A	B	1
1	3	ab	A	B	0
7	8	ab	A	B	0

Therefore the count of the substructure ‘AB’ is two rather than three if the overlap is allowed. During the generation of frequent_1 table a where condition is added to avoid counting the overlapping tuple. The updated query for generating the frequent_1 table is as follows:

```
exec sql
     create table frequent_1(vertex1name, vertex2name, edge1,
                                  count1)
     (select   vertex1name, vertex2name, edge1, count(*)
      from     joined_1
      where    overlap != 1
           group by vertex1name, vertex2name, edge1);
```

For the case where there is overlap on the second vertex rather than the first vertex as shown in the example Figure 4.12. The joined_1 table corresponding to this substructure is shown in Table 5.7

Table 5.7 Tuples in joined_1 table for graph in Figure 4.12

vertex1	vertex2	edge1	vertex1name	vertex2name	overlap
2	4	bc	B	C	0
3	4	bc	B	C	0

In this case, the overlap column of the first tuple in the table is updated to 1. The overlap is detected by adding another condition to the existing where clause during the frequent_1 table generation. The condition is shown below and the updated joined_1 table is shown in Table 5.8.

```
j1.vertex2 = j2.vertex2 and j1.vertex1 < j2.vertex1
```

Table 5.8 Tuples in joined_1 table for graph in Figure 4.12

vertex1	vertex2	edge1	vertex1name	vertex2name	overlap
2	4	bc	B	C	1
3	4	bc	B	C	0

Therefore only the overlapping instance that has the greatest first vertex value is included for calculating the count. In the example, the count of the substructure ‘BC’ is one rather than two if the overlap is not allowed on substructures.

5.3.2 Substructure overlap

For the two-edge substructure shown in Figure 4.13, the tuples in the joined_2 table is shown in Table 5.9.

Table 5.9 Tuples in joined_2 table

vertex1	vertex2	vertex3	edge1	edge2	vertex1name	vertex2name	vertex3name	ext1	overlap
4	5	6	cd	de	C	D	E	2	0
4	5	7	cd	de	C	D	E	2	0

The query for updating the overlap column for joined_2 table is shown below:

```
exec sql update joined_2
     set overlap = 1
     where (vertex1,vertex2,vertex3) in
     (select j1.vertex1, j1.vertex2, j1.vertex3
      from   joined_2 j1, joined_2 j2
      where  j1.vertex1name = j2.vertex1name and
```

```
j1.vertex2name = j2.vertex2name and
j1.vertex3name = j2.vertex3name and
j1.edge1 = j2.edge1 and j1.edge2 = j2.edge2 and
j1.ext1 = j2.ext1 and
(
(j1.vertex1 = j2.vertex1 and j1.vertex3 <
j2.vertex3) or (j1.vertex2 = j2.vertex2 and
j1.vertex3 <  j2.vertex3) or (j1.vertex3 =
j2.vertex3 and ((j1.vertex2 < j2.vertex2) or
(j1.vertex2 = j2.vertex2 and j1.vertex1 <
j2.vertex2))
);
```

As it can be seen from the query, if there is overlap on any vertices other than the last vertex, then the overlap column is updated to 1 for all the overlapping instances except the overlapping instance that has the greatest last vertex value. If there is an overlap on the last vertex then all the remaining vertices are checked starting from the vertex previous to the last vertex to the first vertex until the vertex in which the less than condition is satisfied (non-overlapping vertex). In the table shown for joined_2 table, two instances of the substructure CDE overlap on vertex1 and vertex2. The overlap column of all the overlapping tuples except the tuple that has the greatest vertex3 value is updated to one. In the example, the instance '456' is updated to 1. The updated tuples in the joined_2 table is shown in Table 5.10.

Table 5.10 Updated tuples in the joined_2 table

vertex1	vertex2	vertex3	edge1	edge2	vertex1name	vertex2name	vertex3name	ext1	overlap
4	5	6	cd	de	C	D	E	2	1
4	5	7	cd	de	C	D	E	2	0

Thus the count for the substructure ‘CDE’ is updated to one rather than two as one of the substructures is overlapping. In general for N edge substructures, the SQL query for detecting overlapping instances of a substructure are as follows:

```
exec sql update joined_N
     set overlap = 1
     where (vertex1,vertex2,vertex3,…,vertexN) in
     (select j1.vertex1, j1.vertex2, j1.vertex3,…j1.vertexN
      from   joined_N j1, joined_N j2
      where  j1.vertex1name = j2.vertex1name and
             j1.vertex2name = j2.vertex2name and
             j1.vertex3name = j2.vertex3name and
             …
             j1.vertexN+1name = j2.vertexN+1name and
             j1.edge1 = j2.edge1 and j1.edge2 = j2.edge2 and
             …
             j1.edgeN = j2.edgeN and
             j1.ext1 = j2.ext1 and
             …
             j1.extN-1 = j2.extN-1
             (
             (j1.vertex1 = j2.vertex1 and j1.vertexN+1 <
             j2.vertexN+1) or (j1.vertex2 = j2.vertex2 and
             j1.vertexN+1 <  j2.vertexN+1) or
             …
             (j1.vertexN = j2.vertexN and j1.vertexN+1 <
             j2.vertexN+1) or
             (j1.vertexN+1 = j2.vertexN+1 and
             ((j1.vertexN < j2.vertexN) or
             (j1.vertexN = j2.vertexN and
             j1.vertexN-1 < j2.vertexN-1) or
             …
             (j1.vertexN = j2.vertexN and j1.vertexN-1 =
             j2.vertexN-1 ….j1.vertex2 = j2.vertex2 and
             j1.vertex1 < j2.vertex1)
             );
```

5.4 Implementation of Inexact graph match

This section deals with the implementation of the inexact graph match algorithm. The main challenge was to find instances of substructure that inexactly graph match with another substructure.

5.4.1 Cursor Declarations

A cursor is like a name associated with an SQL query. A *cursor declaration* is used to declare the name of the cursor and to specify its associated query. Three statements OPEN, FETCH and CLOSE operate on cursors. An OPEN statement prepares the cursor for retrieval of the first row in the result set. A FETCH statement retrieves one row of the result set into some designated variables in the host program. After each fetch, the cursor is positioned on the row of the result set that was just fetched. FETCH statement is usually executed repeatedly until all the rows of the result are fetched. A CLOSE statement releases all the resources used by the cursor when it is no longer needed. In addition to their use in retrieving query results into variables in the host program, cursors can play a role in updating (including deleting) rows of data in the database. A special form of the UPDATE statement called the *positioned* update statement can be used to update exactly one row in the database based on the position of the cursor. In a positioned update, instead of a search condition, the *where* clause contains the phrase *current of* followed by a cursor name. The DELETE operation works in a similar way.

The syntax of a cursor declaration is shown below [22]

```
DECLARE--cursor-name—CURSOR {WITH HOLD}
```

```
      {WITH RETURN TO CLIENT/TO CALLER}
FOR statement-name
```

The following example shows a series of statements for using a cursor.

```
EXEC      SQL
DECLARE   c1 CURSOR FOR
          SELECT    vertexname,vertexno
          FROM      edges
          FOR       DELETE ;

EXEC SQL  OPEN c1;
EXEC SQL  FETCH c1 into :vertexname,:vertexno;
     If(vertexno>10)
          EXEC SQL
               DELETE FROM edges
               WHERE CURRENT OF c1;
```

5.4.2 Isomorphic Comparison

In this part of the chapter we explain how the tuples in the table are extracted and used for performing the inexact graph match. The only way of retrieving tuples into program variables is by using cursors. In this algorithm two cursors are declared on the frequent table. The frequent table contains all the substructures after the exact graph match is done. This ensures that we are performing an inexact graph match on a smaller table when compared to the original table having both exact and inexact graph matches. Two tuples are retrieved at a time from the table using the cursors. The cursor declaration is as follows:

```
EXEC SQL DECLARE frequent_1_cursor CURSOR FOR
     SELECT   vertex1name, vertex2name, edge1, count1
     FROM     frequent_1
     ORDER BY count1 DESC;
```

In the above declaration, *order by* with *desc* option is used so that all the tuples that have greater count is placed at the top of the table. Then the tuples are converted into strings using the same format as required by the inexact graph match algorithm of Subdue. These tuples are then passed to the inexact graph match algorithm and the match cost returned is compared with the value computed using threshold and the substructure vertices and edges as explained in CHAPTER 4. If the match cost is less then the count of the tuple in the table is updated. Let the program variable for capturing the count for the first tuple be frequent_1_count and the second tuple be frequent_2_count. If the first tuple inexactly matches with the second tuple then the count of the second tuple will be updated as follows:

```
frequent_1_count = frequent_1_count + frequent_2_count
```

This value is then used to update the count value of the tuple in the table. The query for doing this is as follows:

```
UPDATE frequent_1
SET count1 = :frequent_1_count
WHERE vertex1name = :frequent_1_vertex1name and
      vertex2name = :frequent_2_vertex1name and
      edge1 = :frequent_1_edge1
```

In the above query the frequent_1_count is a program variable so a ':' is placed before the variable to differentiate from a normal attribute of a table. The frequent_1_vertex1name, frequent_1_vertex2name and frequent_1_edge1 are all program variables that hold the value of the tuple retrieved. For higher-edge substructures an extension program variable is also declared to capture the extensions of the substructure.

These program variables are then used in the update clause to update the corresponding tuple’s count value.

In general for N edge substructures an array of host program variables are used. The array declarations were not possible in DB2 database but are possible in the Oracle database.

5.4.3 Alternate Approach

The approach discussed above requires n^2 computation. As it was explained in the design chapter, the main aim of the canonical labeling was to bring potential substructures close to each other and thereby reduce the number of comparisons made. The value of the canonical labeling is generated when a substructure of size k is extended to a substructure of size k+1 for each k. An example of the string is shown below:

```
1,2ABC
```

In the above example, the first 1 represents the starting of the group. In this example ‘A’ is mapped to 1. ‘2’ represents the cost to transform ‘BAC’ to ‘ABC’ and the list of vertices in the substructure are appended finally to form the string.

A PL/SQL function was generated for each pass to compute the canonical label. All the unique vertex labels are taken from the graph and each is assigned a number. This is done in a two-step process. The query for doing this is as follows:

```
Insert into tempuniquevertices(uniquevertexlabel) (
     select distinct(vertexlabel)
     from vertices
)


Insert into uniquevertices(uniquevertexlabel) (
     select uniquevertexlabel, rownum
     from tempuniquevertices
)
```

This two-step process is used because this is the easiest way to assign numbers for all the unique vertex labels that were retrieved. The uniquevertices table is constructed before the start of the substructure generation process. While constructing the canonical string, to find which group the substructure belongs to, we need to check the first vertex label in the string of vertices with the uniquevertices table. The query for doing this is as follows:

```
Select vertexno into uniquevertex
from uniquevertices
where vertexlabel = sortarray(1)
```

In the above query uniquevertex is a program variable in PL/SQL and sortarray is an array in PL/SQL. In PL/SQL the array starts with index 1. This query will fetch the number corresponding to the first vertex label in the string which is indicated by sortarray(1). The vertexno that is obtained through this query is appended as the first character in the canonical string. The canonical PL/SQL function will be called using the following query:

```
Insert into joined_N (
     select vertexnames,edgenames,extensions,
     canonical_value_N(vertexnames)
     from joined_N-1, Joined_base
     where extension conditions
)
```

In the above query canonical function is called which will construct the canonical string as the N edge substructure is constructed. While doing cursor operations for finding inexact graph matches, instead of ordering by the count, the tuples are ordered by the canonical value. Therefore, the substructures that are potentially close enough by vertices come close to each other. While checking for comparison between tuples, the inexact graph match is applied until the tuple matches with a tuple in another group. The count of one substructure is increased if it matches inexactly with another tuple in the table.

5.5 Summary

This chapter addressed issues that were discussed in the design chapter. It gave SQL queries to achieve the solution that was explained conceptually in CHAPTER 4 and also showed how the resultant tables were updated in the database. The generalization was also given for each issue that was explained in this thesis.

CHAPTER 6

PERFORMANCE EVALUATION

This chapter gives an overview of all the scalability tests performed after incorporating all the additional functionality that were added to the DB-Subdue algorithm. It assesses the run time for the Subdue algorithm, the DB-Subdue algorithm and the Enhanced DB-Subdue algorithm (after additions).

6.1 Configuration File

A configuration file is useful for automating the process of performance evaluation. It consists of a number of parameters, which once specified correctly, can be used for running the algorithm in an unattended mode. It can also be run for various datasets with different configuration parameters. The different variables in the configuration file are:

```
DBMS Type$User Name$Password$Table Name$MaxSize$Beam
$Threshold$LogFile$Overlap$Cycles$Heuristic$Inexactapproach$Debu
g $Log Results to file
```

RDBMS Name: The RDBMS name (Oracle or DB2) where the input relation is present

UserId: The user who has access over the input relation

Password: The password associated with the UserId – needed to connect to the database

Table Name: The name of the input relation

Max Size: An integer to specify maximum size of substructure to be discovered

Beam: An integer to specify the beam

Threshold: An integer that determines the inexactness between two substructures. The value is between 0 and 1.

Log file name: The name of the log file into which the results would be written

Overlap: DB-Subdue will allow overlap among substructures. Specifying this argument as 1 prevents overlap

Cycles: Specifying this argument as 1 allows cycles to be detected and handled in the graph

Heuristic: Can take a value of 1 or 2 depending upon the frequency heuristic (count) or the DMDL heuristic.

Inexact approach: Take a value of 0 or 1 depending upon the naïve approach or the canonical label approach for inexact graph match

Debug: If true, then prints the debug statements

Log results to file: If the user wants the time to be logged this parameter is set to 1

For each experiment, the values of all these variables are written in a single line in the order of the variables shown above and are separated by a "$" sign. Thus if the configuration file contains several such lines, the algorithms will be invoked that many times. To skip a line, the line should start with the word "REM". Below is an example of some mining configurations.

REM Experiment on Oracle

Oracle$graphmining$graphmining$T5KV10KE$5$10$0.1$T5KV10KES4A4B10.txt

$1$1$2$0$false$1

Here the first line is ignored as it starts from the word "REM". For second line values are used as follows:

RDBMS to use: Oracle

UserID: subdue

Password: subdue

Input Table: T5KV10KE.

Max Size: 5

Beam: 10

Threshold: 0.1

Log file Name: T5KV10KES4A4B10.txt

Overlap: 1 (avoid overlap)

Cycles: 1 (handle cycles)

Heuristic: 2 (DMDL heuristic)

Inexactapproach: 0 (Naïve approach)

Debug: False (don't print debug statements)

Log results to file: 1 (create the log file with timings).

6.2 Writing Log File

Graph mining is a time-consuming process and at times it happens that for certain mining configurations, mining a given dataset may take several hours. Since we have to compare the performances of these approaches with others, after a given time limit, if the approach does not complete, the discovery process has to be killed. Also for the purpose of studying these algorithms, we need to know about their progress while running a data set. Hence it is very important to note the time at each step of the algorithm and produce a log file containing enough information. This log file can then be processed to generate the useful information such as the number of passes completed, time taken for each pass. For this purpose, we generate a log file. The log is written after finishing the algorithm on a data set. This log contains all the individual timings for the SQL queries and the final time taken. Below is a sample content of these logging files.

```
Size1  Size2  Size3  Size4  Total
0.990  1.200  2.230  2.650  7.070
```

The log file contains the individual times taken for processing substructures of each size.

6.3 Graph Generator

This section explains the graph generator used for generating the datasets. These datasets were used for the performance evaluation with appropriate parameters. The graph generator accepts many parameters for constructing the graph. Some of them are listed below:

- Graph output filename
- Number of vertices in the graph
- Number of edges in the graph
- Number of vertex labels
- Number of 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., where # in “v#” is less than the number of vertex labels already entered
  - ✓ Number of edges

- ✓ For each substructure edge
  - ➢ The edge label. It must of the form “e0”, “e1”, etc., where # in “e#” is less than the number of edge labels already entered.
  - ➢ 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 – 1)

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

```
T20V30E.g
20
30
3
2
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 three distinct vertex labels (v0, v1, and v2). There are two distinct edge labels (e0 and e1). 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 dataset is labeled as T20V30E, which means that the graph generated from the graph generator has 20 vertices and 30 edges.

6.4 Experimental Results

6.4.1 Dataset without Cycles

This section gives the performance comparison between the Subdue main memory algorithm, the DB-Subdue algorithm and the Enhanced DB-Subdue algorithm. Experiments have been performed for different datasets. These experiments were performed on datasets in which cycles are not present and overlap is present and the overlap will be detected as far as DB-Subdue is concerned, as it is the default behavior. The substructures that are embedded are shown in Figure 6.1 and Figure 6.2.

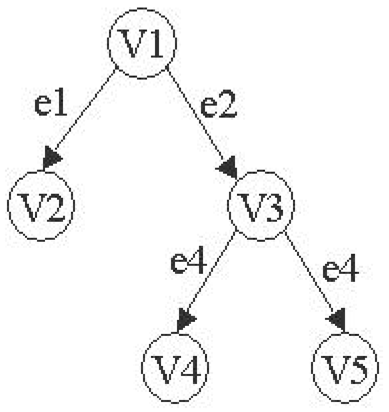


Figure 6.1 Substructure1 with no cycles

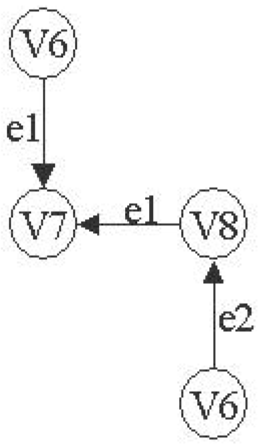


Figure 6.2 Substructure2 with no cycles

The graphs chosen do not have any cycles so that a comparison can also be made with the DB-Subdue algorithm. The parameter settings for the experiments are shown in Table 6.1.

Table 6.1 Parameter Settings

Parameters	Main memory	DB-Subdue	Enhanced DB-Subdue
Size	4	4	4
Beam	4,10	4,10	4,10
Threshold	0	0	0

Apart from these settings, cycles and overlap and MDL are set to true through the configuration file for the enhanced DB-Subdue so that the extra time taken for performing all the operations can be measured. The run-times for all the algorithms are shown in Figure 6.3 and Figure 6.4. They provide a graphical comparison between all the three algorithms that were discussed in this thesis. Datasets are represented along the x-axis and the logarithm of time is represented along the y-axis. As it can be seen from the comparisons, the main memory algorithm was better for the dataset T50V100E. But as the size of the dataset increased, the time taken by the main memory algorithm increased substantially. The crossover point was as low as

250 vertices and 500 edges for both DB-Subdue and Enhanced DB-Subdue in comparison with main memory subdue.

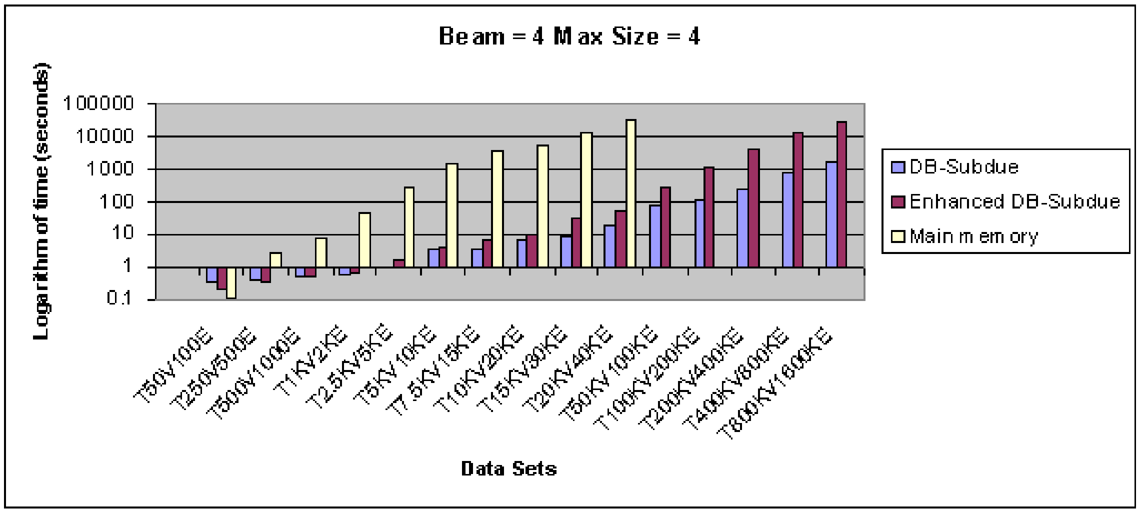


Figure 6.3 Graphical comparison of the approaches

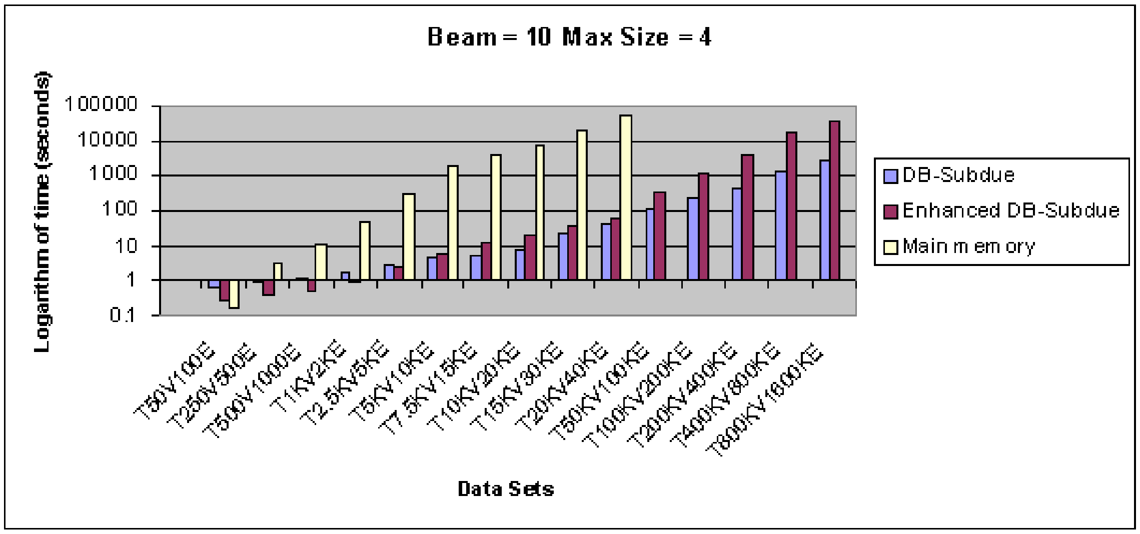


Figure 6.4 Graphical comparison of the approaches

The comparison was the same for a beam value of 4 as well as a beam value of 10. The Enhanced DB-Subdue algorithm had a significant increase in time in comparison to the DB-Subdue algorithm. This can be seen clearly through the individual timings for each operation shown for a mid-range dataset T15KV30KE and a high-range dataset T400KV800KE dataset in Figure 6.5 and Figure 6.6 respectively.

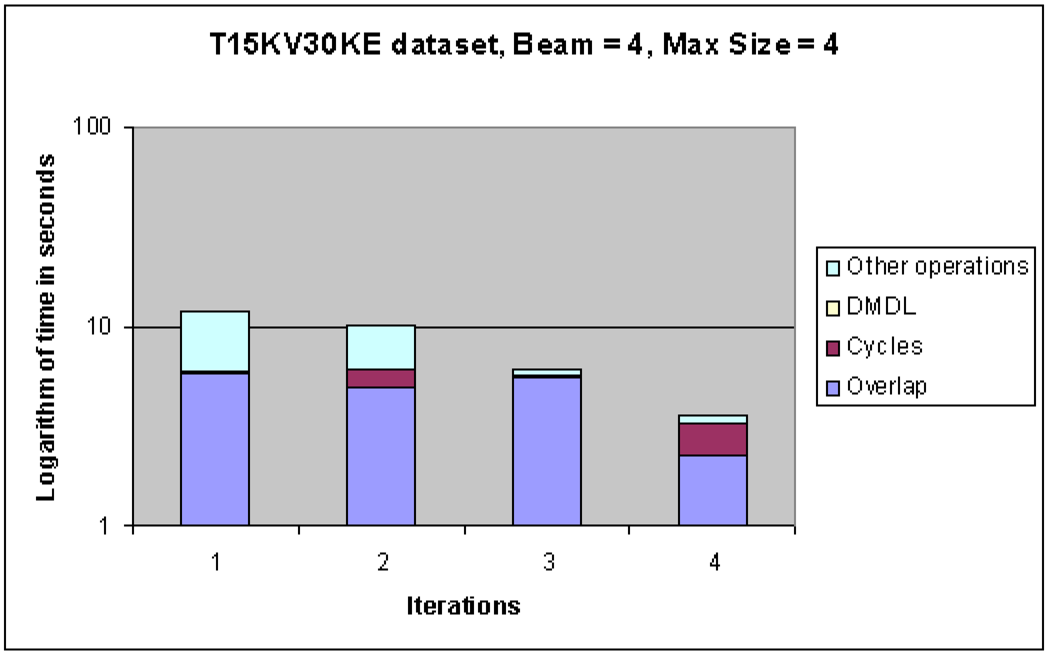


Figure 6.5 Individual timings for T15KV30KE dataset

The increase in run-time is mainly due to the update operation used for overlap detection as shown in Figure 6.5 and Figure 6.6 compared to all the other operations. Update operation is one of the costly operations in database. Since for high datasets, the frequent table size is large, the update operation takes a long time. Another reason for the increase in time is due to the use of OR clauses in the where condition. In SQL, OR clauses are expensive when compared to the AND clauses. This is one of the operations that need to be optimized in the future. Though the increase in time was significant for large datasets the time taken was still low when compared to the main memory counterpart. The graphs show that as the number of iterations increases the time taken by overlap decreases, the reason for this is because the number of tuples in the table decreases as the substructure size increases (fewer instances of the substructures).

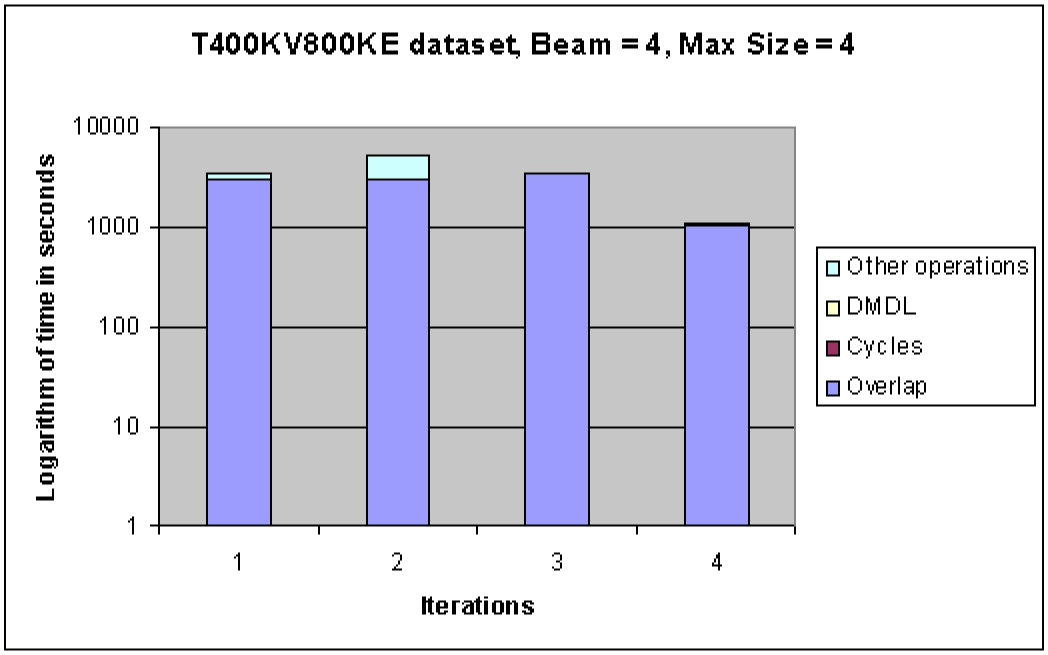


Figure 6.6 Individual timings for T400KV800KE dataset

6.4.2 <u>Dataset with Cycles</u>

This section illustrates the performance comparison for Enhanced DB-Subdue and Subdue main memory. The datasets used in the previous section did not have any cycles. These graphs were chosen so that the comparison can be made between Enhanced DB-Subdue and main memory. The substructures that were embedded are shown in Figure 6.7, Figure 6.8, Figure 6.9 and Figure 6.10. These graphs were generated using the same graph generator that was used to generate graphs in Figure 6.1 and Figure 6.2.

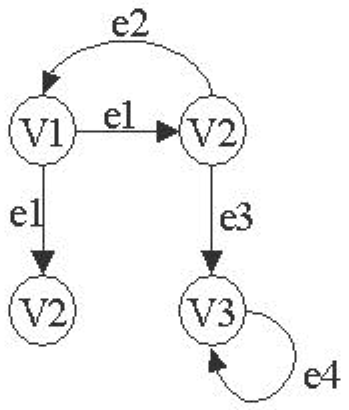


Figure 6.7 Substructure1 with cycles

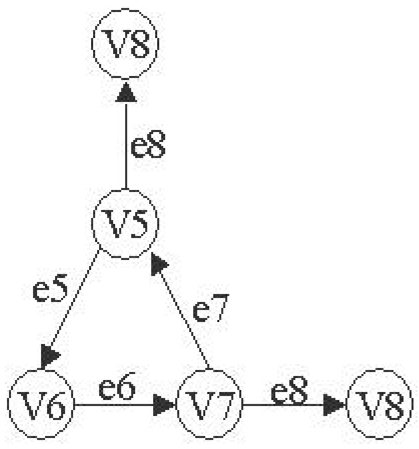


Figure 6.8 Substructure2 with cycles

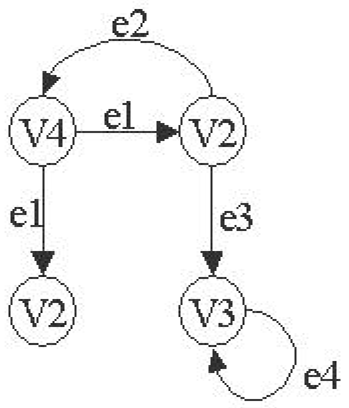


Figure 6.9 Substructure3 with cycles

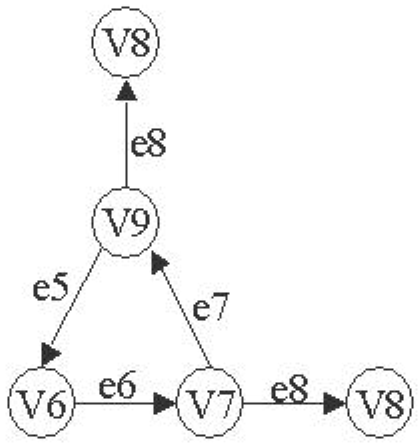


Figure 6.10 Substructure4 with cycles

The parameter settings for both the Subdue main memory and Enhanced DB-Subdue are shown in Table 6.2.

Table 6.2 Parameter settings for graphs with cycles

Parameters	Main memory	Enhanced DB-Subdue
Size	5	5
Beam	4,7,10	4,7,10
Overlap	False	False
Threshold	0	0

Threshold value is set to zero in these performance experiments, so that only exact graph matches are considered. The run-times are shown for the different parameters and for a beam value of 4, 7 and 10 in Figure 6.11, Figure 6.12 and Figure 6.13 respectively.

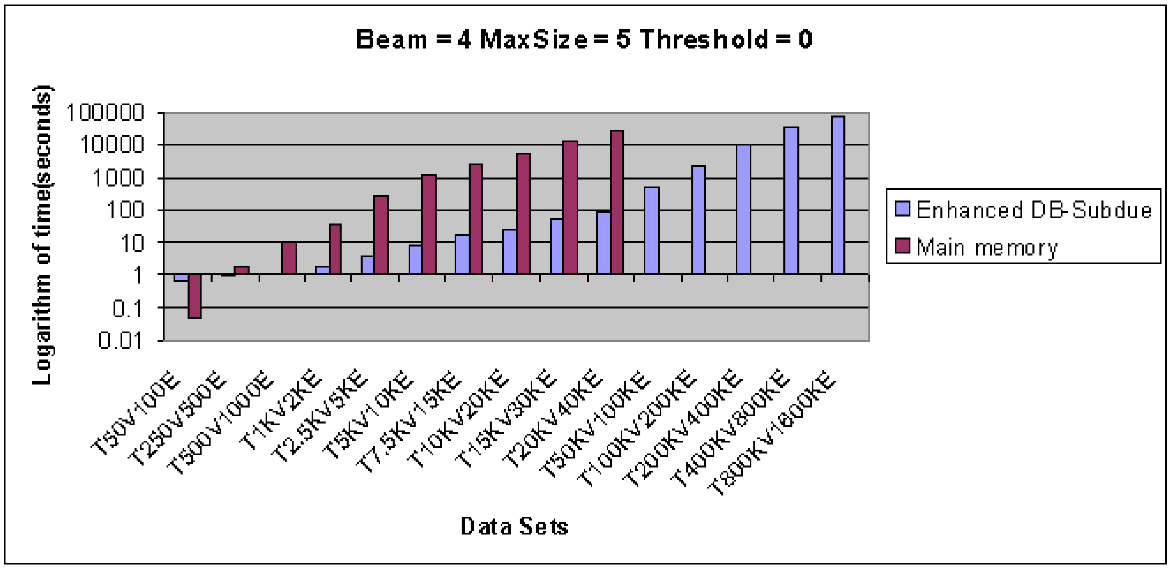


Figure 6.11 Graphical comparison of the approaches

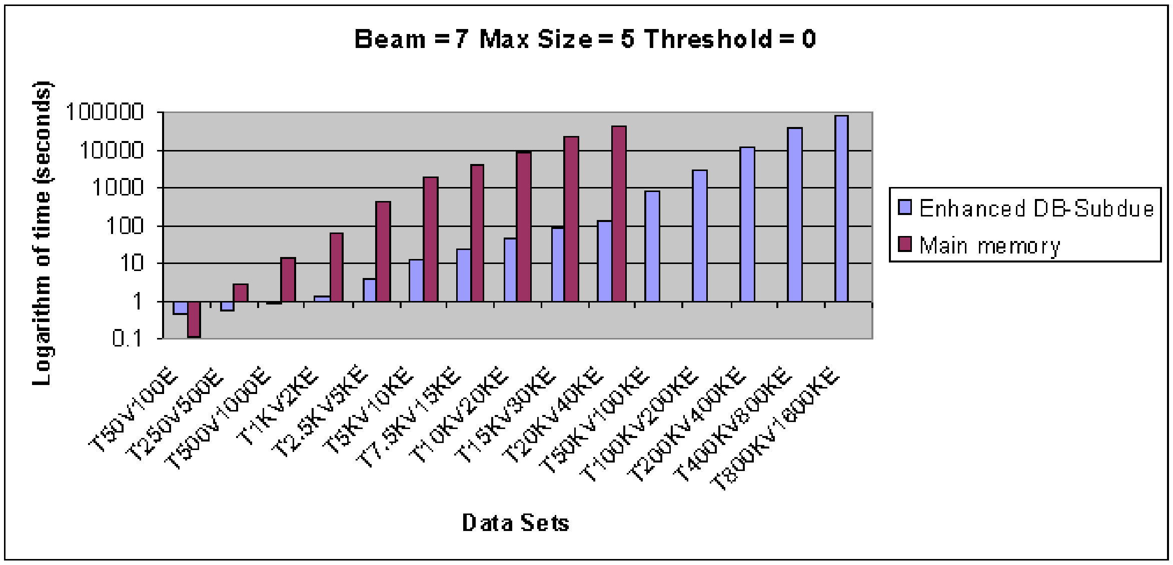


Figure 6.12 Graphical comparison of the approaches

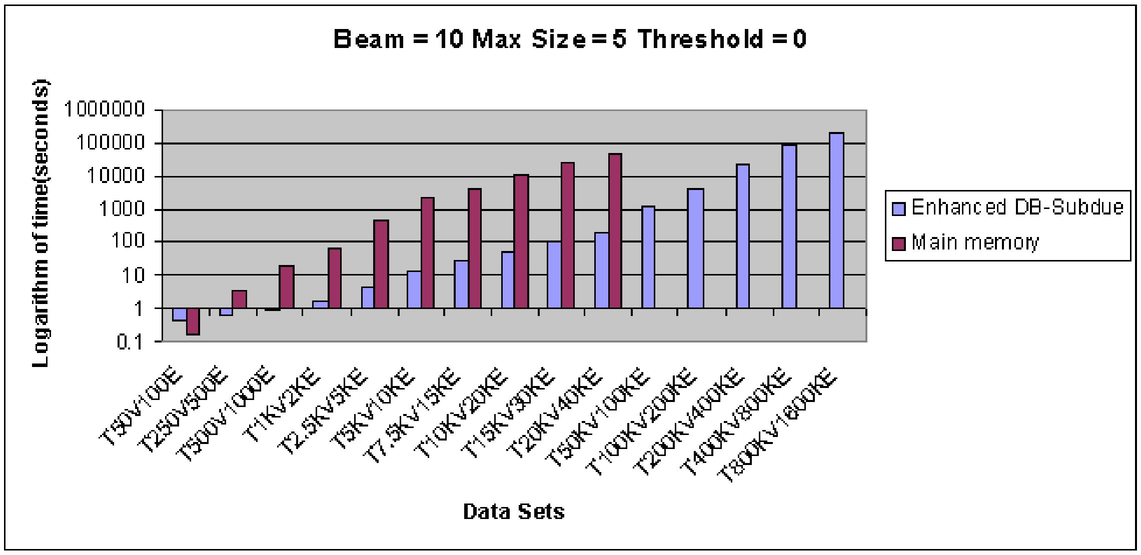


Figure 6.13 Graphical comparison of the approaches

The enhanced DB-Subdue shown in Figure 6.11, Figure 6.12 and Figure 6.13 includes all the additional functionality (DMDL, Cycles and overlap) except the inexact graph match comparison. As it can be seen from the comparisons, although the Enhanced DB-Subdue had a significant increase in run-time when compared to the DB-Subdue algorithm, the performance was still comparable to the main memory algorithm for large datasets.

6.4.3 Dataset for Inexact graph match

We have compared graphs that have cycles and overlap. But the inexact graph match was not taken into account in those comparisons. In this section we will compare the main memory Subdue with the EnhancedDB-Subdue (EDB) that includes inexact graph match. In 6.4.2, the third and fourth substructures that are shown in Figure 6.9 and Figure 6.10 were embedded so that the substructures in Figure 6.7 and Figure 6.8 can inexactly graph match with them and their count would be increased. The comparisons also include both the approaches for Enhanced DB-Subdue. The parameter settings for these experiments are the same except that the threshold is increased to allow inexactness between the graphs. The new parameter settings are shown in Table 6.3. The run-times for the experiments are shown in Figure 6.14, Figure 6.15 and Figure 6.16.

Table 6.3 Parameter settings for inexact graph match

Parameters	Main memory	EDB Naïve Approach	EDB Canonical Approach
Size	5	5	5
Beam	4	4	4
Overlap	False	False	False
Threshold	0.1,0.2,0.3	0.1,0.2,0.3	0.1,0.2,0.3

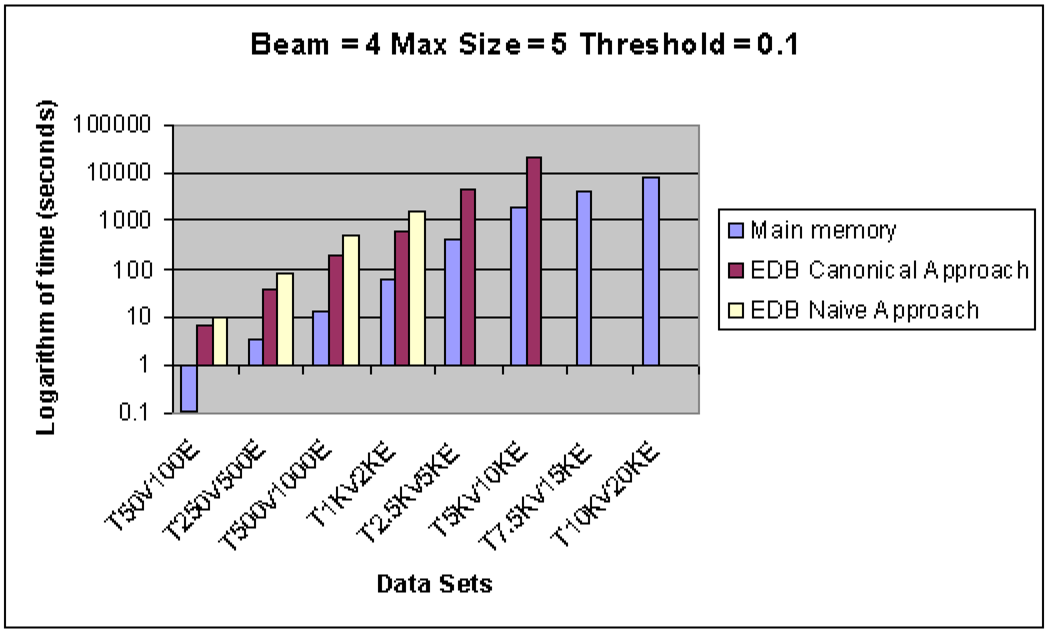


Figure 6.14 Graphical comparison including Inexact graph match for threshold = 0.1

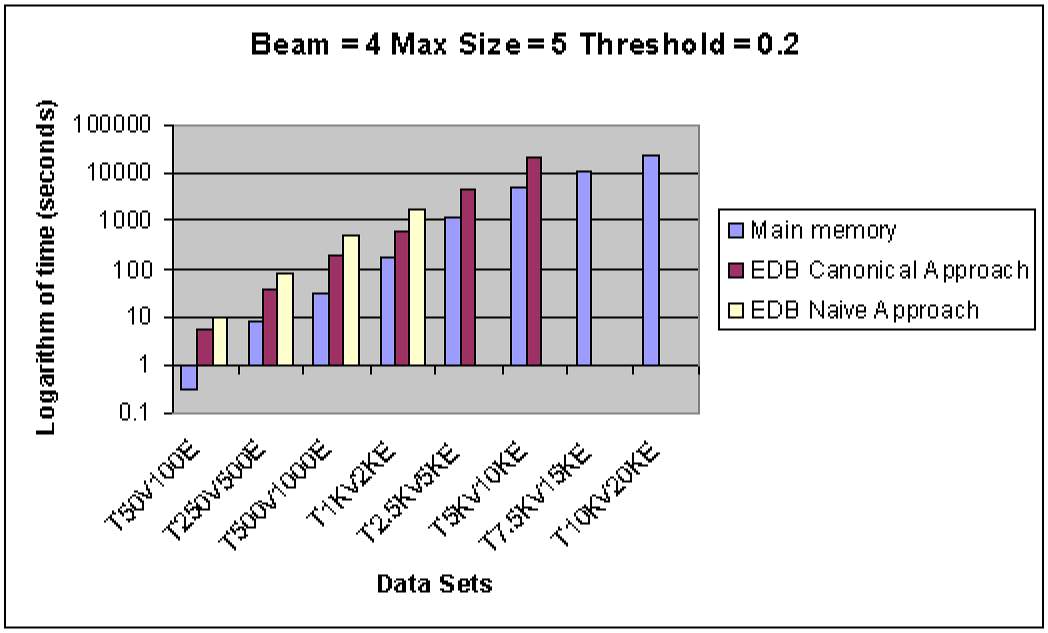


Figure 6.15 Graphical comparison including inexact graph match for threshold = 0.2

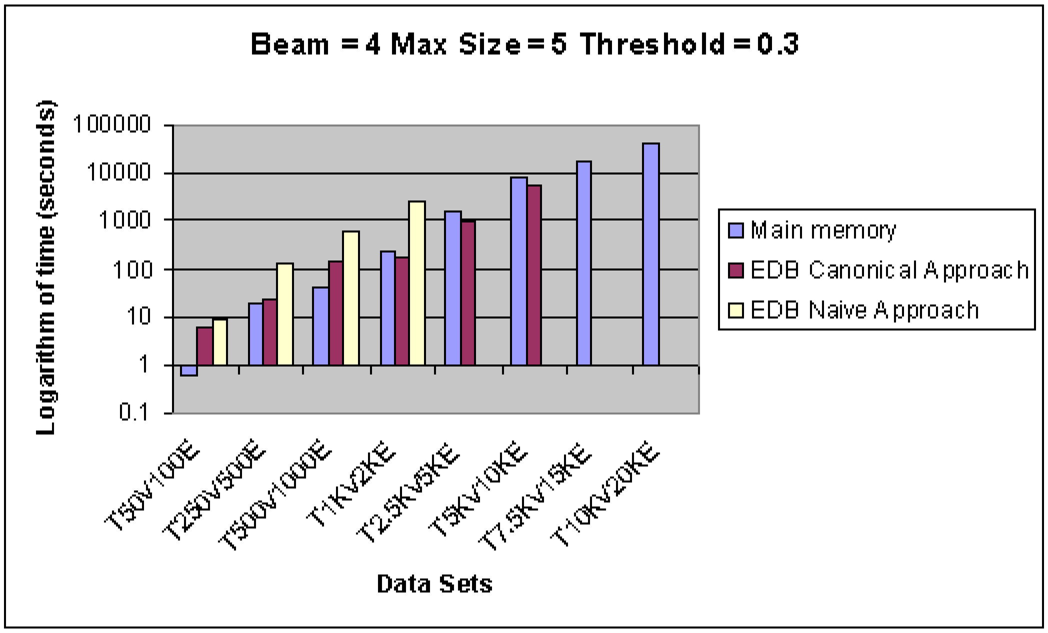


Figure 6.16 Graphical comparison including inexact graph match for threshold = 0.3

From the graphs it can be seen that after including the inexact graph match in Enhanced DB-Subdue, the main memory version performed better than both the approaches of enhanced DB-Subdue. The main reason was that for the naïve approach n^2 comparisons were made and also cursor operations are used which were very costly. In the canonical approach the number of comparisons were reduced but still the cursor operations were very costly. But as it can be seen the canonical labeling approach performed better than the naïve approach. This was mainly because the potential substructures were brought close to each other and comparisons were made only within the same group. For a threshold of 0.3 as shown in Figure 6.16, for a dataset of T5KV10KE, the time taken by the canonical approach was slightly less than that of the main memory algorithm. The reason for this is that the number of tuples in the table decreases as the threshold goes up and therefore the number of comparisons made also decreases to a great extent. The Enhanced DB-Subdue algorithm ran into memory problems and therefore scalability tests could not be done beyond T5KV10KE dataset. But canonical labeling approach is less accurate than the naive approach and the main memory approach as the inexact graph matches can still be across groups. The results obtained could be different for different threshold values. This is because as the threshold increases, more number of substructures inexactly graph-match and a substructure with smaller count for lower threshold value can have a higher count for higher threshold values.

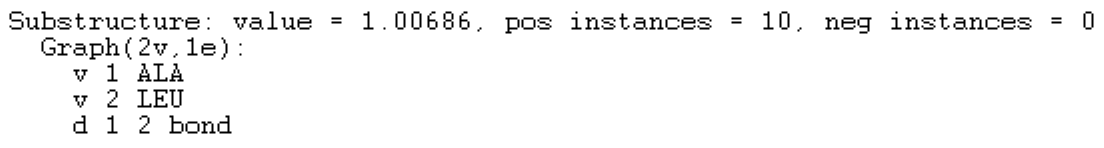
6.4.4 Real-world dataset

Experiments were also performed on a real-world dataset. The real-world dataset chosen was Protein Data Bank (PDB). The PDB is the single worldwide repository for the processing and distribution of 3-D structure data of large molecules of proteins and nucleic acids. Tests were performed on protein datasets: hemoglobin, myoglobin, ribonuclease A, and global. Programs written by the Subdue group were used to convert raw PDB files to graph format. Two of the graph files generated were: one for representing the sequences of all the proteins (seq_graph), and another for representing the secondary structure of all the proteins (secondary_graph). This section gives the output given by Subdue main memory algorithm and the Enhanced DB-Subdue algorithm. The parameters for these experiments are shown in Table 6.4.

Table 6.4 Parameter settings for real-time dataset

Parameters	Main memory	Enhanced DB-Subdue
Size	5	5
Beam	4	4
Overlap	False	False

For seq_graph , the output showing the best substructure for main memory algorithm is shown in Figure 6.17 and the output for Enhanced DB-Subdue is shown in Figure 6.18. The run-time taken by the main memory algorithm was 0.39 seconds and the run-time taken by Enhanced DB-Subdue was 0.64 seconds. The MDL and DMDL value returned by main memory and Enhanced DB-Subdue was not the same as the heuristic used for evaluating the substruc tures are different.



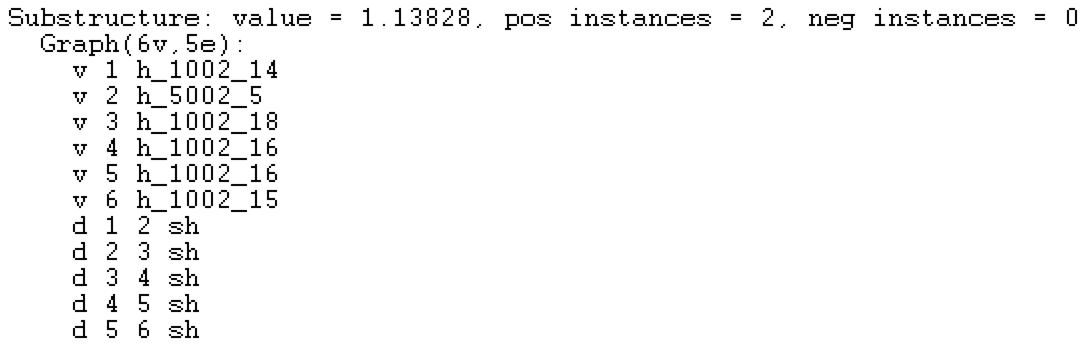
```
Substructure: value = 1.00686, pos instances = 10, neg instances = 0
  Graph(2v,1e):
    v 1 ALA
    v 2 LEU
    d 1 2 bond
```

Figure 6.17 Main memory output for seq_graph

v1	v2	e1	C1	Value
ALA	LEU	bond	10	1.02900004

Figure 6.18 Enhanced DB-Subdue output for seq_graph

For secondary_graph , the output showing the best substructure for main memory algorithm is shown in Figure 6.19 and the output for Enhanced DB-Subdue is shown in Figure 6.20. The run-time taken by main memory algorithm was 0.1 seconds and the run-time taken by Enhanced DB-Subdue algorithm was 0.4 seconds. Here again the MDL and DMDL values returned by each approach were different although the result (substructures) was the same due to the same reason as explained before.



```
Substructure: value = 1.13828, pos instances = 2, neg instances = 0
  Graph(6v,5e):
    v 1 h_1002_14
    v 2 h_5002_5
    v 3 h_1002_18
    v 4 h_1002_16
    v 5 h_1002_16
    v 6 h_1002_15
    d 1 2 sh
    d 2 3 sh
    d 3 4 sh
    d 4 5 sh
    d 5 6 sh
```

Figure 6.19 Main memory output for secondary_graph

v1	v2	v3	v4	v5	v6	e1	e2	e3	e4	e5	x1	x2	x3	x4	c1	value
h_1002_14	h_5002_5	h_1002_18	h_1002_16	h_1002_16	h_1002_15	sh	sh	sh	sh	sh	2	3	4	5	2	1.167

Figure 6.20 Enhanced DB-Subdue output for secondary_graph

These different experiments were performed to show the correctness of the Enhanced DB-Subdue algorithm. The results given by main memory and Enhanced DB-Subdue were the same.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis we have developed several enhancements to the graph-based data mining algorithm DB-Subdue previously developed using relational databases. The different enhancements addressed in this thesis include detection of cycles and avoiding wrong substructures to be generated using cycles. Overlapping substructures were handled properly and overlap is given as an argument to the user, so that the user can allow or avoid overlap depending upon the domain. One of the main challenges was to differentiate substructures that have the same signature. This was achieved using the Database Minimum description length metric. The inexact graph match issue was also addressed. A naïve algorithm as well as an optimized approach was proposed. All the enhancements were performed using pure SQL statements to improve the performance of the algorithm. Graph mining was run on data sets that have 800K vertices and 1600K edges. All the enhancement algorithms except for the inexact graph were able to achieve functionality and the desired scalability.

We are currently developing techniques to improve the canonical labeling algorithm to achieve both functionality and scalability. Another improvement would be to extend database graph mining to concept learning and classification. The current algorithm runs for only one iteration. After we compress the graph using the best substructure discovered, the algorithm can be rerun on the compressed graph. Scalability and performance can be improved through partitioned and incremental approaches. The incremental approach will mine the data only on the new data that is added rather than mining the whole graph again after the addition is made. Biasing technique can also be introduced which will give higher weights to certain vertex and edge labels in the graph making some substructures more important than other substructures in the graph.

REFERENCES

1. Agrawal, R. and R. Srikant. Fast Algorithms for Mining Association Rules. in Proceedings 20th International Conference Very Large Databases, VLDB. 1994. Chile.

2. Han, J., J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. in ACM SIGMOD International Conference on Management of Data. 2000.

3. Thomas, S., Architectures and Optimizations for integrating Data Mining algorithms with Database Systems, in CSE. 1998, University of Florida: Gainesville.

4. Sarawagi, S., S. Thomas, and R. Agrawal. Integrating Mining with Relational Database Systems: Alternatives and Implications. in SIGMOD. 1998. Seattle.

5. Mishra, P. and S. Chakravarthy. Performance Evaluation of SQL-OR Variants for Association Rule Mining. in Dawak (Data Warehousing and Knowledge Discovery). 2003. Prague.

6. Mishra, P. and S. Chakravarthy. Performance Evaluation and Analysis of SQL-92 Approaches for Association Rule Mining. in BNCOD Proceedings. 2003.

7. Cook, D.J. and L.B. Holder, Graph-Based Data Mining. IEEE Intelligent Systems, 2000. **15**(2): p. 32-41.

8. Cook, D.J. and L.B. Holder, Substructure Discovery Using Minimum Description Length and Background Knowledge. Artificial Intelligence Research, 1994. **1**: p. 231-255.

9. Quinlan, J.R. and R.L. Rivest, Inferring decision trees using the minimum description length principle. Information and Computation, 1989. **80**: p. 227-248.

10. Brazma, A., et al. Discovering patterns and subfamilies in biosequences. in Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology. 1996.

11. Leclerc, Y.G., Constructing simple stable descriptions for image partitioning. International journal of Computer Vision, 1989. **3(1)**: p. 73 -102.

12. Pentland, A., Part segmentation for object recognition. Neural Computation, 1989. **1**: p. 82 -91.

13. Pednault, E.P.D. Some experiments in applying inductive inference principles to surface reconstruction. in Proceedings of the International Joint Conference on Artificial Intelligence. 1989.

14. Derthick, M. A minimal encoding approach to feature discovery. in Proceedings of the National Conference on Artificial Intelligence. 1991.

15. Rao, R.B. and S.C. Lu. Learning engineering models with the minimum description length principle. in Proceedings of the National Conference on Artificial Intelligence. 1992.

16. Rissanen, J. Stochastic Complexity in statistical inquiry. in World Scientific Publishing Company. 1989.

17. Bunke, H. and G. Allerman, Inexact graph match for structural pattern recognition. pattern recognition letters, 1983. **1**(4): p. 245-253.

18. Beera, R., Relational Database algorithms and their optimization for Graph Mining, in Department of Computer Science and Engineering. 2003, University of Texas at Arlington: Arlington.

19. Kuramochi, M. and G. Karypis, An Efficient Algorithm for Discovering Frequent Subgraphs. 2002, Department of Computer Science/Army HPC Research Center, University of Minnesota.

20. Read, R.C. and D.G. Corneil, The graph isomorph disease. Journal of Graph Theory, 1977. **1**: p. 339–363.

21. Fortin., S., The graph isomorphism problem. 1996, Department of Computing Science, University of Alberta.

22. Feuerstein, S., Oracle PL/SQL Programming. First ed, ed. D. Russel. 1995: O'Reilly & Associates Inc.

BIOGRAPHICAL INFORMATION

Ramanathan Balachandran was born September 29, 1979 in Chennai, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Madras University, Chennai, India in May 2001. In the Fall of 2001, 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 and Engineering from The University of Texas at Arlington, in December 2003. His research interests include Data mining, Graph mining and web-based technologies.