RELATIONAL DATABASE ALGORITHMS AND THEIR OPTIMIZATION FOR GRAPH MINING

The members of the Committee approve the masters
thesis of Ramji Beera

Sharma Chakravarthy Supervising Professor	__________________________________
Diane Cook	__________________________________
Lawrence Holder	__________________________________

RELATIONAL DATABASE ALGORITHMS AND THEIR OPTIMIZATION FOR GRAPH MINING

by

RAMJI BEERA

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

May 2003

To My Parents, Family and Friends

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. Diane Cook and Dr. Lawrence Holder for serving on my committee.

I am grateful to Anoop Sanka, Raman Adaikkalavan, Hari Prasad Yalamanchali, Naveen Pandrangi, Nishanth Reddy Vontela, and Ramanathan Balachandran 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 help, support and encouragement.

I would like to acknowledge the support of the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-2-0543), and by NSF (grants IIS-0123730 and IIS-0097517).

April 14, 2003

ABSTRACT

RELATIONAL DATABASE ALGORITHMS AND THEIR OPTIMIZATION FOR GRAPH MINING

Publication No.____

Ramji Beera, M.S.

The University of Texas at Arlington, 2002

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. Several SQL-based approaches for (association rule) mining have been studied in the literature.

The main focus of this thesis is on the design and development of algorithms for graph mining (Subdue) using relational DBMS. We develop several approaches for discovering the repetitive substructures in a graph. Each approach is analyzed and optimized further to improve its performance. Two different approaches, cursor-based and User Defined approaches are studied in this thesis. The experiments evaluate these approaches and compare their performance with the main memory algorithm for a graph-based data mining (Subdue). The larger goal of this thesis is to achieve scalability.

CHAPTER 1

INTRODUCTION

Database technology has been used with great success in traditional data processing. But with the ability to store enormous amounts of business data, it is important to find a way to mine that directly from the database and extract nuggets to leverage for business advantage. If the data can be mined directly, it can used to find abstractions or relations that improve the understanding of the data and help in making business decisions.

The large amounts of data that can be collected and stored entail that we figure out a way to interpret the data and discover interesting patterns within the data. Much of the research has addressed techniques for discovering interesting concepts from relations in databases. The techniques developed so far have dealt with data using non-structural and attribute value representations. The research has addressed issues that involve data relevance, missing data, noise and uncertainty, and utilization of domain knowledge[1]. Some of the popular data mining techniques are: classification and association rule mining*Classification* rule mining is a process of grouping items based on a classifying attribute. A model is then built based on the values of other attributes to classify each item to a particular class. *Association Rule* mining is the process of identifying the dependency of one item(s) with respect to the occurrence of other item(s). A majority of the mining algorithms were built for data stored in flat file systems. Since current database systems are dominated by relational databases the ability to perform data mining using standard SQL queries[2] will ease the implementation of data mining. SQL techniques have been successfully used for implementing for association rule mining[3].

In contrast to earlier work, recent data mining projects have been collecting structural data, which describe relations among the data objects. So, there is a need for techniques and algorithms that would decipher the relationships among data objects. A graph mining approach to data 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, protein structures, groups of related actions, etc). Subdue is a mining approach that works on a graph representation.

The goal of Subdue's[4] approach to mining is to find common and repetitive substructures within the data. The motivation for this process has been to find interesting substructures that would be able to compress the data and to identify substructures that would enhance the interpretation of the data. The Subdue substructure discovery is a process of identifying the concepts describing interesting and repetitive substructures within the structural data. Once a substructure is discovered, a pointer to the instances of the substructure is used to simplify the data.

The Subdue system discovers the interesting and repetitive substructures in the data using the principle of Minimum Description Length [1]. Subdue replaces the best substructure discovered using MDL by a single pointer and makes passes over the data, thus producing a hierarchical description of the structural data. Subdue, also uses the concept of inexact graph match to bound the algorithm computationally.

Although Subdue provides us a tool for finding the interesting and repetitive substructures within the data, it is limited by the fact that it is a main memory algorithm. The algorithm constructs the entire graph in main memory and then mines it using a search algorithm. This poses problems when the data size is very large, which is usually the case for mining applications. The algorithm needs to be mapped to a persistent representation of the graph to overcome main memory limitations. One of the approaches for providing persistence and scalability would be to use database techniques that are capable of handling large data sizes[10].

Computations over databases were not developed with arbitrary algorithms in mind. Hence, databases do not provide functionality to support "mining" in the traditional. Existing query languages such as SQL are computationally incomplete, as they do not provide all the primitive programming language constructs. The data structures are also limited to a set or a table in the cause of relational database management systems. So in order to provide computational-completeness, SQL constructs can be in a host programming language such as C or JAVA. C was chosen over Java because all the code of Subdue has been written in C.

This thesis provides an approach to substructure discovery in a database environment and uses DB2 as the DBMS. The thesis addresses mapping and representation of the substructures to tuples in the database and describes how the discovery is achieved purely through SQL-based approaches. The main focus of the thesis is on developing algorithms to discover repetitive substructures and their optimization. The algorithms have to be carefully designed to achieve this functionality and be able to work for larger data sets. The SQL queries developed have to be carefully analyzed and optimized to achieve the desired performance, which is comparable or even better than main memory algorithm. Although parallel versions of the main memory have been developed to speed up the computation and handle large data sets, they suffer from loss of information when the data set is partitioned.

The roadmap of this thesis is as follows: Chapter 2 discusses the back-ground and related work done in the field of data mining and the approaches used for mining structural data. Chapter 3 summarizes the first approach taken for the substructure discovery using the cursors. Chapter 4 summarizes the approach taken using the User Defined Functions (UDF) in DB2. Chapter 5 summarizes the second approach taken and includes various optimizations of the base approach. Chapter 6 discusses various performance related optimizations and their comparison for data sets of different sizes as well as with main memory approach performance. Chapter 7 concludes the thesis and the future work.

CHAPTER 2

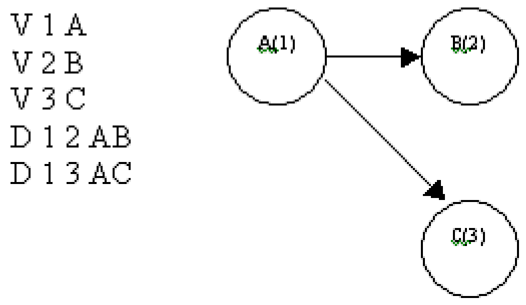
RELATED WORK

This chapter describes the Subdue main memory algorithm. It discusses how substructures are discovered in a systematic way. It also discusses how Subdue uses the graph isomorphism and inexact graph match concepts to make it a polynomial-time algorithm. It also discusses briefly the concept of Minimum Description Length (MDL) and the size-based evaluation principles used for graph compression. It summarizes the various parameters used by the Subdue algorithm, their relevance, and what they mean conceptually.

2.1 Structural Data Representation

The substructure discovery system represents the data as a labeled graph. Objects in the data represent vertices or small sub graphs and the relationships between them are represented as edges. A substructure is a connected sub-graph within the graph. Figure 2-1 shows a sample input for Subdue. An instance of the substructure in the graph is a set of vertices and edges that match the substructure theoretically.

The input to Subdue is a file, which describes the graph. All the vertices are listed first followed by the edges. Each vertex has a unique vertex number, and a label. Each edge has an edge label and the vertex numbers, to which it connects, from source to destination. The edge can be an undirected edge (u) or a directed edge (d). An edge with label e is regarded as a directed edge unless it the -undirected flag is specified at the command prompt, which will cause all edges to be treated as undirected.



```
V 1 A
V 2 B
V 3 C
D 1 2 AB
D 1 3 AC
```

Figure 2-1 Input file for Subdue

2.2 Parameters for Control Flow

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

1. Beam: This parameter specifies the maximum number of substructures kept in the substructure list to be expanded. Others are discarded. The default is 4.
2. Limit: This parameter specifies the maximum number of substructures to be evaluated *in each iteration*. The default value is (Number of Vertices + Edges)/2.
3. Size: This parameter specifies the minimum and maximum size to be reported to the user after the discovery, and maximum size also acts as a halting condition for Subdue. The size here indicates the number of vertices in the substructure.
4. Overlap: This parameter guides the algorithm to consider overlapping of the instances of the substructures. Two instances of a substructure are said to overlap is they have a vertex common to each other. Overlap plays significant role in calculating the compression value because with overlap we have to maintain extra information.
5. Nsubs: This parameter reports the best n substructures discovered.
6. 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.

7. Iterations: The Number of iterations to be made over the input graph. The best substructure from iteration **i** pass will be used to compress the graph for next iteration **i+1**. Default is 1.
8. Prune: With this argument Subdue will discard the child substructure which has lesser value than the parent substructure. This will substantially reduce the search space.

2.3 Compression Using Minimum Description Length

The minimum description length principle, described by Rissanen[5], states that the best theory to describe a set of data is a theory that minimizes the description length of the whole data set. The MDL principle has been used in various applications such as decision tree induction, image processing and various learning models of non-homogenous engineering domains. This is used for evaluating a substructure discovered by checking the number of bits needed if it is used in compressing the graph.

Subdue's implementation of MDL principle is in the context of graph compression using a substructure. Here, the best substructure is one that minimizes DL(G) + 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 substructure discovered, and DL(G|S) is the number of bits required to encode the input graph G after it has been compressed using the substructure S[4].

Let DL (G) = N (G) = number of bits needed to represent the graph.

So, N (G) returns the number of bits to represent the graph. If the graph is compressed using a substructure S in the graph which has *i* instances, then the number of bits needed to represent the compressed data would be

N (G) = N (S) + N (G/S).

The term N (G/S) represents the number of bits needed to represent the graph after compressing the graph by substituting all the instances of the sub-graph by just one vertex. The compression is better if there are more instances of the substructure in the graph. The compression achieved would be

Compression = 1 - (MDL of compressed graph)/ (MDL of the original graph)

= 1 - (N (S) + N (G/S))/N (G)

Subdue outputs the best substructures based on the above compression value.

2.4 Compression using the Size

The compression achieved by a substructure can also be evaluated using the size parameter as well. Size of a graph is defined as the number of vertices plus the number of edges in the graph. Mathematically:

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

So assuming there is no overlap between the instances of a substructure, the size of graph after compressing with the substructure would be

Size (G/S) = (Number of Vertices (G) – *i**Number of Vertices (S) + *i*) + (Number of Edges (G) – i*Number of Edges (S)), where *i* is the number of instances of the substructure S. This is an approximation of the MDL theory. This theory though uses simple and more efficient method of coding it does not capture the optimal coding used by MDL.

2.5 Inexact Graph Match

Although exact graph match can be used to find interesting substructures in the graph, most of the substructures in the graph may be slight alterations of a substructure. This difference can be attributed to the noise and distortion or might just illustrate the slight differences between the substructures in general. Comparing two graphs exactly has been shown to be an NP complete problem.

In order to deal with inexact graph matches, an approach developed by Bunke and Allerman[6] is used, where each distortion is assigned a cost. A distortion is a basic transformation such as deletion, insertion and substitution of vertices and edges. So, as long as the cost of difference between two graphs falls within a user given threshold the graphs are considered isomorphic. Employing computational constraints such as bound on the number of substructures considered and the number of partial mappings considered during the inexact graph match, keeps the Subdue algorithm to run in polynomial time.

An example of inexact graph match is shown below. The Figure 2-2 shows the two graphs that are compared.

$$\begin{tikzcd} A \arrow[r] \arrow[d] & C & A \arrow[r] \arrow[d] & D \\ B & & B & \end{tikzcd}$$

Graph1 Graph2

Figure 2-2 Inexact graph match

Assuming that an edge label is concatenation of the vertex labels, the two graphs would be different by a cost of two, namely the edge label AD and AC are different and the vertex labels C and D do not match. If a user had a threshold of two then the two graphs will be considered isomorphic.

2.6 The substructure Discovery Algorithm

Below, the algorithm used for Subdue [4] is presented.

```
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
```

2.6.1 Notations

ParentList ( ): It consists of substructures to be expanded. Initially it is empty. The number of elements in the Parent List is guided by the beam width.

ChildList ( ): It consists of substructures that are expanded. Initially it is empty.

BestList ( ): It consists of best substructures found so far.

ProcessedSubs ( ): It represents the number of substructures processed so far, hence it acts as one of the halting conditions.

2.6.2 Flow

The algorithm starts with the initializations of the Parent List, Child List and the Best List to empty sets. The parent list that contains the substructures to be expanded is populated with all the unique vertex labels in the graph which are sorted by their out degree. So each vertex is represented in the parent list as a unique substructure based on their label.

The inner while loop plays the vital role in the algorithm. Each substructure is taken from the parent list and expanded in all possible ways. This is done by adding an edge and a vertex to the instance, or just an edge if both the vertices are already present in the instance does this. The first instance of each unique expansion becomes a definition for a new child substructure. All the child instances that were expanded in this way become the instances of that child substructure. Some of the child instances, which were expanded in a different way but match the substructure within a threshold using the inexact graph match, are also included in the instances of that child substructure.

Each of the child substructures is then evaluated using the MDL heuristic and inserted into the Child List based on this heuristic. The beam width is enforced on the Child List, all the substructures after the beam width are removed and thus do not participate in the future extensions. The Best List also uses the same mechanism to keep its cardinality to the limit specified. Once the Best List and the Child List are updated, the Parent List is swapped with the Child List, which would be then used to make the next round of extensions. The algorithm's run time is guided by the user specified beam width and the Limit. The inexact graph match [6] is used to bound the run time.

2.6.3 Halting Conditions

There are many halting conditions for the algorithm. All of these parameters have a default value, which can be changed by the user. The most significant of these parameters is the Limit, which is basically the limit on the number of substructures processed so far. Although the default is set to (Number of vertices + Number of Edges)/2 the user can give a value that has a bearing on the output of the program.

One of the other halting conditions is the pruning parameter, which is more of a graph-dependent parameter unlike the limit, which is just a number. Initially in the discovery process, the number of instances of each substructure is usually very large, but as the discovery process continues the size of the substructure increases and thus the number of instances reduce. Using the pruning mechanism, which discards child substructures with values less than the parents, we can have a halting condition when there would be no child substructures left after pruning. Without the pruning argument the child list is always kept full no matter what the value of the substructure is as compared to the parent substructure.

Another way of halting the algorithm is using the size parameter, which controls the maximum size of a substructure. For example, a maximum size of 5 guides the algorithm to not explore a substructure of size greater than 5 (number of vertices). In the Child List none of the substructures with size greater than 5 are inserted and thus emptying the Child and the Parent List. The minimum size parameter does not have a bearing on the halting but it has an effect on the substructures inserted inside the Best List. The minimum size also guides the output to show only those substructures, which have a size greater than or equal to the size mentioned as the minimum size.

2.6.4 Next Iterations

After finding the best substructure, which would compress the graph in the first iteration, this substructure would be actually used to compress the graph by replacing each of the substructures in the graph by a single node. Although each of the substructures is compressed to a single node, it still needs to maintain other information about the edges connecting the rest of the graph. Once the graph is constructed, this graph would be used for the next iteration as the input graph for finding interesting substructures. This process can continue depending on the number of iterations the user might specify or either the algorithm fails to find a substructure, which can compress the graph. So according to the algorithm it might never even go to a second iteration if it is never able to find a substructure, which can compress the graph in that iteration.

CHAPTER 3

CURSOR-BASED APPROACH

This chapter introduces the first of the approaches taken for Subdue discovery process using a relational DBMS. It includes the basics of writing static SQL in C programs. The basic idea in this approach is to use the cursor operations in DB2 to update the count (number of instances) of each substructure. The algorithm starts with initializing the data from the input. All the single-edge substructures are stored in the Joined_1 table. Cursors are used to compute the count of each substructure. The count attribute indicates the number of instances of the substructure. The count attribute is used to prune the substructures containing only a single instance. Single-instance substructures cannot create larger substructures of counts greater than one if exact graph match is used. The substructures are expanded by a single edge and stored in a different table. The expansion is done using the *join* operator in SQL. The above algorithm is repeated for the extended substructures. The halting condition would be to reach the maximum size of the substructure, which is a user-specified number.

The reason behind choosing this approach is that SQL is not a computationally complete language and hence the Subdue main memory algorithm cannot be applied in a database context. Mapping the graphs representation to the existing structures (*tuples* and *tables*) in the database is important. Concepts from databases, such as *cursors*, UDF’s and stored procedures have to be used to achieve the functionality.

3.1 Using static SQL in C programs

DB2 UDB (Universal Database) [7] provides two ways in which an application program can interact with a database, called static SQL and dynamic SQL. In static SQL, the application developer must know exactly what SQL statements are needed and embed these SQL statements directly into an application program. The program is then processed by the DB2 pre-compiler, which converts each SQL statement into an optimized access plan and stores the plan in the database. In the application program the original SQL statements are replaced by calls to run time routines that load and execute the access plans. Static SQL provides good performance because it optimizes the SQL statements at compile time and prepares the access plan. The alternative to static SQL is dynamic SQL, which presents SQL statements to the database at run time.

Each SQL statement has to be prefixed by the two words EXEC SQL. Host variable is the name of the variable declared in the program in which SQL statement is embedded. The name of a host variable is distinguished from the column name with a colon prefix to the host variable. Since the database columns and host variables are not in the same name space, host variables can be named after the column names with which they compare. All the declarations of host variables must be declared in a *declare section:* these variables are specially marked for processing by the compiler. A simple example of using host variables and embedded SQL is shown below.

- Inserting a new row into a table called SUPPLIERS from input host variables

```
EXEC SQL
     INSERT INTO SUPPLIERS(suppno,name,address)
     VALUES (:suppno,:sname,:saddr)
```

3.2 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 the 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 host programs, 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 [7]

```
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;
```

3.3 Discovery Algorithm

The steps of the algorithm remain the same for the database approach. The first step is to find substructures of length n and sort them based on their count. The count of a substructure corresponds to the number of substructures that exactly match the current substructure. These substructures of size n will be used for the extensions to generate substructures of size n + 1. In this algorithm beam and limit are not used.

Pseudo Code for this algorithm is given below:

```
 1)   Subdue-DB(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   two copies of joined_i-1 table
 8)        DECLARE   Cursor c1 on Joined_i
 9)        DECLARE   Cursor c2 on Joined_i
10)        WHILE (c1)
11)             FETCH     c1   into g1
12)             WHILE(c2)
13)                  FETCH     c2   into g2
14)                  If (Isomorphic (g1, g2) =0)
15)                  Update    Joined_i
```

```
16)                        Set  count = count + 1
17)                        Where     current of c1
18)          Delete from c1 where count = 1
20)    i++
```

3.3.1 Initialization of data

The algorithm starts with initialization of data. The input is read from a delimited ASCII file and loaded into the specific tables. The input file created from the graph generator is not compatible for loading the tuples in the table. A function called change_db accepts the input file for Subdue and creates two files, the vertex file and the edge file. The file created is a delimited ASCII file, which consists of streams of data values- ordered by row and by column within each row. The comma delimiter separates column values and the new line character separates each row.

Below is an example of a delimited ASCII file, which loads all the edges into the edges table.

```
1,2,e1
3,2,e1
4,3,e2
5,6,e1
7,6,e1
8,7,e2
9,10,e1
```

Once the data is loaded into the tables, the table has values as shown in Figure 3-1.

Vertex1	Vertex2	EdgeName
1	2	E1
3	2	E1
4	3	E2
5	6	E1
7	6	E1

Figure 3-1 Tuples in EDGES table

The Vertices table is also loaded in the same way. The Vertices table is shown in Figure 3-2.

VertexNo	VertexLabel
1	A
2	B
3	C
4	A

Figure 3-2 Tuples in Vertices table

The next step in the algorithm is the initialization of the Joined_1 table. The Joined_1 table will consist of all the substructures of size one, size being the number of edges. The new table Joined_1 has been created because the edges table does not contain information about the vertex labels. So the Edges table and the Vertex table are joined to get the Joined_1 table. The SQL query for doing this would be

```
Insert    into Joined_1(Vertex1, Vertex2, Vertex1name,
          Vertex2name, edgename)
```

```
(
 Select   e.Vertex1, e.Vertex2, v1.VertexLabel,
          v2.VertexLabel, e.EdgeName
 From     Edges e, Vertices v1, Vertices v2
 Where    e.Vertex1 = v1.VertexNo and e.vertex2
          = v2.VertexNo
)
```

The resultant table Joined_1 table is shown in Figure 3-3.

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
7	6	E	D	E1

Figure 3-3 Joined_1 table

3.3.2 Substructure Discovery

The substructure discovery algorithm starts with one-edge substructures unlike in Subdue, which starts with all the unique vertex labels. In the database version, each instance of the substructure is represented as a tuple in the table. The next step in the algorithm is getting the counts of the individual substructures. The count attribute indicates the number of substructures that are similar or exactly match the substructure under consideration. In SQL, there is no way of distinguishing one substructure from another, since each of the substructures is represented by a tuple in the table. In this method the count of each substructure is updated by comparing it with every other substructure. This was the first method developed for the database environment and has n squared complexity, where n is the number of tuples in the table. In order to compare each tuple with every other tuple, cursors are used to retrieve the information for each tuple. The SQL query to retrieve the information can be expressed as,

```
Declare   Cursor graph1 for
          Select    Vertex1, Vertex2, Vertex1Name,
                    Vertex2Name, EdgeName
          From      Joined_1
```

Similarly, another cursor, *graph2* is declared on the same table. Each tuple in the Joined_1 would be compared to every other tuple in the table, using the isomorphism code found in Subdue. With the information taken from the cursor, a graph is constructed. Since the substructure is a single edge, the graph would be a single-edge graph. For a given substructure, the count of that substructure is increased by one if any other substructure is isomorphic to this one. After this pass, each tuple will have an attribute count, which indicates the number of substructures to which it is isomorphic. At the end of this pass, the table Joined_1 will have the values shown in Figure 3-4.

Vertex1	Vertex2	Vertex1Name	Vertex2Name	EdgeName	Count
1	2	A	B	E1	2
3	2	C	B	E1	1
4	3	A	C	E2	3
5	6	B	D	E1	1
7	6	E	D	E1	1

Figure 3-4 Joined_1 table after count attribute updated

The count essentially captures the number of instances of that substructure, so a count of five means the substructure has five occurrences in the graph. The tuples with count one are substructures with only instance and hence any bigger graph that contains this edge will have a count of one, so these tuples are removed from the table. So in the Joined_1 table only those tuples with count greater than one are retained.

3.3.2.1 *Two-Edge Substructures*

In a main memory approach, every substructure, which is necessarily a sub-graph, can be defined as a *structure* in the programming language. Extensions to two or more edges are generated by growing the substructure appropriately. In the database environment as there are no *structures,* the only information to be used will be the single edge substructures, which are basically tuples in the Joined_1 table. The number of attributes of the table needs to be increased to capture substructures of increased size.

The Joined_1 table is joined with itself to get the Joined_2 table, which will have all the substructures with two edges. The tuples in this table would have the information about all the two-edge substructures that includes the edge names and vertex names. The SQL query needs to generate all possible two-edge substructures with no duplicates. The single-edge substructure can be extended to two-edge substructures either on the first or second vertex. Hence there will be two queries, one each for extending on each vertex, to generate the two-edge substructures. Also the queries need to make sure that duplicates are not generated.

For example consider the graph shown in Figure 3-5,

$$\begin{tikzcd}
A(1) \arrow[r] \arrow[d] & B(2) \arrow[r] & A(4) \arrow[r] \arrow[d] & C(5) \\
C(3) & & B(5) &
\end{tikzcd}$$

Figure 3-5 Graph

For the graph in Figure 3-5, the joined_1 table after updating the count attribute has the following values.

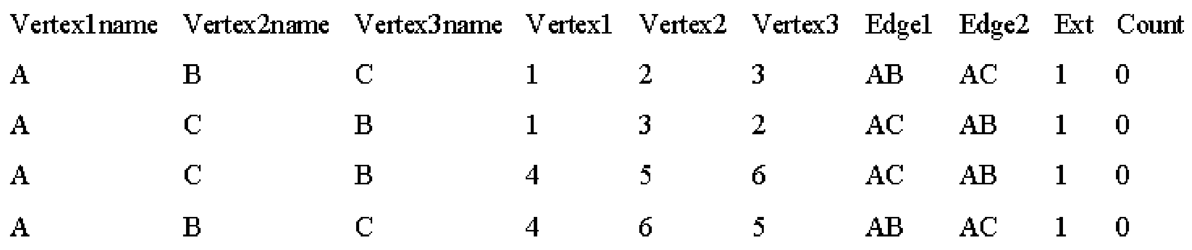
Vertex1	Vertex2	Vertex1Name	Vertex2Name	EdgeName	Count
1	2	A	B	AB	2
1	3	A	C	AC	2
4	5	A	C	AC	2
4	6	A	B	AB	2

Figure 3-6 Joined_1 table

The query to generate all the possible two edge substructures is shown below. This query does not eliminate the duplicates.

```
Insert    into
          Joined_2(Vertex1,Vertex2,Vertex3,Vertex1Name,Ve
          rtex2Name,Vertex3Name,
          Edge1Name,Edge2Name,Ext1,count)
          (
          Select
                    (j1.vertex1,j1.vertex2,j2.vertex2,j1.
                    vertex1name,j1.vertex2name,j2.vertex2
                    name,j1.edge1name,j2.edge1name,1,0)
          From      Joined_1 j1, Joined_1 j2
          Where     j1.vertex1=j2.vertex1 and
                    j1.vertex2!=j2.vertex2
          Union
          Select
                    (j1.vertex1,j1.vertex2,j2.vertex2,j1.
                    vertex1name,j1.vertex2name,j2.vertex2
                    name,j1.edge1name,j2.edge1name,2,0)
          From      Joined_1 j1, Joined_1 j2
          Where     j1.vertex12=j2.vertex1
```

The resulting Joined_2 table is shown in Figure 3-7



Vertex1name	Vertex2name	Vertex3name	Vertex1	Vertex2	Vertex3	Edge1	Edge2	Ext	Count
A	B	C	1	2	3	AB	AC	1	0
A	C	B	1	3	2	AC	AB	1	0
A	C	B	4	5	6	AC	AB	1	0
A	B	C	4	6	5	AB	AC	1	0

Figure 3-7 Joined_2 table

The table Joined_2 has an attribute **Ext**, which aids in constructing the graph. Every tuple in the Joined_2 table represents a two-edge substructure. The attributes of the table vertex numbers, labels and edge labels give the information about the substructure. But the attribute **ext** describes the direction of each edge in the graph. In the Joined_1 table the edge is always from the first to the second vertex. But in the Joined_2 table though the first edge is from vertex one to vertex two we cannot say that for the second edge. The information known is that the vertex three is part of the edge but the direction and the connecting vertex is not known. For this reason the **ext** attribute is maintained. For example, if the **ext** is 1 then the edge is from vertex 1 to vertex 3. In general in an N edge graph if **Exti** is j then the i+1th edge is from the vertex number in the attribute vertex j to the vertex number in the attribute vertex i+2.

The first and second tuples are duplicates in the above table, and so are the tuples fourth and the fifth. Hence when the count is updated it would be wrongly updated to 4 for each tuple, because every tuple is isomorphic to every other tuple. To overcome this problem the criterion for extension needs to be changed. Instead of extending two different tuples to the same substructure we restrict the extension to only one tuple. In the where condition instead of having ***j1.vertex2 != j2.vertex2***, having ***j1.vertex2 < j2.vertex2*** would ensure there are no duplicates generated by the join. By having the less than condition we are limiting the extension to only one tuple. The condition also satisfies the completeness of the algorithm, that is, all substructures are generated. The completeness is ensured because only one of the tuples A->B or tuple A->C is extended to the other. For the above example, the resulting table would be

Vertex1name	Vertex2name	Vertex3name	Vertex1	Vertex2	Vertex3	Edge1	Edge2	Ext	Count
A	B	C	1	2	3	AB	AC	1	0
A	C	B	4	5	6	AC	AB	1	0

Figure 3-8 Joined_1 table

The count of each tuple is then updated, to two in this example. If there are any tuples with a count of one they are deleted from the table.

3.3.2.2 Generalization

For higher extensions, substructures having more than two edges need to be generated. The substructures with **n** number of edges are stored in the Joined_**n** table, the attributes in that table would be **n**+1 vertex numbers, **n**+1 vertex names, **n** edges, **n**-1 extensions and one attribute for the count. The extensions are needed to know the connectivity within the graph. For example, in a three-edge substructure if the extensions have the value twos and three, it means that the second edge is from vertex two to vertex three and the third edge is from vertex three to vertex four. The information contained in vertex names and edge names are also important because they aid in constructing the graph and play an important role in detecting isomorphism. The vertex numbers are needed for the higher extensions.

A generalized query for generating the **n** edge substructures can be expressed as

```
Insert    into Joined_n
```

```
Vertex1,Vertex2…..VertexN+1,Vertex1Name,Vertex2
Name…..VertexN+1Name
Edge1Name,Edge2Name…..EdgeNName,Ext1,Ext2…..Ext
N-1,0
(
Select
          j1.Vertex1,j1.Vertex2….j1.vertexN,j2.
          vertex2,j1.Vertex1Name,j1.Vertex2Name
          …j1.vertexNName,j2.vertex2name,j1.edg
          e1Name,j1.edge2name….j1.EdgeN-
          1Name,j2.edgename,j1.Ext1,j1.Ext2…..j
          1.ExtN-2,p,0
From      Joined_N-1 j1, Joined_1 j2
Where     j1.vertexP= j2.vertex1 and
          j1.vertexP+1 <
          2.vertex2…j1.VertexN<j2.Vertex2
)
```

The number **P** varies from 1 to **N**. Since an **N**-1-edge substructure can be extended from any of the n possible vertices, the number of queries needed would be n. The number P in the above query achieves this functionality. The idea here is that an edge could be added to the existing substructure to get a larger substructure.

3.3.2.3 Negative Extensions

All the above queries assume that all edges are outgoing from a vertex, but we need to handle graphs with incoming edges.

For example consider the graph shown in Figure 3-9:

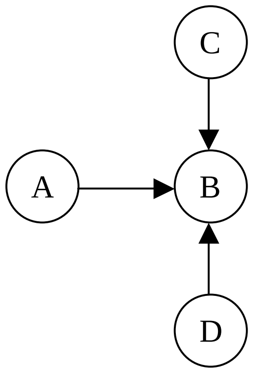


Figure 3-9 An example graph with incoming

edges

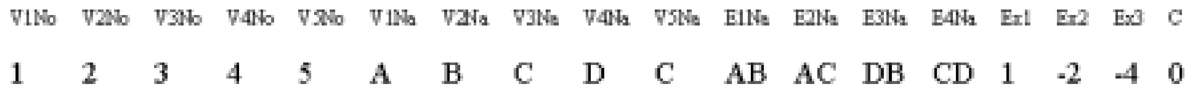
In the above graph all the edges are coming into the vertex B. In the first pass all the single edges are detected correctly as AB, CB and DB but in the second pass there are no edges going either out of A or B or any other vertex, so extending by the query described above, would result in no tuples in the Joined_2 table although there are several 2-edge substructures. In order to overcome this problem, in addition to extending edges, which are going out from the vertex, the query should also extend by those edges that are coming into the vertex. Distinction has to be made between an edge going out from a vertex and an edge coming into the vertex, since the difference cannot be inferred from the vertex number or the label. We use the extension number to differentiate them. For all the edges coming in, the extension number will be negative. For example if the ext1 attribute has a value –2 that means the second edge is from vertex 3 to vertex 2 and if the ext1 has attribute value 2 then the edge is from vertex 2 to vertex 3. In general if the extension ***i*** is **–j**, then the edge i+1 is from vertex i+2 to j. The query for generating all the edges can be expressed as,

```
     Insert     into Joined_N
                Vertex1,Vertex2…..VertexN+1,Vertex1Name,Vertex2
                Name…..VertexN+1Name
                Edge1Name,Edge2Name…..EdgeNName,Ext1,Ext2…..Ext
                N-1,0
                (
                Select
                          j1.Vertex1,j1.Vertex2….j1.vertexN,j2.
                          vertex2,j1.Vertex1Name,j1.Vertex2Name
                          …j1.vertexNName,j2.vertex2name,j1.edg
                          e1Name,j1.edge2name….j1.EdgeN-
                          1Name,j2.edgename,j1.Ext1,j1.Ext2…..j
                          1.ExtN-2,-p,0
                From      Joined_N-1 j1, Joined_1 j2
                Where     j1.vertexP= j2.vertex2 and j1.vertex1
                          < j2.vertex2….j1.VertexP-1<j2.Vertex2
                )
```

P is a variable from 2 to N

3.3.2.4 Constructing the graph

In order to use the isomorphism code, two graphs have to be constructed in the form of strings and given as input to the isomorphism code. The string should be similar to the input given to Subdue except that the information is not a file input but a string. For constructing the graph from the tuple, the needed information are the vertices and the edges in the graph. Since the tuple stores all the vertex numbers and the vertex labels, they are loaded as it is, but for the edges, the edge names are known but it does not give the information as to how the graph is connected between the vertices. The extensions, which are maintained as attributes in the table help in constructing the graph. For example, if the ext3 attribute has value 1, that means the fourth edge is from the vertex number in attribute vertex one to vertex number in attribute vertex five. For example, consider a tuple in the Joined_4 table, which contains all of the four-edge substructures. V in the table stands for the Vertex, E stands for Edge and Ex stands for extension. No stands for number and Na stand for name



V1No	V2No	V3No	V4No	V5No	V1Na	V2Na	V3Na	V4Na	V5Na	E1Na	E2Na	E3Na	E4Na	Ex1	Ex2	Ex3	C
1	2	3	4	5	A	B	C	D	C	AB	AC	DB	CD	1	-2	-4	0

Figure 3-10 A tuple in Joined_4 table

From the information in the table the graph can be constructed as follows. First, all the vertex numbers and their labels are enumerated as shown below

V 1 A

V 2 B

V 3 C

V 4 D

V 5 C

The second part of creating the graph is deriving the information about the edges. The first edge is written without any information from the table because the direction is always from the first vertex to second vertex. For the rest of the edges, depending on the extension number the edge direction is determined. If the extension number is negative, then the edge is coming into the vertex, else it is going out. Since the extension 2 and 3 are negative they are the edges coming into the respective vertices.

So the edges for the above tuple are

D 1 2 AB

D 1 3 AC

D 4 2 DB

D 5 4 CD

The constructed graph is shown in Figure 3-11.

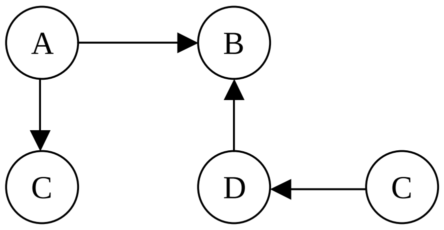


Figure 3-11 Constructed graph from the tuple

Once the graphs are constructed they can be used as input to the isomorphism code, which returns a floating-point number indicating how different the graphs are. So if the returned number is zero, they are isomorphic.

3.3.3 Input data generation

The input graphs have been created using the *graphgen* code developed by the AI group at UTA [8]. The program reads the parameters from a file and creates a graph. The file has the following parameters, each described on a new line.

1) Number of vertices in the graph
2) Number of edges in the graph
3) Number of distinct vertex labels
4) Number of distinct edge labels
5) Number of substructures to be embedded in the graph

6) Number of patterns to embed in the graph

    For each pattern

    i. Number of instances

    ii. Number of vertices

        For each pattern vertex

        - Each vertex label of form v#, where # is less than the number mentioned in the parameter 3

    iii. Number of edges

        For each pattern edge

        - The edge label of form e#, where # is less than the number mentioned in 4.
        - The first vertex that this edge is attached to. An integer ranging from 0 to (8.) minus one.
        - The second vertex that this edge is attached to. An integer ranging from 0 to (8.) minus one.

The substructures of size three and size four have been embedded in the graphs. The number of instances of each of the substructures has been set to 3% the number of edges in the graph. The substructures that were embedded inside the graph have been described in Figure 3-12. The number of vertices is set to half the number of edges, and the number of distinct vertex labels and edge labels have been set to half the number of vertices and edges, respectively.

3.3.4 Performance

This being the first approach, we wanted to get a feel for the time taken by the database approach and how it compares with the main memory approach (Subdue 4.3.a.1) for the same graph. Since there was no pruning in this approach the comparison is not exactly the same, as the main memory implements all the basic pruning techniques by using the beam and the limit.

Table 3.1 shows the configuration of the database we used for running the test cases.

Table 3-1 Configuration of the Database – SubdueDB

PageSize	4KB
LogFileSize	40000*PageSize(4KB)
Database	DB2(UDB)
Version	6.1

Table 3-2 Configuration of the system

RAM	348MB
Hardware	SUNW,Ultra-5_10
OS version	5.6
Compiler	GCC

Experiments were performed to compare the run times of each of the two approaches, main memory and database. Table 3-3 shows the total times taken by both Subdue and the database approach for the respective data sets. From the results we can clearly see that the database approach is no comparison to the main memory approach. This can be attributed to the UPDATE operation we are doing, which is one of the most expensive operations in a database.

Table 3-4 shows the individual timings taken by the database algorithm. Table 3-4 shows the timings for the Subdue approach. The timings have been divided for each size (of the sub-graph), namely 1 edge, then 2 edge, and so on. For the one-edge pass the timings are given for the cursor operations and deletion time. For the higher edges the time taken for extension, cursor and delete operations are also mentioned. The data set is represented as T**n**V**m**E where **n** represents the number of vertices and **m** represents the number of edges.

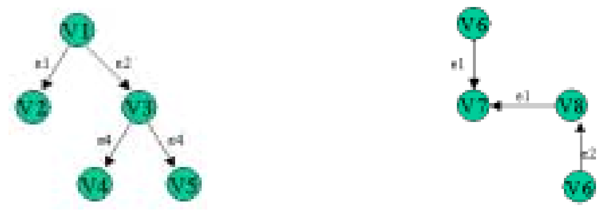
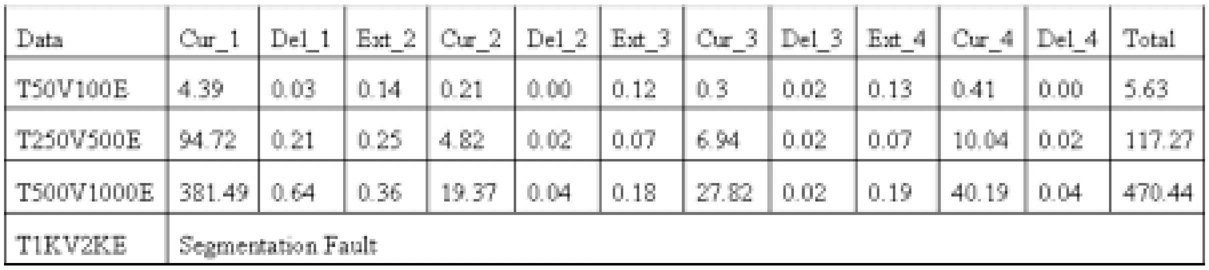


Figure 3-12 The substructures embedded in the graph

Table 3-3 Timings comparison

Data Set	Database	Subdue
T50V100E	5.63	0.17
T250V500E	117.27	3.56
T500V1000E	470.44	13.21
T1000V2000E	Segmentation fault	45.11

Table 3-4 Individual timings for Cursor-based approach



Data	Cur_1	Del_1	Ext_2	Cur_2	Del_2	Ext_3	Cur_3	Del_3	Ext_4	Cur_4	Del_4	Total
T50V100E	4.39	0.03	0.14	0.21	0.00	0.12	0.3	0.02	0.13	0.41	0.00	5.63
T250V500E	94.72	0.21	0.25	4.82	0.02	0.07	6.94	0.02	0.07	10.04	0.02	117.27
T500V1000E	381.49	0.64	0.36	19.37	0.04	0.18	27.82	0.02	0.19	40.19	0.04	470.44
T1KV2KE	Segmentation Fault											

3.3.5 Conclusion

From the performance point of view, the database algorithm does not do very well. The best case input for the algorithm would be only one occurrence of all the edges, which would make the algorithm, stop after the first pass because there would be no tuples participating in the next pass. The worst-case input for the algorithm would be a large number of repeating edges and a large number of instances of each substructure. Although performance-wise the algorithm does not perform as good as the main memory, functionality-wise, it discovers all the substructures. The user can also mention a threshold for graph isomorphism, which would make the algorithm to check for substructures, which need not be exact graph match but can differ by the threshold specified by the user.

From the results it is evident that a large amount of time was used for the first pass. The reason for this is that the graph generator generates the output in such a way that all the substructures are first embedded and then fills the remaining graph with edges appearing only once and thereby reducing the number of tuples participating in the higher extensions. This entails that for improvement; subsequent approaches should focus on minimizing the time taken for updating the counts. The maximum data set that completed successfully was the 1000 edges graph. The next data set could not complete because of a segmentation fault. The segmentation fault occurred in the isomorphism code, the number of comparisons made for a 2000 edge graph would be 2000*2000, so the program (or the data space/buffer used for this purpose) runs out of memory.

CHAPTER 4

UDF-BASED APPROACH

This chapter explores an alternative to the cursor-based approach described in the previous section. User-defined functions (or UDFs) are unique to DB2 and were introduced to provide better interaction between SQL statements and language (C, C++, Java) code. The idea is to offer a tighter integration between relational and algorithmic approaches that is not provided efficiently by the cursor-based approach. In this approach, user-defined functions can be invoked as part of SQL statements, tables can be returned from those functions, and memory allocation as well as complex data structures (that cannot be created using SQL) can be created in UDFs. Also, UDF's execute in two modes: fenced and unfenced. In the fenced mode, user code is executed in a separate address space to ensure that it does not crash the database server. This is also somewhat inefficient, as the data needs to be passed from one address space to another. In the unfenced mode, the user code is executed as part of the database server address space. The performance is better (as data is passed within the same address space) but at the risk of crashing the system. Typically, the user code is debugged using the fenced mode and then executed in the unfenced mode to achieve better performance.

We wanted to explore this approach to determine its effectiveness as compared to the previous approach. Our preliminary results are reported in this chapter.

4.1 Functions in UDB system

1) *Built-in Functions*

Some functions are built into the code of the UDB system. These functions are found in the SYSIBM schema. Some of the functions are

- Arithmetic and String operators: +, -, *, /, || etc
- Scalar functions: *substr*, *concat*, *length*, *days* and so on
- Column functions: *avg, count, min, max, stdev, sum, variance*

In addition to the built in functions in the SYSIBM schema, many other functions are shipped with UDB, in the SYSFUN schema. Although these functions are shipped with the system, they are not implemented directly by system code. They are implemented as preinstalled external functions [7].

2) *System-generated Functions*

These functions are automatically generated when a distinct type is created and are found in the same schema as the distinct type. They include casting functions and the comparison operators for the distinct type [7].

3) *User-Defined Functions*

The user, using a statement called CREATE FUNCTION [7], which names the new function and specifies its semantics, explicitly creates these functions. UDF's are further classified into the following sub categories

- *Sourced Functions*

  A sourced function duplicates the semantics of another function, called its source function. A sourced function can be an operator, a scalar function, or a column function. Sourced functions are particularly useful for allowing a distinct type to selectively inherit the semantics of its source type.

- *External scalar functions*

  An external scalar function that is written by a user in a host programming language and that returns a scalar value. External functions can be written in C or JAVA. The CREATE FUNCTION statement for an external scalar function tells the system where to find the code that implements the function. The name of the function can be an operator like '+'. The external function can do computation on the parameters passed to it but cannot access or modify the database.

- *External table Functions*

  A UDF can return a table rather than just a scalar value. Similar to the external scalar function the table function is written in C or JAVA, and cannot contain any embedded SQL statements. The program should return a tuple to the result table each time it is called, and must indicate the end of the result table by a special return code.

SQL supports the concept of function overloading. This means that several functions can have the same name, as long as they are different schemas or take different types of parameters.

4.2 Creating an External Scalar Function

An external function is a function whose implementation is written in some host programming language. The ability to create own external functions is a powerful feature in

UDB. To enhance the usefulness of built in data types by adding new functions that operate on them is a very powerful feature in DB2 because SQL by itself is not a complete programming language. The external functions are defined and installed in the database. These functions can be shared among all the database applications, which will avoid duplicating the code in each application. External functions can be used wherever built in functions are used.

The syntax of a CREATE FUNCTION [7] statement to create an external scalar is as follows:

```
>>-CREATE FUNCTION--function-name---------------------------->
 >----(--+------------------------------------+---)---*------>
         '----data-type1--+------------+--+--'
>----RETURNS--+-data-type2--+------------+----------------+>
               '-data-type3--CAST FROM--data-type4--+-------+-'
>----*----+--------------------------+--*------------------->
          '-SPECIFIC--specific-name--'
 >-----EXTERNAL--+--------------------+---*--------------->
                 '-NAME--+-'string'---+-'
                         '-identifier-'
 >----LANGUAGE--+-C----+--*---PARAMETER STYLE--+-DB2SQL------>
               +-JAVA-+                       '-DB2GENERAL-'
               '-OLE--'
                         (1)          .-FENCED-----.
>----*----+-DETERMINISTIC-------+--*----+-----------+--*---->
          '-NOT DETERMINISTIC---'       '-NOT FENCED-'
       .-NOT NULL CALL--.
>-----+---------------+--*--NO SQL--*---------------------->
      '-NULL CALL------'
                                        .-NO SCRATCHPAD--.
```

```
>-----+-NO EXTERNAL ACTION-+--*----+---------------+--*----->
      '-EXTERNAL ACTION----'       '-SCRATCHPAD-----'
       .-NO FINAL CALL--.       .-ALLOW PARALLEL----.
>-----+-----------------+--*----+-------------------+--*------>
      '-FINAL CALL-----'       '-DISALLOW PARALLEL-'
       .-NO DBINFO--.
>-----+------------+--*------------------------------------><
      '-DBINFO-----'
```

4.2.1 Description of the Syntax

- Function name

The name of the function being defined. It is a qualified or an unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier (with a maximum length of 18 characters). The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter must not identify a function described in the catalog. The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

- Data type1

Identifies the number of input parameters of the function, and specifies the data type of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed. It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example,

CREATE FUNCTION changename() ...

No two identically named functions within a schema are permitted to have exactly the same type for all corresponding parameters. It can also specify the data type of the parameter.

- RETURNS

This mandatory clause identifies the output of the function.

- Data Type 2

Specifies the data type of the output. In this case, exactly the same considerations apply as for the parameters of external functions described above under *data-type1* for function parameters.

- Data Type 3 CAST FROM Data Type 4

Specifies the data type of the output. This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code. For example,

```
CREATE FUNCTION GET_PAY_DATE(CHAR(6))
     RETURNS DATE CAST FROM CHAR(10)
```

The function code returns a CHAR(10) value to the database manager, which, in turn, converts it to a DATE and passes that value to the invoking statement. The *data-type4* must be *castable to* the *data-type3* parameter.

- SPECIFIC specific-name

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server. The *specific-name* may be the same as an existing *function-name*.

If *specific-name* is not specified, the database manager generates a unique name.

- EXTERNAL

This clause indicates that the CREATE FUNCTION statement is being used to register a new function and tells the system how to find the C function that serves as its implementation. This function must be compiled, linked and placed in a directory on the server machine, from which it can be dynamically loaded by the database system when needed. The most complete form of an EXTERNAL clause gives the full path name of the binary file that implements the function, followed by a "!", followed by the name of the proper entry point in that file. For example, the following clause tells the system where the function *isomorph* is implemented in the file *myudf.c*

EXTERNAL NAME '/cse/home/ramji/udfs/myudf!isomorph'

If no path name is specified the system looks for the function in the *sqllib/function* directory associated with the database.

- LANGUAGE

This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

C

This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA

This means the database manager will call the user-defined function as a method in a Java class.

- PARAMETER STYLE

This clause is used to specify the conventions used for passing parameters and returning the value from functions. With language C, PARAMETER STYLE DB2SQL should be specified and with language JAVA, PARAMETER STYLE DB2GENERAL should be specified.

- DETERMINISTIC or NOT DETERMINISTIC

This mandatory clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC function would be a random-number generator. An example of a DETERMINISTIC function would be a function that determines the square root of the input.

- FENCED or NOT FENCED

This clause specifies whether or not the function is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its internal resources (e.g. data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED. SYSADM authority, DBADM authority or a special authority (CREATE_NOT_FENCED) is required to register a user-defined function as NOT FENCED.

- NOT NULL CALL or NULL CALL

This optional clause may be used to avoid a call to the external function if any of the arguments is null. If NOT NULL CALL is specified and if at execution time any one of the function's arguments is null, the user-defined function is not called and the result is the null value. If NULL CALL is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

- NO SQL

This mandatory clause indicates that the function cannot issue any SQL statements.

- NO EXTERNAL ACTION or EXTERNAL ACTION

This mandatory clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Specifying EXTERNAL ACTION prevents optimizations that assume functions have no external impacts. For example, sending a message, ringing a bell, or writing a record to a file.

- NO SCRATCHPAD or SCRATCHPAD

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. This scratchpad has the following characteristics:

It is 100 bytes in size.

It is initialized to all X'00"s.

It is persistent. Its content is preserved from one external function call to the next. Any changes made to the scratchpad by the external function on one call will be there on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each sub query. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.

- NO FINAL CALL or FINAL CALL

When a function is used in an SQL statement, the function may be called multiple times during the processing of the statement, depending on how it is used. This optional clause specifies whether the function is called one extra time at the end of processing the

SQL statement. The purpose of such a final call is to enable the external function to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external function acquires system resources such as memory and anchors them in the scratchpad.

- ALLOW PARALLEL or DISALLOW PARALLEL

This optional clause specifies whether, for a single reference to the function, the invocation of the function can be parallelized. In general, the invocations of most scalar functions should be parallelizable, but there may be functions (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL is specified for a scalar function, then DB2 will accept this specification.

- DBINFO

This optional clause causes UDB to pass an extra parameter to the function, containing a pointer to a data structure containing information such as the current database, current author id, and the name of the table and column that is being modified by the current statement.

An example of a CREATE FUNCTION statement,

```
CREATE FUNCTION isomorph(VertexNo,VertexName)
RETURNS INT
EXTERNAL NAME ‘/cse/home/ramji/udfs/myudf !
isomorph’
DETERMINISTIC
NO EXTERNAL ACTION
NULL CALL
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL;
```

4.3 UDF’s over Cursors

One of the main reasons for choosing DB2, as the database is the functionality of UDF’s provided by the database. UDF’s are one of the most powerful tools that allow the user to program in a host programming language like C or JAVA. Although the first approach taken provided good functionalities, the running time of the approach was very large to continue using the approach. One of the main overheads in using that approach was that of cursors, although when the algorithm is pushed to the limit it runs out of memory in the isomorphism code, the time taken by the cursors is also one of the main overheads. Cursors are one of the main overheads when working in a static SQL code, the usage of the cursors has to be minimized as much as possible. But the reason behind using the cursors has been that each tuple in the table has to be computed against every other tuple in the table to get the count of each of the substructures that are isomorphic. In order to substitute the cursors by some other application, the application should provide the functionality of retrieving the values of the attributes and work on them. The next consideration in choosing the UDF's should be the run time of the UDF’s over cursors. The UDF’s would run much faster than the cursors because they are just like the system built functions but written by the user and stored in the specific directory, and the user determines it can be used.

4.3.1 Implementation Details

The information needed to compute the isomorphism between two graphs is their vertex labels, edge labels and the connectivity between the edges. The basic working of the algorithm still remains the same except that instead of using cursors, UDF's are used to compute the number of instances of each of the substructure. The pseudo code for the algorithm is given below:

```
1)      Subdue-DB(input file,size)
2)          Load vertices into vertices table;
3)          Load edges into edges table;
4)          Load joined_1 table : join vertices and edges
            table
5)          i=2;
6)              WHILE(i<size)
7)              Load Joined_i table (substructures of size
                i) from joined_i-1 table
8)              IsomorphismUDF(Joined_i,Joined_i);
9)              Update_count;
10)             Delete from Joined_I where count=1;
```

The flow of the algorithm is same as that of the cursor approach except that the updating of the counts of the substructures is implemented using the UDF's. The mapping and the extensions of the substructures are same as that of the cursor approach.

The UDF would take in as inputs the attributes of the table namely the vertex numbers, vertex names, edge names and the extensions and returns the number of instances of each of the substructure. The UDF is a table UDF, which would return exactly the number of substructures in the Joined_1 table that is basically the number of instances of that substructure. The way it is done is that each of the tuple in the Joined_1 table is compared with every other tuple in the table to count the number of instances of that substructure in the graph.

4.3.1.1 *Table UDF's in DB2*

Table UDF's work in a different way than the scalar external UDF's. An example for creating a table UDF *isomorph_2()* is described below,

```
create function isomorph_2
```

```
(integer,varchar(20),integer,varchar(20),varchar(20),inte
ger,varchar(20),integer,varchar(20),varchar(20),integer)
        returns table ( instances integer)
        specific isomorph_2_apr18
        external name 'myudf!isomorph_2'
        language c
        parameter style DB2SQL
        variant
        not fenced
        scratchpad
        final call
        no SQL
        disallow parallel
        no external action
```

The main difference between the CREATE FUNCTION statement for an external scalar function and table function lies in the returns clause. In the case of a table function, it specifies a column name and data type for each of the columns of the table to be returned by the function. The isomorph_2 is a table function that integers and varchars as input. It returns a table, which has column instances of type integer. The clause final call is necessary for a table function. The disallow parallel is also necessary because a table function runs on a single node. The rest of the parameters are same as any normal external scalar function.

The isomorph_2 is implemented in the following way:

```
void SQL_API_FN isomorph_2
(
long *vertex1,
char *vertex1name,
long *vertex2,
char *vertex2name,
char *edge1,
long  *vertex3,
char *vertex3name,
long *vertex4,
char *vertex4name,
char *edge2,
long *count,
long *instances,
short *vertex1_ind,
short *vertex1name_ind,
        short *vertex2_ind,
        short *vertex2name_ind,
        short *edge1_ind,
        short *vertex3_ind,
        short *vertex3name_ind,
        short *vertex4_ind,
        short *vertex4name_ind,
        short *edge2_ind,
        short *count_ind,
        short *instances_ind,
        char *sqlstate,
        char *fnname,
        char *specificname,
        char *message,
        SQLUDF_SCRATCHPAD *scratchpad,
        SQLUDF_CALL_TYPE *calltype)

        {
        long *pad=(long *)scratchpad->data;

        switch(*calltype)
        {
                case SQL_TF_OPEN:
                *pad=0;
                break;
                case SQL_TF_FETCH:
                see_last++;
                cost=check(vertex1,vertex1name,vertex2,vertex2name,edge1,vertex3,vertex3name,vertex4
,vertex4name,edge2);
                        if(cost==0.00)
                        count_instances++;
                if(see_last<*(count))
                {
                strcpy(sqlstate,"02000");
                }
                else
                {
                        if(*pad<1)
                        {
                        (*pad)++;
                        *instances=count_instances;
                        strcpy(sqlstate,"00000");
                        return;
                        }

                strcpy(sqlstate,"02000");
                see_last=0;
                count_instances=0;
                }
                break;

        case SQL_TF_CLOSE:
        break;
        }
        }

        float check(long *vertex1,char *vertex1name,long *vertex2,char *vertex2name,char *edge1,long
*vertex3,char *vertex3name,long *vertex4,char
        *vertex4name,char *edge2)
        {
        graphptr g1,g2;
```

```
char *temp=(char *)malloc(3000*sizeof(char));
char *graph1=(char *)malloc(3000*sizeof(char));
char *graph2=(char *)malloc(3000*sizeof(char));
float f1=0;
NumLabels=0;
LabelList=NULL;
Directed=TRUE;
Threshold=1.0;
sprintf(temp,"v %d %s\n",*vertex1,vertex1name);
strcat(graph1,temp);
sprintf(temp,"v %d %s\n",*vertex2,vertex2name);
strcat(graph1,temp);
sprintf(temp,"d %d %d %s\n",*vertex1,*vertex2,edge1);
strcat(graph2,temp);
sprintf(temp,"v %d %s\n",*vertex3,vertex3name);
strcat(graph2,temp);
sprintf(temp,"v %d %s\n",*vertex4,vertex4name);
strcat(graph2,temp);
sprintf(temp,"d %d %d %s\n",*vertex3,*vertex4,edge2);
strcat(graph2,temp);
g1=read_graph(graph1);
g2=read_graph(graph2);
f1 = fm(g1,g2,max_node(g1->nv),0);
return f1;
}
```

The SQL query, which can be used to call the function, is explained below.

```
Select    t1.instances
From      joined_1  j1,joined_1 j2
          ,table(isomorph_2(j1.vertex1,j1.vertex1name,j1.verte
          x2,j1.vertex2name,j1.edge1,j2.vertex1,j2.vertex1name
          ,j2.vertex2,j2.vertex2name,j2.edge1,100)) as t1
```

This query returns the number of instances of each of the substructure in the table, which has 100 tuples in the table.

The isomporh_2 function takes as input all the information for comparing all the single edges substructures. The two vertex numbers, two vertex names and the edge name from each of the table are the input for the function. The *count* variable is the number of tuples in the table. *Instances* variable is for the output to be written to the table. The indicator variables are for checking if the input value is a null or not.

When a table function is invoked in an SQL statement, a series of calls is made to the C program that implements the table function. The first of these is the OPEN_CALL, with the final call indicator set to the value SQL_TF_OPEN. The OPEN call allows the table function to perform preliminary actions such as allocating memory, and initializing scratchpad. No data is returned by the OPEN call. Following the OPEN call, the system calls the table function with a series of FETCH calls, with the final call indicator set to SQL_TF_FETCH. On each of these calls, the table function is expected to return one of the rows to the result table. But by doing that the number of tuples returned to the table will be the cross product of the two tables. Since one tuple is to be returned for each of the tuple in the table, we don't return a tuple till we see *count* number of tuples. So till the *see_last* is less than *count* we copy the SQL state 02000 which will make the system call the function again for the same tuple. Once we have processed all the tuples of one table and compared that to one tuple of the other table the SQL state is set to 00000, which tells the system to return a tuple to the result table. For every call of the function the check() function is called to check for isomorphism and if the function returns a value 0 the count of that substructure is incremented by 1. Similar functions have been written for two edge and three edge substructures.

4.4 Experiment Results and Conclusion

The functions were tested for inputs of cardinality of 100 to 500. The function resulted in the correct output for cardinality until 500, but when tested for 500 the function resulted in an SQL0430N error. We have tried to fix this error in many ways. This is one area of difficulty when working with UDF's because there are no debugging tools provided by the database. Simple debugging tools like *printf* cannot be used because the UDF runs as background process where *stdout* does not have any meaning. When the function returns an error code of SQL0430N and it is run without fixing the error the system just hangs. So after testing out with trials, the function that seems to be causing the error was identified as the isomorphism function *fm().* The error being that memory allocated to some variables is not being freed properly. The function when called returns output for few tuples in the table and then terminates. We also came to a conclusion that dynamic memory allocation can be a problem in UDF's. The memory allocation has to be done in a specific way when using UDFs. All scratchpad memory needs to be allocated at the beginning and released at the end. Since that may not be happening in that exact way in the fm() function, the system returns errors.

The advantage of using UDF's are that they are much faster than the cursors, although not much testing could be done on UDF's but UDF's do not have the client/server overhead since they run in the database environment and work just like normal SQL queries. The disadvantages of UDF's are that they are difficult to code, they are non-standard, and there are no easy ways of debugging a UDF. All of the code written must be logically correct; otherwise there might be a chance of yielding incorrect values or sometimes hang the system. The logical flow of the system cannot be tested nor can the intermediate values of various parameters be checked.

CHAPTER 5

ENHANCED CURSOR-BASED APPROACH (ECBA)

This chapter identifies the major drawbacks in the cursor-based approach and how rewriting the SQL expressions and maintaining additional tables can overcome these drawbacks. This chapter also describes the intuition behind this approach, which involves the SQL-based queries to get the counts of the substructures without using the CURSORS. The chapter also describes how the graphs are compared without calling the isomorphism code.

5.1 Why a new Approach

The previous two approaches though provides correct functionality, do not provide an acceptable performance. The need for a better algorithm to get the counts of the substructures is critical for improving the performance. The UDFs though provide a better and faster way of attaining this functionality are very hard to code and test. So any new approach should try to obtain the counting of the substructures using an SQL query rather than relying on cursors to speed up the algorithm.

5.1.1 Graph Representation

Consider Figure 5-1 which shows how a substructure of 3 edges is stored in the database. The vertex numbers are represented as VNo, vertex names as VNa, edge names as ENa, and the extensions as Ex

V1No	V2No	V3No	V4No	V1na	V2Na	V3Na	V4Na	E1Na	E2Na	E3Na	Ex1	Ex2
1	2	3	4	A	B	C	D	AB	BC	DC	2	-3
5	6	7	8	A	B	C	D	AB	BC	DC	2	-3
13	14	18	19	E	F	G	H	EF	FG	GH	2	3

Figure 5-1 Joined_3 table containing all the 3 edge substructures

There are three tuples in the table of which the first two are isomorphic. The graph representations of these tuples are shown in Figure 5-2.

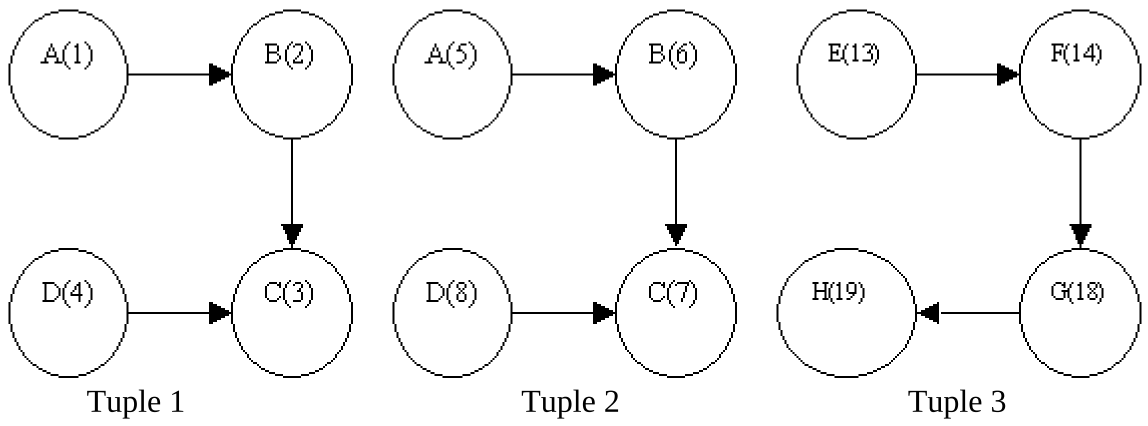


Figure 5-2 Graphs for the tuples in table

From the representation one can come to conclusion that the vertex numbers are not used for comparing graphs for isomorphism. The first and the second tuples are not only isomorphic but are exact graphs. So there is a way of counting the instances of the substructure if the graph match is exact instead of an inexact graph match using SQL instead of cursors. The vertex names, edge names and the extensions form a signature for the graph. But to attain this type of matching the extensions have to be taken care so that all the graphs are expanded in all possible ways unlike in the first approach where we expand to one substructure in only one possible way. The generalization is explained in section 5.3.4.1. Figure 5-3 shows how a single edge substructure is expanded to a two-edge substructure in the first approach and how it should have been done to do an exact graph match.

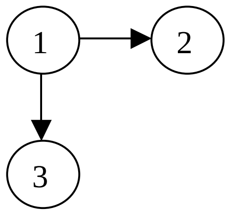


Figure 5-3 An example for graph extension

In Figure 5-3 all the single edge substructures would be 1->2 and 1->3. In the first approach only the substructure 1->2 will be expanded to two-edge substructure but not the other tuple since the second vertex number is lesser. So the way extensions are performed need to be changed in order to use SQL for updating the counts.

5.1.2 Graph Extension Revisited

The graphs are now extended in all possible ways irrespective of their vertex numbers, but care is taken to generalize the expansion. Consider the graph shown in Figure 5-4.

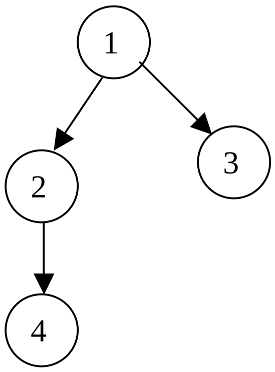


Figure 5-4 Graph extension example

For all the single edge graphs, the first vertex is treated as the root. The root is assigned a level 1 and the vertex to which it is connected is assigned a level 2. When the substructure is expanded a new edge and vertex are added. Depending on the vertex from which the edge was expanded the new vertex is assigned a level. If it were expanded on the first vertex then it has a level 2 else a level 3. The two edge substructures are 1->2->4 and 1->(2,3). Now for the substructure 1->(3,2) the new edge was added from a level 1 and for 1->2->4 the new edge was added from level 2. The next expansion is dependent on the previous level of expansion. If the previous level of expansion was 2 then the next expansion has to be on level 2 or more, so no expansion can take place at level 1. This expansion guarantees that all the substructures are discovered and does not generate duplicates. The generalized SQL query for expansion would look like:

```
Insert    into Joined_N
          Vertex1,Vertex2…..VertexN+1,Vertex1Name,Vertex2Name…
          ..VertexN+1Name,Edge1Name,Edge2Name…..EdgeNName,Ext1
          ,Ext2…..ExtN-1,0
     (
          Select
                    j1.Vertex1,j1.Vertex2….j1.vertexN,j2.verte
                    x2,j1.Vertex1Name,j1.Vertex2Name…j1.vertex
                    NName,j2.vertex2name,j1.edge1Name,j1.edge2
                    name….j1.EdgeNName,j2.edgename,j1.Ext1,j1.
                    Ext2…..j1.ExtN-2,p,0
          From      Joined_N-1 j1, Joined_base j2
          Where     j1.vertexP= j2.vertex1 and j1.vertexP+1 <
                    j2.vertex2….j1.VertexN<j2.Vertex2 and
                    j.extN-2<=p and j.extN-2>0
     )
```

The variable **P** in the where condition can vary from 1 to N, as the substructure can be expanded on any of the vertices. The condition **j.extN-2 <=P** satisfies the criterion of levels discussed above. The query is the same as the query used in the cursor based approach except that instead of Joined_1 table in the FROM statement there is Joined_base table. The reason for this would be explained in the next section. The query assumes that the edges are going out of a vertex, so to cover substructures where edges are coming in, the query used in the cursor-base approach is used.

5.2 Initialization of Data

The input is a file, which contains all the vertices and the edges in the graph. This information is loaded into the database as the Vertices and the Edges tables. The Vertices table will have the information about all the vertices, namely their vertex numbers and their labels. The edges table will have the information about the edges, namely the vertex numbers of the edge and the edge label. As we have already noted the vertex labels are needed for making any graph comparison. Since the edges table does not have the vertex labels, a new table is created which has all the single edge substructures. This table is named as the Joined_base table and the query to create this table is shown below:

```
Insert    into
          Joined_base(vertex1,vertex2,vertex1name,vertex2
          name,edge)
     (
     Select
               v1.vertexNo,v2.vertexNo,v1.vertexname,v2.v
               ertexName,e.edgename
     From      edges e, vertices v1, vertices v2
     Where     e.vertex1=v1.vertexNo and e.vertex2
               =v2.vertexNo
     )
```

This table forms the basis for the rest of the algorithm. Since the extension of substructures takes place by adding an edge to an existing substructure, the base table has to be used for extending. An example of the Joined_base table is shown in Figure 5-5

Vertex1	Vertex2	Vertex1Name	Vertex2Name	Edge
1	2	A	B	AB
1	3	A	C	AC
2	4	B	X	FOO
5	4	Z	X	BAR

Figure 5-5 Joined_base table

5.3 Algorithm

The idea behind developing the new algorithm is to use a different scheme for updating the counts of the substructures. The pseudo code for the algorithm is explained below:

```
1)    Subdue-DB(input file, size)
2)         Load vertices into vertices table;
3)         Load edges into edges table;
4)         Load joined_base table : join vertices and
           edges table
5)         WHILE(i<size)
6)              Load Joined_i table (substructures of size
                i)
7)              From beam_joined_i-1 , Joined_base
8)              Create    Frequent_i table
9)              DECLARE   Cursor c1 on Frequent_I order by
                count
10)             DECLARE   Cursor c2 on Frequent_i
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++
```

The main difference between this approach and the other approaches discussed is that SQL statements update the count of the substructures. Cursors are not used for updating the counts. With this approach, the concept of beam can also be implemented.

5.3.1 Flow of the algorithm

The algorithm starts with initializing the vertices and the edges table. The Joined_base table is loaded by making a join on the vertices and the edges table. The Joined_base table contains all the single edge substructures including the vertex labels, vertex number connecting the edges and the edge names. This table forms the base for any substructure expansions in the future.

The algorithm proceeds with finding all the single edge substructures and their counts. The Joined_1 table is loaded from the vertices and the edges table. The Joined_1 table contains the instances of all the single edge substructures. This table does not have the *count* attribute that maintains the number of instances of the substructure. The Frequent_1 table is created to store the substructures of size 1 and their counts. So the Frequent_1 table does not have the information of the vertex numbers. In order to get the counts of the substructures the Joined_1 table is projected on the attributes vertex labels and edge labels and *grouped by* the same attributes. By doing a group by we are creating a signature for each of the substructure and collecting the counts of the substructures that have the same signature. For example an edge A->B with edge name AB will have a signature AAB. Hence all the exact instances of this substructure are grouped as one tuple with their count updated.

Once the Frequent_1 table is created the concept of beam is implemented. The substructures are sorted on the attribute *count* and the best *beam* numbers of substructures are inserted into a table Frequent_beam_1. So only the substructures in the Frequent_beam_1 table will be expanded to larger substructures. But since the Frequent_beam_1 table does not maintain the attributes vertex numbers it cannot be used to expand to larger substructures. This Frequent_beam_1 table is joined with Joined_1 table, which has all the instances of single edge substructures to generate the instances of the substructures present in the table.

These tuples are loaded into the table Joined_beam_1. The Joined_beam_1 table can be joined with Joined_base table to generate the two edge substructures.

This halting condition for the algorithm would be when the size of the substructure reaches the user specified **max size**. Another halting condition would be when there are no substructures left in the table for expanding.

5.3.2 Discovering the single edge substructures

We will consider the graph shown in Figure 5-6 in explaining the algorithm.

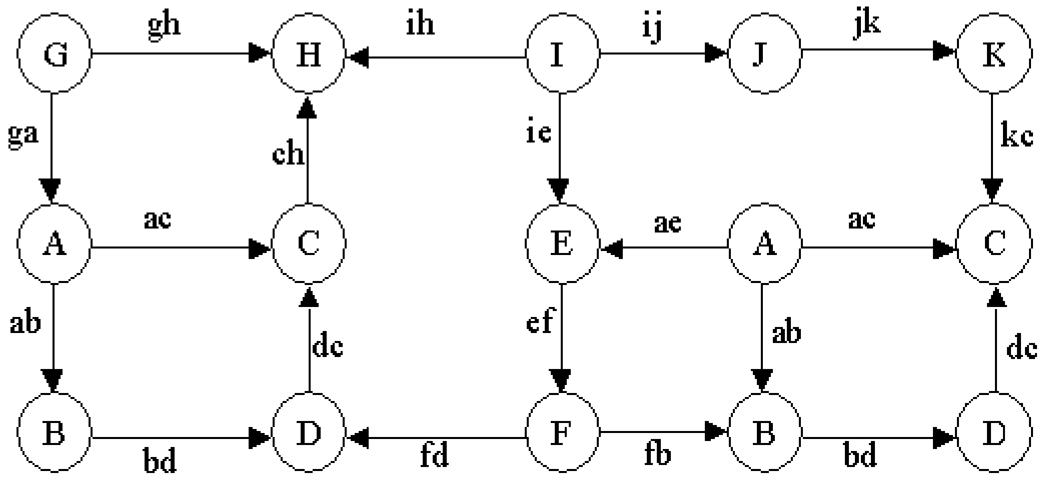
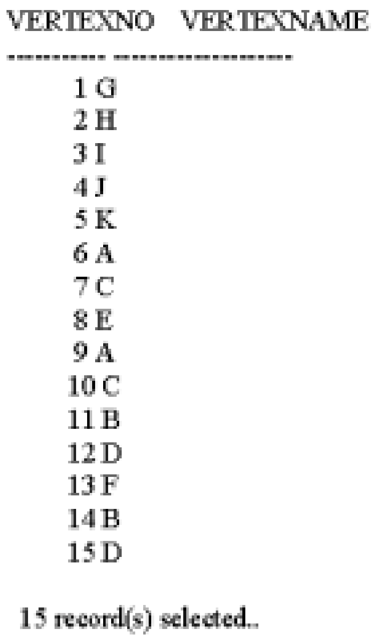


Figure 5-6 An example graph

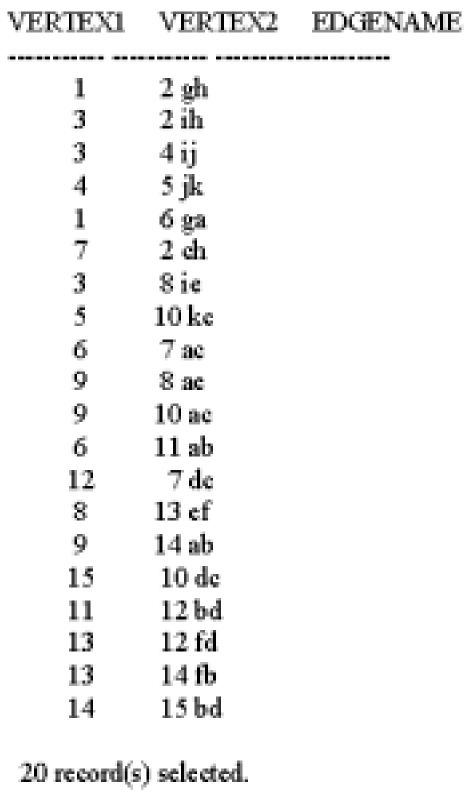
For the graph shown in the Figure 5-6, the vertices and the edges table are shown in the Figure 5-7 and 5-8 respectively.



```
VERTEXNO    VERTEXNAME
---------- --------------------
         1 G
         2 H
         3 I
         4 J
         5 K
         6 A
         7 C
         8 E
         9 A
        10 C
        11 B
        12 D
        13 F
        14 B
        15 D

 15 record(s) selected..
```

Figure 5-7 Vertices table

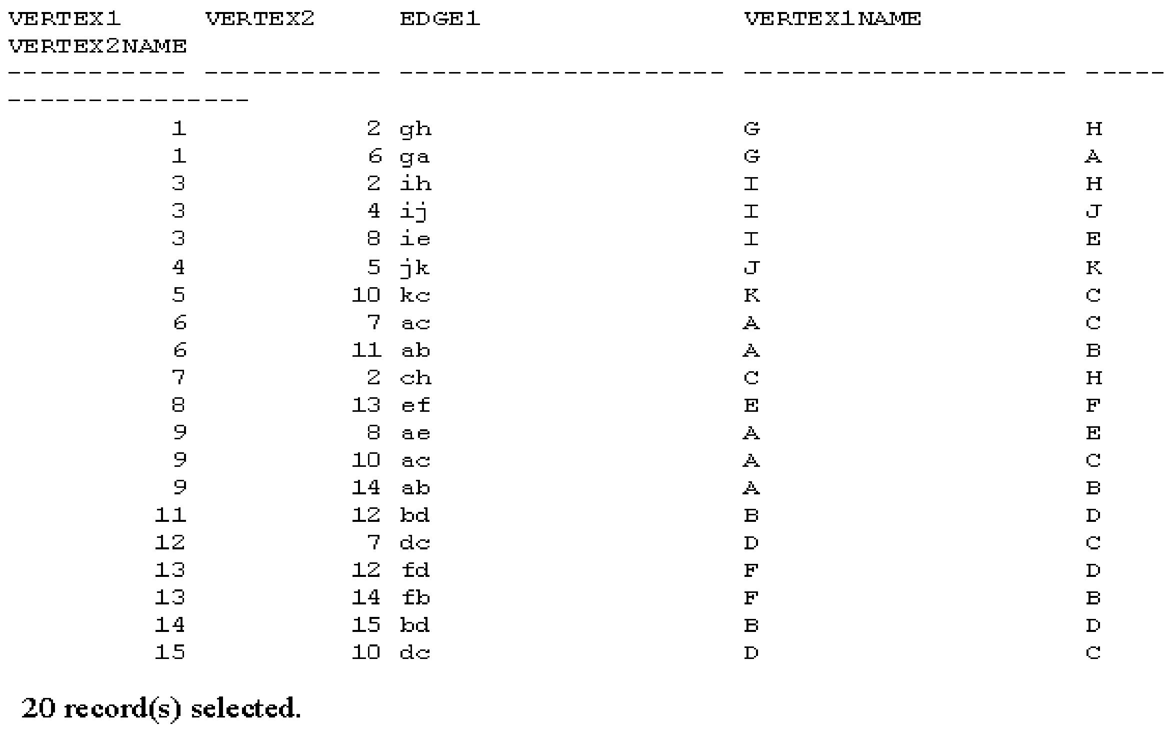


```
VERTEX1    VERTEX2    EDGENAME
---------- ---------- ------------------
      1          2 gh
      3          2 ih
      3          4 ij
      4          5 jk
      1          6 ga
      7          2 ch
      3          8 ie
      5         10 ke
      6          7 ac
      9          8 ae
      9         10 ae
      6         11 ab
     12          7 dc
      8         13 ef
      9         14 ab
     15         10 de
     11         12 bd
     13         12 fd
     13         14 fb
     14         15 bd

 20 record(s) selected.
```

Figure 5-8 Edges table

The Joined_base, which is created by joining the vertices and the edges table, is shown in the Figure 5-9. This table will form the base table for expanding the substructures.



```
VERTEX1     VERTEX2     EDGE1                VERTEX1NAME
VERTEX2NAME
----------- ----------- -------------------- -------------------- -----
---------------
          1           2 gh                   G                    H
          1           6 ga                   G                    A
          3           2 ih                   I                    H
          3           4 ij                   I                    J
          3           8 ie                   I                    E
          4           5 jk                   J                    K
          5          10 kc                   K                    C
          6           7 ac                   A                    C
          6          11 ab                   A                    B
          7           2 ch                   C                    H
          8          13 ef                   E                    F
          9           8 ae                   A                    E
          9          10 ac                   A                    C
          9          14 ab                   A                    B
         11          12 bd                   B                    D
         12           7 dc                   D                    C
         13          12 fd                   F                    D
         13          14 fb                   F                    B
         14          15 bd                   B                    D
         15          10 dc                   D                    C

 20 record(s) selected.
```

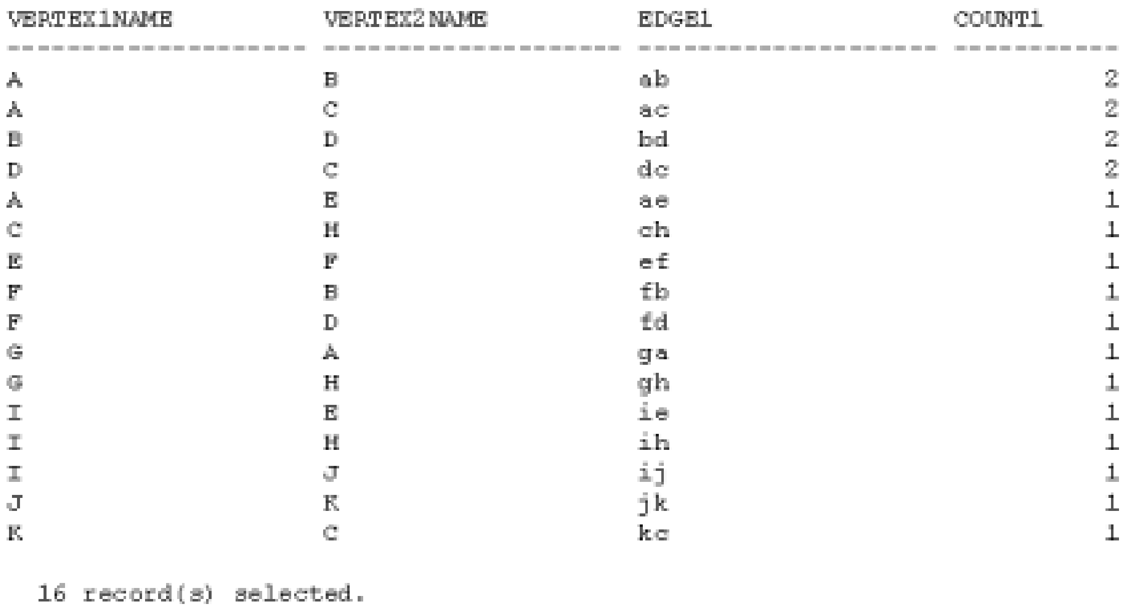
Figure 5-9 Joined_base table

The Joined_1 table that has all the single edge substructures is just a replica of the Joined_base table. So the table would be the same as shown in Figure 5-9. Using the query shown below we create the Frequent_1 table that has the substructures of size 1 and contains the counts of each individual substructure.

```
Insert    into Frequent_1 (Vertex1Name, Vertex2Name,
          EdgeName, count)
          Select
                    j.vertex1name,j.vertex2name,j.edgenam
                    e,count(*)
          From      Joined_1
```

```
Group     by
          j.vertex1name,j.vertex2name,j.edgenam
          e
```

The frequent_1 table has all the single edge substructures with their counts updated in the *count* attribute By grouping the edge substructures on their vertex names and edge name, we have essentially grouped the instances of each substructure and thus updating their counts. The Frequent_1 table created is shown in Figure 5-10.



```
VERTEX1NAME        VERTEX2NAME        EDGE1              COUNT1
------------------ ------------------ ------------------ ----------
A                  B                  ab                          2
A                  C                  ac                          2
B                  D                  bd                          2
D                  C                  dc                          2
A                  E                  ae                          1
C                  H                  ch                          1
E                  F                  ef                          1
F                  B                  fb                          1
F                  D                  fd                          1
G                  A                  ga                          1
G                  H                  gh                          1
I                  E                  ie                          1
I                  H                  ih                          1
I                  J                  ij                          1
J                  K                  jk                          1
K                  C                  kc                          1

  16 record(s) selected.
```

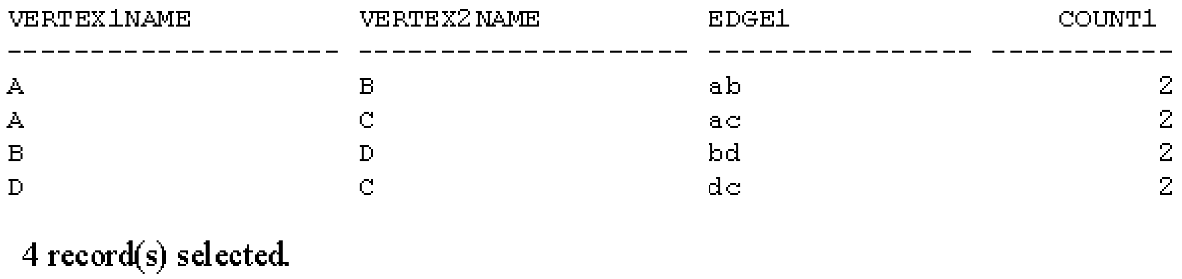
Figure 5-10 Frequent_1 table

The next step in the algorithm would be deleting all the single instance substructures from the Frequent_1 table. These instances are also deleted from the Joined_base table. Since the Joined_base table will be used for expansions to larger substructures, we do not want to expand a substructure by an edge, which has only one instance. The updated Frequent_1 table and the Joined_base table are shown in Figure 5-11 and Figure 5-12 respectively. The query to delete tuples from Joined_base, which have single instance, is shown below:

```
Exec  sql  delete
           From joined_base j
           where(j.vertex1name, j.vertex2name, j.edge1) in
                (select   t.vertex1name, t.vertex2name,
                          t.edge1
                 From     Frequent_1 t
                 Where    t.count1=1)
```

The query to delete tuples of single instance from frequent_1 is shown below:

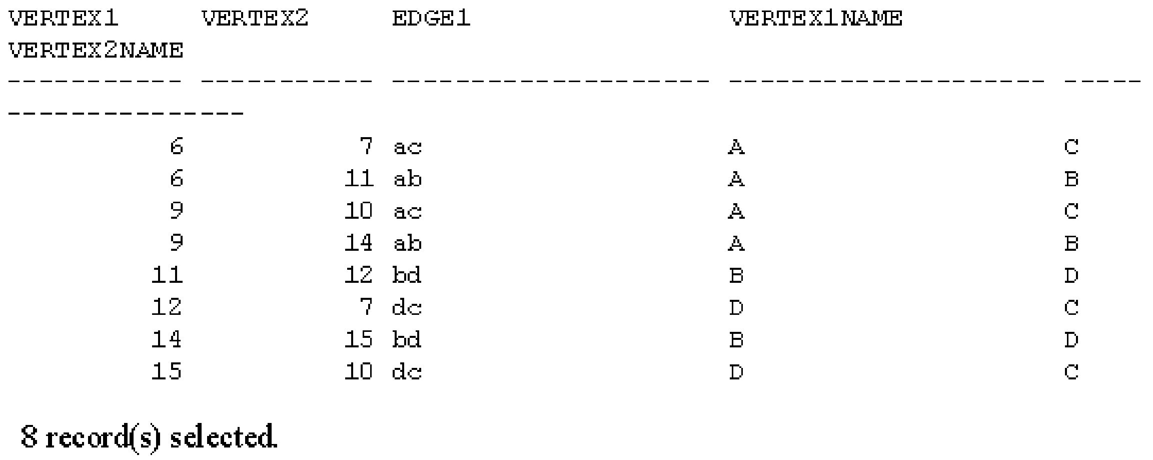
```
Exec sql
     Delete    From Frequent_1
               Where count1=1
```



```
VERTEX1NAME          VERTEX2NAME          EDGE1                 COUNT1
-------------------- -------------------- ---------------- -----------
A                    B                    ab                         2
A                    C                    ac                         2
B                    D                    bd                         2
D                    C                    dc                         2

 4 record(s) selected.
```

Figure 5-11 Updated Frequent_1 table



```
VERTEX1     VERTEX2     EDGE1                VERTEX1NAME
VERTEX2NAME
----------- ----------- -------------------- -------------------- -----
---------------
          6           7 ac                   A                    C
          6          11 ab                   A                    B
          9          10 ac                   A                    C
          9          14 ab                   A                    B
         11          12 bd                   B                    D
         12           7 dc                   D                    C
         14          15 bd                   B                    D
         15          10 dc                   D                    C

 8 record(s) selected.
```

Figure 5-12 Updated Joined_base table

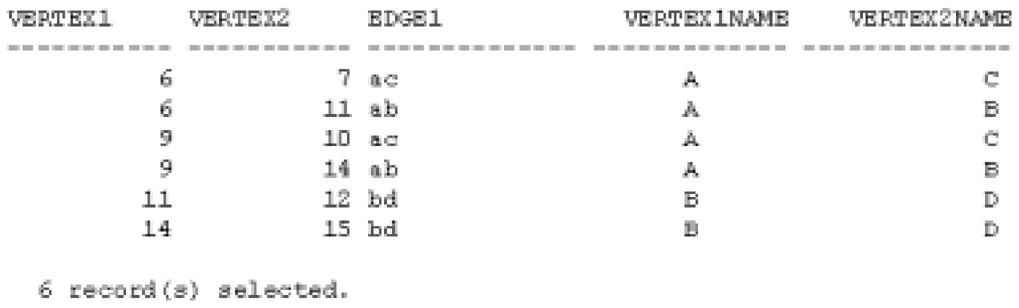
5.3.2.1 Implementing Beam for single-edge substructures

In order to implement the concept of *beam,* the number of substructures that will be extended to two edge substructures should be restricted to the *beam* size. The Frequent_1 table has all the substructures of size 1. So of these substructures beam number of substructures is to be selected for future expansions. So the substructures are sorted on the attribute count. By choosing the count attribute for sorting we are essentially implementing the compression based on *size* compared to MDL. Since the compression achieved by the substructure is directly proportional to the number of instances of the substructure, and we are dealing with substructures of same size we sort it base on the count attribute. The cursors are used to insert the beam number of tuples from Frequent_1 table into Frequent_1_beam table. The tuples are inserted in the descending order based on their counts. So if the beam were only three then the first three tuples from the Frequent_1 table are inserted into the Frequent_1_beam table.

The Frequent_beam_1 table has beam number of tuples in the table. These substructures form the best 3 substructures to compress the graph of those size substructures. At this stage the algorithm finishes processing all the single edge substructures. In order to expand the single edge substructures to two edge substructures, the tuples in the Frequent_beam_1 table cannot be used because they do not have an attribute for the vertex numbers. So in order to extend to a two-edge substructure the instances of the substructures in the Frequent_beam_1 table have to be gathered. The Joined_1 table has the instances of not only the substructures in the Frequent_beam_1 table but also the instances of all the single edge substructures. So by making a join with the Joined_1 table with the requient_beam_1 table we can gather all the instances of the substructures in the Frequent_beam_1 table, which can be used to expand to two-edge substructures. The so gathered substructures are loaded into new table Joined_beam_1 table. The query to load tuples into Joined_beam_1 is shown below:

```
Exec SQL  insert    into
                    Joined_beam_1(vertex1, vertex2,
                              Vertex1name, vertex2name, edge1)
          (
               Select    j.vertex1, j.vertex2, j.vertex1name,
                         j.vertex2name, j.edge1
               From      joined_1 j, frequent_beam_1 f
               Where     f.vertex1name = j.vertex1name and
                         j.vertex2name = f.vertex2name and
                         j.edge1 = f.edge1
          )
```

The resulting Joined_beam_1 table is shown in Figure 5-13. Only the tuples in this table will participate in the higher extensions. These tuples represent the instances of the substructures in the Frequent_beam_1 table.



```
VERTEX1     VERTEX2     EDGE1           VERTEX1NAME    VERTEX2NAME
----------- ----------- -------------- -------------- -------------
          6           7 ac                          A             C
          6          11 ab                          A             B
          9          10 ac                          A             C
          9          14 ab                          A             B
         11          12 bd                          B             D
         14          15 bd                          B             D

  6 record(s) selected.
```

Figure 5-13 Joined_beam_1 table

5.3.3 Extending to two-edge substructures

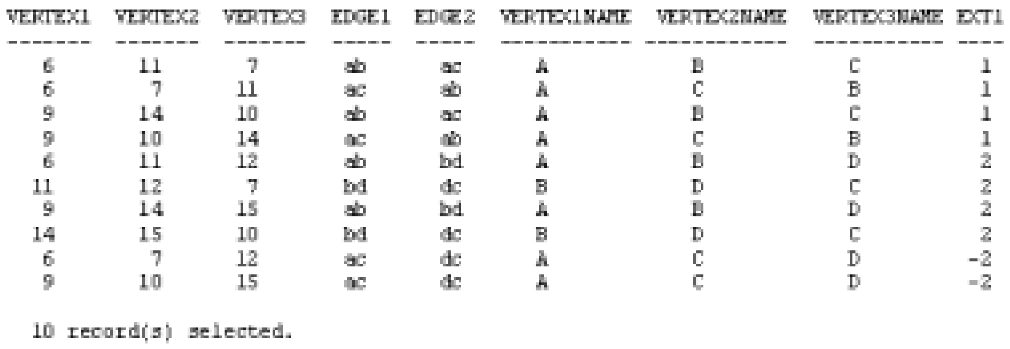
The Joined_beam_1 table contains all the instances of the single edge substructure as shown in Figure 5-13. The single edge substructure can be expanded to a two-edge substructure on any of the two vertices in the edge. All the possible single edge substructures are listed in the Joined_base table. So by making a join with the Joined_base table 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 vertex in the Joined_base table. The following query extends a single edge substructure in all possible ways,

```
Insert    into
          Joined_2(Vertex1,Vertex2,Vertex3,Vertex1Na
                    me,Vertex2Name,Vertex3Name,
                    Edge1Name,Edge2Name,Ext1)
          (
               Select
                         j1.vertex1,j1.vertex2,j2.vertex
                         2,j1.vertex1name,j1.vertex2name,
                         j2.vertex2name,j1.edge1name,j2.e
                         dge1name,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,j
                         2.vertex2name,j1.edge1name,j2.ed
                         ge1name,2
               From      Joined_1 j1, Joined_base j2
               Where     j1.vertex12=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
```

)

This above query has three sub queries within it. The first query extends the substructure by adding an edge going out of the first vertex if any. The second sub query extends the substructures by adding an edge going out of the second vertex. The third sub query extends the substructure by adding an edge coming into the second vertex. These queries take care of generating all the possible two edge substructures. All the resulting two edge substructures are stored in the Joined_2 table.

The resulting table Joined_2 is shown in Figure 5-14.



```
VERTEX1  VERTEX2  VERTEX3  EDGE1  EDGE2  VERTEX1NAME  VERTEX2NAME  VERTEX3NAME EXT1
-------  -------  -------  -----  -----  ----------- ------------  ----------- ----
   6       11        7      ab     ac       A            B            C          1
   6        7       11      ac     ab       A            C            B          1
   9       14       10      ab     ac       A            B            C          1
   9       10       14      ac     ab       A            C            B          1
   6       11       12      ab     bd       A            B            D          2
  11       12        7      bd     dc       B            D            C          2
   9       14       15      ab     bd       A            B            D          2
  14       15       10      bd     dc       B            D            C          2
   6        7       12      ac     dc       A            C            D         -2
   9       10       15      ac     dc       A            C            D         -2

  10 record(s) selected.
```

Figure 5-14 Joined_2 table

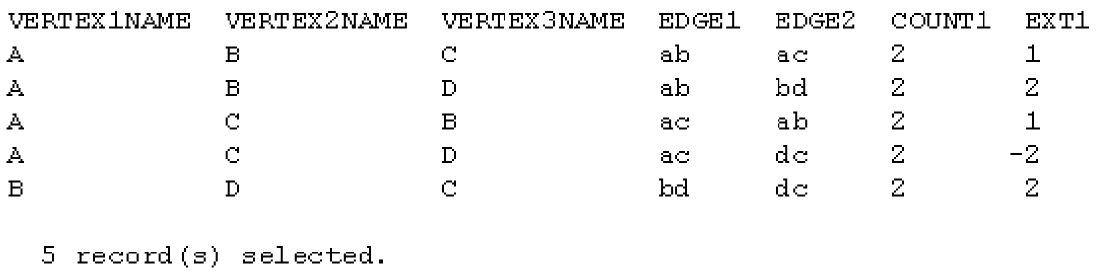
The attributes needed to store all the information about the substructure has increased from a single edge substructure. Apart from the edge name, vertex names newly added, in order to describe the substructure an extra attribute Ext1 is used. The Ext1 attribute describes how the new edge was added to the existing substructure. For example, consider the first tuple in the table Joined_2 table. The Ext1 is 1, meaning the new edge was added on the first vertex and the direction of the edge is going out of the first vertex. In the last tuple the Ext1 is –2 indicating that the new edge was added on the second vertex and the direction is coming into the vertex.

5.3.3.1 Updating the counts for two-edge substructures

All the two-edge substructures for consideration are listed in the table Joined_2. In order to update the counts of each substructure, the instances of each substructure have to be grouped into one single substructure. This can be achieved by using the *group by* statement in the SQL. But care should be taken not to *group by* instances of a substructure with instances of another substructure. The following query describes how the instances of the substructure can be grouped by.

```
Insert    into
          Frequent_2(vertex1name,vertex2name,
                     vertex3name,edge1,edge2,ext1,
                     count1)
                     (
                     Select   vertex1name, vertex2name,
                              vertex3name, edge1, edge2,
                              ext1, count (*)
                     From     Joined_2
                     Group    by
                                   vertex1name,vertex2nam
                                   e, vertex3name,
                                   edge1,  edge2,  ext1 )
                    )
```

The substructures are grouped by the vertex names, edge names and the ext1. By grouping by the vertex names and the edge names we have created a signature for the substructure. By including the attribute ext1, we are grouping all the substructures, which were expanded in an exact way and have the same labels. This comparison we are doing is an exact graph match. The generalization for any graph is explained in the section 5.3.4.1. The resulting Frequent_2 table is shown in Figure 5-15. Each tuple represents a substructure and the *count1* attribute represents the number of instances of the substructure. The *count1* represents the number of instances of that substructure



```
VERTEX1NAME  VERTEX2NAME  VERTEX3NAME  EDGE1  EDGE2  COUNT1  EXT1
A            B            C            ab     ac     2        1
A            B            D            ab     bd     2        2
A            C            B            ac     ab     2        1
A            C            D            ac     dc     2       -2
B            D            C            bd     dc     2        2

  5 record(s) selected.
```

Figure 5-15 Frequent_2 table

5.3.3.2 Implementing beam for two edge substructures

The Frequent_2 table consists of all of the two edge substructures with their number of instances updated. For implementing the beam concept, we need to restrict the number of substructures being expanded to beam. SQL does not provide a functionality of selecting some **X** number of tuples from a table. The selection is always done on a condition on an attribute value. For this reason if we have to select the first three tuples from a table we make use of the cursors. The cursors fetch tuples from a table in an order and hence we can restrict to fetching the required number of tuples.

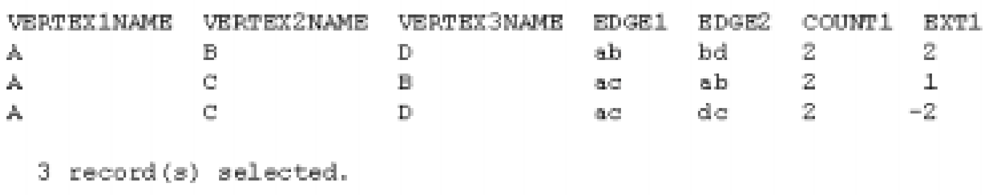
In the Frequent_2 table shown in Figure 5-15 the first and third tuples are actually the same substructures. But since they were expanded in a different way they are not considered the instances of the same substructure. They are treated, as two different substructures. The explanation for a generalized version is explained in the section 5.3.4.1. So Care should be taken while inserting the tuples from the Frequent_2 table into a new table Frequent_beam_2.

The same substructures should not be inserted into the Frequent_beam_2 table because these will be the substructures that will be expanded in the future expansions.

```
     Declare   graph1_cursor cursor for
               Select
                         vertex1name,vertex2name,vertex3name,e
                         dge1,edge2,ext1,count1
               From      Frequent_2
               Order     by   Count1    desc
```

A similar cursor is declared on the Frequent_2 table. The first tuple is inserted into the Frequent_beam_2 table. When the second substructure is fetched from the table it will be compared with the tuples already present in the Frequent_beam_2 table. If there are any substructures, which is the same as this substructure then that substructure will not be added to the Frequent_beam_2 table. By doing this we are restricting the substructures in the Frequent_beam_2 table to be distinct and not the same substructure.

. The Frequent_beam_2 table thus created is shown in Figure 5-16.



```
VERTEX1NAME  VERTEX2NAME  VERTEX3NAME  EDGE1  EDGE2  COUNT1  EXT1
A            B            D            ab     bd     2       2
A            C            B            ac     ab     2       1
A            C            D            ac     dc     2      -2

  3 record(s) selected.
```

Figure 5-16 Frequent_beam_2 table

5.3.3.3 *Creating the Joined_beam_2 table*

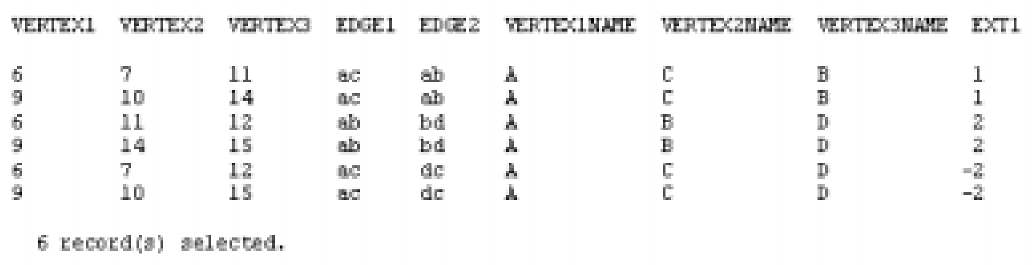
In order to expand to a three-edge substructure from a two-edge substructure we have to create a table for storing the instances of the substructures that are going to get expanded.

So the instances of the beam number of substructures are loaded into a new table Joined_beam_2 table. The query to achieve this functionality is shown below.

.

```
Insert    into
          Joined_beam_2(vertex1name,vertex1,vertex2name
                        ,vertex2,vertex3name,vertex3,edg
                        e1, edge2,ext1)
          (
               Select   j.vertex1name,j.vertex1,
                        j.vertex2name,j.vertex2,
                        j.vertex3name,j.vertex3,j.edge1,
                        j.edge2, j.ext1
               From     Frequent_beam_2 f,Joined_2 j
               Where    f.vertex1name=j.vertex1name and
                        f.vertex2name=j.vertex2name and
                        f.vertex3name=j.vertex3name and
                        f.edge1=j.edge1 and
                        f.edge2=j.edge2 and
                        f.ext1=j.ext1
               )
```

The resulting Joined_beam_2 is shown in the Figure 5-17.



```
VERTEX1  VERTEX2  VERTEX3  EDGE1  EDGE2  VERTEX1NAME  VERTEX2NAME  VERTEX3NAME  EXT1

6        7        11       ac     ab     A            C            B               1
9        10       14       ac     ab     A            C            B               1
6        11       12       ab     bd     A            B            D               2
9        14       15       ab     bd     A            B            D               2
6        7        12       ac     dc     A            C            D              -2
9        10       15       ac     dc     A            C            D              -2

  6 record(s) selected.
```

Figure 5-17 Joined_beam_2 table

5.3.4 <u>Generalization</u>

In this section the generalization for the larger substructures is explained. The **n-1** edge substructures are stored in the Joined_beam_**n**-1 table. In order to expand to n edge substructures an edge is added to the existing **n**-1 edge substructure. So the Joined_beam_**n**-1 table is joined with the Joined_base table to add an edge to the substructure. The query to generate the Joined_n table containing all the instances of an **n**-edge substructure is shown below:

```
Insert    into Joined_N
          Vertex1,Vertex2…..VertexN+1,Vertex1Name,Vertex2
          Name…..VertexN+1Name
          Edge1Name,Edge2Name…..EdgeNName,Ext1,Ext2…..Ext
          N-1,0
          (
          Select
                    j1.Vertex1,j1.Vertex2….j1.vertexN,j2.
                    vertex2,j1.Vertex1Name,j1.Vertex2Name
                    …j1.vertexNName,j2.vertex2name,j1.edg
                    e1Name,j1.edge2name….j1.EdgeN-
                    1Name,j2.edgename,j1.Ext1,j1.Ext2…..j
                    1.ExtN-2,-p,0
          From      Joined_N-1 j1, Joined_1 j2
          Where     j1.vertexP= j2.vertex2 and j1.vertex1
                    < j2.vertex2….j1.VertexP-1<j2.Vertex2
          )
```

P is a variable from 2 to N

A Joined_**n** table will have the following attributes to describe the substructure

- N+1 vertex numbers, describing the various vertex numbers
- N+1 vertex labels, describing the vertex labels
- N edges labels, describing the edge labels
- N-1 extension number, describing which vertexes connecting that edge

5.3.4.1 Generating the Frequent_n table

The Joined_n contains only the instances of the n edge substructure. In order to group the substructures, which are exact we create a table Frequent_n which would group all the exact graphs and update their counts. The query to creating the Frequent_n table is shown below:

```
Insert    into
          Frequent_n
                    (Vertex1name,Vertex2name…..VertexN+1n
                    ame,Edge1,Edge2,…….EdgeN,Ext1,Ext2….E
                    xtN-1)
          (


          Select
                    j.vertex1name,j.vertex2name…j.vertexN
                    +1name,j.edge1,j.edge2..j.edgeN,j.ext
                    1,j.ext2..j.extN-1
          From      Joined_N-1 j
          Group by  j.vertex1name,j.vertex2name…j.vertexN
                    +1name,j.edge1,j.edge2..j.edgeN,j.ext
                    1,j.ext2..j.extN-1

     )
```

By grouping the vertex names and edge names we have grouped all the labels of the substructure. But for two substructures to be exact the graphs have to be expanded in the exact way. Consider the graph shown in the Figure 5-18. The two graphs are exactly the same assuming that they have the same edge names. Let us assume these are two occurrences of the same substructures in the graph.

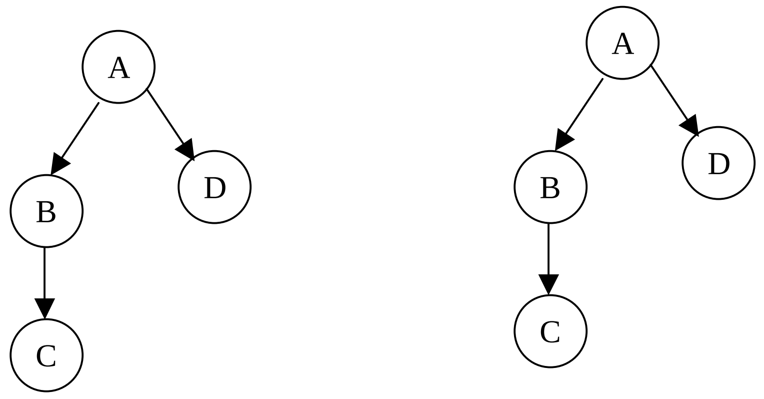


Figure 5-18 Exact graph match

The expansion starts from a single edge in our algorithm and proceeds with adding an edge to the existing substructure. Let us represent the expansion as ABCABBC2, meaning the vertex names are A, B, C and the edge AB was expanded to edge BC on the second vertex B. This will form the signature for the substructure. So the graphs shown in the Figure 5-18 could have been expanded to in many ways. The following are some possibilities

ABDCABADBC12

ADBCABADBC13

ABDCABADBC12

This means the different vertex labels in the graph are A, B, D, and C and the edges in the substructure are AB, AD, and BC. The extensions tell how the graph looks like, the first edge is from vertex1 to vertex2 and then 1 means the second edge is from vertex1 to vertex3 and the number 2 means the third edge is from vertex2 to vertex4. The instances of the substructures that have the exact signature are grouped together. So although the substructures might be theoretically the same, they are only grouped together if they were expanded in the exact same fashion starting from the first vertex. So functionality wise we are grouping the exact instances of the substructure to get their counts.

5.3.4.2 Implementing the beam for ***n****-edge substructures*

Once the Frequent_n table is loaded, we have all the substructures of size n with their counts updated. In order to implement the beam we create a table Frequent_beam_n, which will contain exactly beam number of substructures of size n. A cursor is declared on the Frequent_n table to sort them on the attribute count. The cursor fetches a tuple from the Frequent_n table and inserts that tuple into the Frequent_beam_n table. After inserting the first tuple into Frequent_beam_n table another tuple is fetched from Frequent_n table. This substructure is compared with the already inserted substructure in the Frequent_beam_n table and if they are found to be exact then this substructure is not inserted into the table. The comparison is done using the isomorphism code. This would continue till we insert exactly beam number of tuples into Frequent_beam_n table

The Frequent_beam_n table contains only the substructures of size n. In order to extend to n+1 edge substructures the vertex numbers of the substructures are needed. Since the Frequent_beam_n table does not contain the vertex numbers of the substructures the instances of these substructures have to be collected. So we create a new table Joined_beam_n to store the instances of these substructures. The Joined_n table contains all the instances of the n edge substructures. So by joining the Frequent_beam_n table and the Joined_n table we can get the instances of the substructures in the Frequent_beam_n table. The query for loading the Joined_beam_n is shown below:

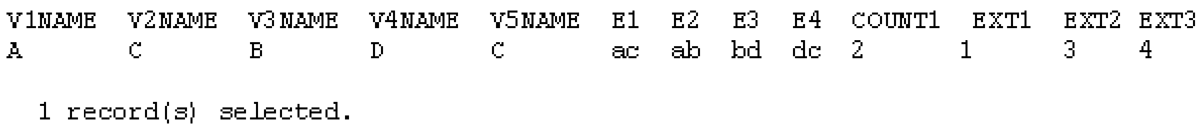
```
Insert    into
```

```
Joined_beam_n (ATTRIBUTES)
(
     Select    j.vertex1name … j.vertexN+1name,
               j.vertex1..j.vertexN+1,
               j.edge1…j.edgeN,
               j.ext1..j.extN-1
     From      Frequent_beam_N f,Joined_N j
     Where     f.vertex1name=j.vertex1name…
               f.vertexN+1name=j.vertexN+1name
               and f.edge1= j.edge1…
               f.edgeN=j.edgeN and
               f.ext1=j.ext1…
               and  f.extN-1=j.extN-1
     )
```

5.3.5 Halting conditions

There are two halting conditions for the algorithm. One of the user specified parameter is the **max size.** Once the algorithm discovers all the substructures of the **max size** the program terminates. Another halting condition would be when there are no tuples in the Joined_n table. This necessarily means that there are no more substructures discovered of size n.

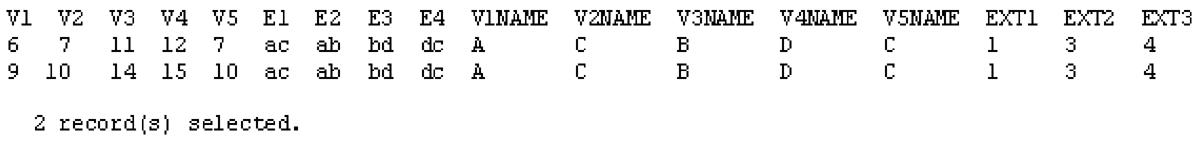
For example if the **max size** for the graph shown in Figure 5-6 is specified as 4, the algorithm terminates after discovering the 4 edge substructures. The Frequent_beam_4 table is shown in the Figure 5-19 for this graph. The algorithm has correctly discovered the substructure we have embedded. The instances of this substructure are shown in the Figure 5-20. This is also the table Joined_beam_4



```
V1NAME  V2NAME  V3NAME  V4NAME  V5NAME  E1  E2  E3  E4  COUNT1  EXT1  EXT2  EXT3
A       C       B       D       C       ac  ab  bd  dc  2       1     3     4

  1 record(s) selected.
```

Figure 5-19 Frequent_beam_4 table



```
V1  V2  V3  V4  V5  E1  E2  E3  E4  V1NAME  V2NAME  V3NAME  V4NAME  V5NAME  EXT1  EXT2  EXT3
6    7  11  12   7  ac  ab  bd  dc  A       C       B       D       C       1     3     4
9   10  14  15  10  ac  ab  bd  dc  A       C       B       D       C       1     3     4

  2 record(s) selected.
```

Figure 5-20 Joined_beam_4 table

5.3.6 Limitations to the algorithm

Some of the limitations to the current algorithm are discussed in this section.

1. Number of columns: There is a limitation to the number of columns a table can have in the database DB2. The maximum number of columns we can have in the system is only 500. Joined_n table would need 4n+1 attributes for describing the 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. So only the algorithm could discover a 124-edge substructure.
2. Cursors: We use the cursors for implementing the beam. The host variables are declared for exchanging data between the database and the host programming language using the cursors. DB2 does not support declaring array of host variables and hence the generalization for the algorithm was not

achieved. Separate host variable had to be declared for each pass. Right now the algorithm works for a maximum size of 6.

CHAPTER 6

PERFORMANCE ANALYSIS AND OPTIMIZATIONS

This chapter assesses the various approaches taken before we arrived at the final approach discussed in the previous chapter. The various SQL queries involved in the discovery process that have an affect on the run time are discussed. This chapter also discusses how the various SQL statements affect the performance and how they were optimized to achieve better performance.

The previous chapter discussed how the beam was implemented using the Frequent_beam_n table. The Joined_beam_n contains all the instances of the substructures in the Frequent_beam_n table. Although this was the final approach discussed, there were many other SQL alternatives that were explored in implementing this. The various approaches that were taken in implementing this functionality are discussed in this chapter.

6.1 Configuration File

The configuration file is useful for automating the 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 used for running the algorithm on several datasets with varying configurations without any user intervention. The variables defined in the configuration file are:

DBMS Type$User Name$Password$Table Name$MaxSize$Beam$Approach Number$LogFile$Debug (value 0 or 1)$Log Results to file (value 0 or 1)$Level of logging

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

Database Name: The database that contains your input relation.

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.

Approach Number: The approach number to be used for the algorithm. It is an integer value. All the approaches and their optimizations are given a unique integer value to identify them.

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

Beam: An integer to specify the beam.

Debug: If true, then prints the debug statements.

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

Log level: An integer number to specify the level of logging needed.

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 DB2. Approach -ECBA

DB2$subduedb$graphmining$graphmining$T5KV10KE$4$5$10$false$T5KV10KES4A4B10 .txt$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: DB2

Database Name: Subduedb

UserID: graphmining

Password: graphmining

Input Table: T5K10KE.

Approach Number: 4 (For ECBA)

Max Size: 5

Beam: 10

Debug: False (don’t print debug statements)

Log file Name: T5KV10KES4A4B10.txt

Log level: 1 (write the overall time taken).

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 that size.

6.3 Use of Correlated queries

The first approach taken to implement the beam was using a correlated query. The Frequent_n table contains all the substructures of size n. The Frequent_beam_n table contains all the substructures that have to be expanded for generating the n+1-edge substructures. The Joined_n table contains all the instances of the n-edge substructures. If the beam number is set to **P** and if there are **K** tuples in the Frequent_n table then the **K-P** substructures do not participate in generating the n+1-edge substructures. Hence if the instances of the tuples, which are not going to participate in generating the n+1-edge substructures, are deleted from the Joined_n table then we are left with only the instances of the substructures, which will be expanded to n+1-edge substructures. Hence the tuples from the Frequent_n table that do not form the beam are inserted into a new table EdgeN_subs. By removing the instances of these substructures from the Joined_n table, we attain the functionality needed. This query for doing this is shown below

```
Delete    from joined_N as j
     where Exists
     (
          Select    f.vertex1name
          From      edgeN_subs     f
          Where     f.vertex1name=j.vertex1name and
                    f.vertex2name=j.vertex2name…
                    f.vertexN+1name=j.vertexN+1name and
                    f.edge1=j.edge1 and  f.edge2=j.edge2…
                    f.edgeN=j.edgeN and  f.ext1=j.ext1
                    …f.extN-1=j.extN-1
          )
```

The above query is a correlated query because the sub-query contains an identifier that represents a row of the outer query. In the above query the identifier j (Joined_n table) is from the outer query.

6.3.1 Input data set generation for testing

The graph generator used for testing some of the results had been developed by Dr. Holder [9]. The generator accepts many parameters before it constructs the graph. Some of them are listed below.

- Number of vertices in the graph.
- Number of edges in the graph.
- Number of Vertex labels. The vertex labels can be given a probability with which they appear in the graph. The sum of the probabilities must add up to 1.
- Number of edge labels. This is similar to the vertex labels.
- Connectivity. Connectivity is the number of external connections on each instance of the substructure.
- Coverage. Coverage is the percentage of the final graph to be covered by the instances of the substructures.
- Overlap. Overlap is the percentage of the total instances that overlap.
- Substructure definition to be embedded in the graph. The different vertices in the substructure followed by the edges.

Graphs have been constructed with number of edges twice as many as the vertices in the graph. The overlap was set to 0.0 because the database algorithm does not implement the concept of overlap. The connectivity was set to 5. Ten different vertex labels and edge labels have been included each with a probability of 0.1. The coverage was set to 0.2. The graphs have been created to test the scalability and the performance of the algorithm. The embedded substructure is shown in the Figure 6-1.

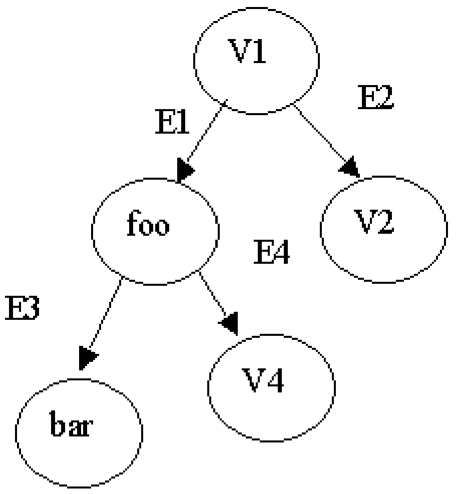


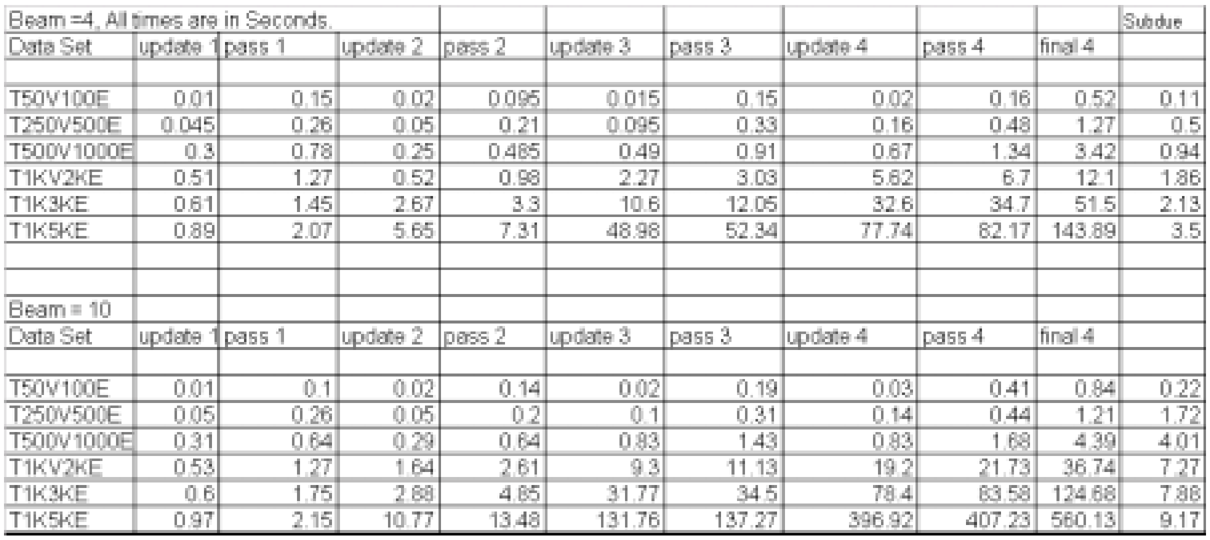
Figure 6-1 Embedded substructure

Table 6-1 Parameter Settings

Parameters	Subdue	Database
Size	4	4
Beam	4,10	4,10
Overlap	True	True

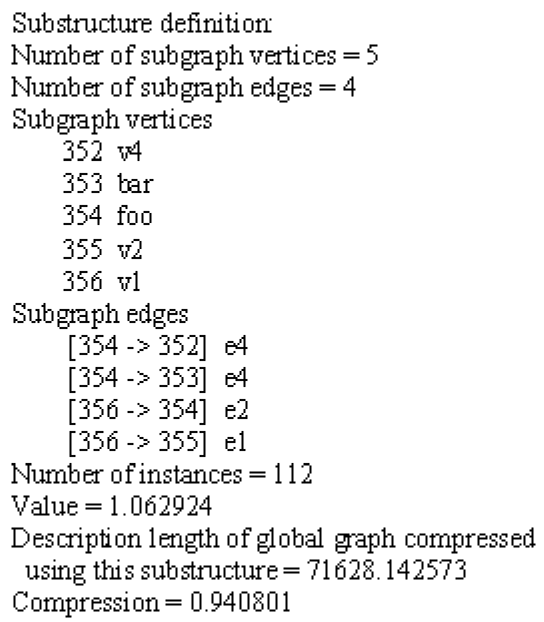
Table 6-1 shows the various parameters on which the Subdue algorithm and the database algorithm were run. Overlap was set to true in the main memory approach because the database approach currently does not support the non-overlap option. The run times for the algorithms are shown in table 6-2.

Table 6-2 Timings using the correlated query



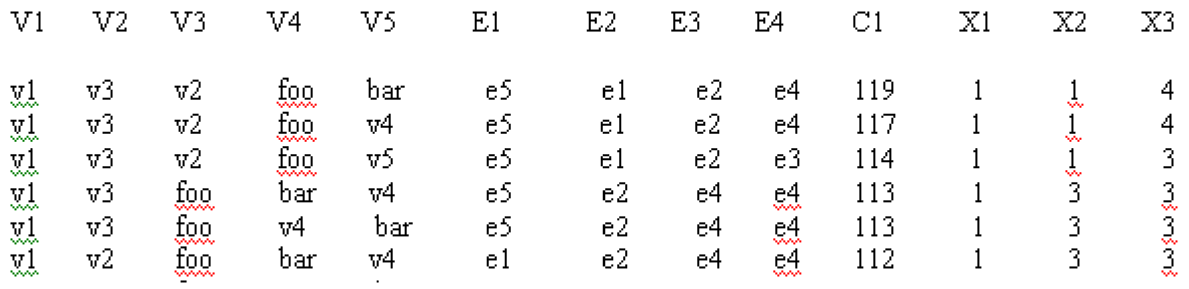
Beam =4, All times are in Seconds.										Subdue
Data Set	update 1	pass 1	update 2	pass 2	update 3	pass 3	update 4	pass 4	final 4	
T50V100E	0.01	0.15	0.02	0.095	0.015	0.15	0.02	0.16	0.52	0.11
T250V500E	0.045	0.26	0.05	0.21	0.095	0.33	0.16	0.48	1.27	0.5
T500V1000E	0.3	0.78	0.25	0.485	0.49	0.91	0.67	1.34	3.42	0.94
T1KV2KE	0.51	1.27	0.52	0.98	2.27	3.03	5.62	6.7	12.1	1.86
T1K3KE	0.61	1.45	2.67	3.3	10.6	12.05	32.6	34.7	51.5	2.13
T1K5KE	0.89	2.07	5.65	7.31	48.98	52.34	77.74	82.17	143.89	3.5
Beam = 10										
Data Set	update 1	pass 1	update 2	pass 2	update 3	pass 3	update 4	pass 4	final 4	
T50V100E	0.01	0.1	0.02	0.14	0.02	0.19	0.03	0.41	0.84	0.22
T250V500E	0.05	0.26	0.05	0.2	0.1	0.31	0.14	0.44	1.21	1.72
T500V1000E	0.31	0.64	0.29	0.64	0.83	1.43	0.83	1.68	4.39	4.01
T1KV2KE	0.53	1.27	1.64	2.61	9.3	11.13	19.2	21.73	36.74	7.27
T1K3KE	0.6	1.75	2.88	4.85	31.77	34.5	78.4	83.58	124.68	7.88
T1K5KE	0.97	2.15	10.77	13.48	131.76	137.27	396.92	407.23	560.13	9.17

The timings were noted for updating the Joined_n table, where n ranges from 1 to 4, and the time taken for discovering substructures of size n while varying the beam. The overall timings for each graph are shown in the column labeled “final 4”. The same test cases were run on Subdue and the timings are also shown in table 6-2. We can see from the results that Subdue outperforms the database version. The overall times taken for the algorithm show that the time was not increasing linearly. The update operation for pass 4 was taking the maximum amount of the time. This can be attributed from the fact that there would be a greater number of tuples in the Joined_4 table. For the test case, T1KV5KE, one of the best substructures found in Subdue is in Figure 6-2. This is one of the substructures embedded in the graph. The frequent_4 table, which contains all the four edge substructures and their counts, are shown in Figure 6-3. As the Figure shows, the number of instances found in both algorithms is the same. The number of instances of the substructure is represented as attribute C1 in Figure 6-3. We use this information to validate that the database algorithm works correctly. In order to improve the performance of the database algorithm, the query for updating the Joined_4 table needs to be optimized.



```
Substructure definition:
Number of subgraph vertices = 5
Number of subgraph edges = 4
Subgraph vertices
       352  v4
       353  bar
       354  foo
       355  v2
       356  v1
Subgraph edges
        [354 -> 352]  e4
        [354 -> 353]  e4
        [356 -> 354]  e2
        [356 -> 355]  e1
Number of instances = 112
Value = 1.062924
Description length of global graph compressed
  using this substructure = 71628.142573
Compression = 0.940801
```

Figure 6-2 Output for Subdue for data set T1KV5KE



V1	V2	V3	V4	V5	E1	E2	E3	E4	C1	X1	X2	X3
v1	v3	v2	foo	bar	e5	e1	e2	e4	119	1	1	4
v1	v3	v2	foo	v4	e5	e1	e2	e4	117	1	1	4
v1	v3	v2	foo	v5	e5	e1	e2	e3	114	1	1	3
v1	v3	foo	bar	v4	e5	e2	e4	e4	113	1	3	3
v1	v3	foo	v4	bar	e5	e2	e4	e4	113	1	3	3
v1	v2	foo	bar	v4	e1	e2	e4	e4	112	1	3	3

Figure 6-3 Frequent_4 table tuples

6.4 Using the Minus operator

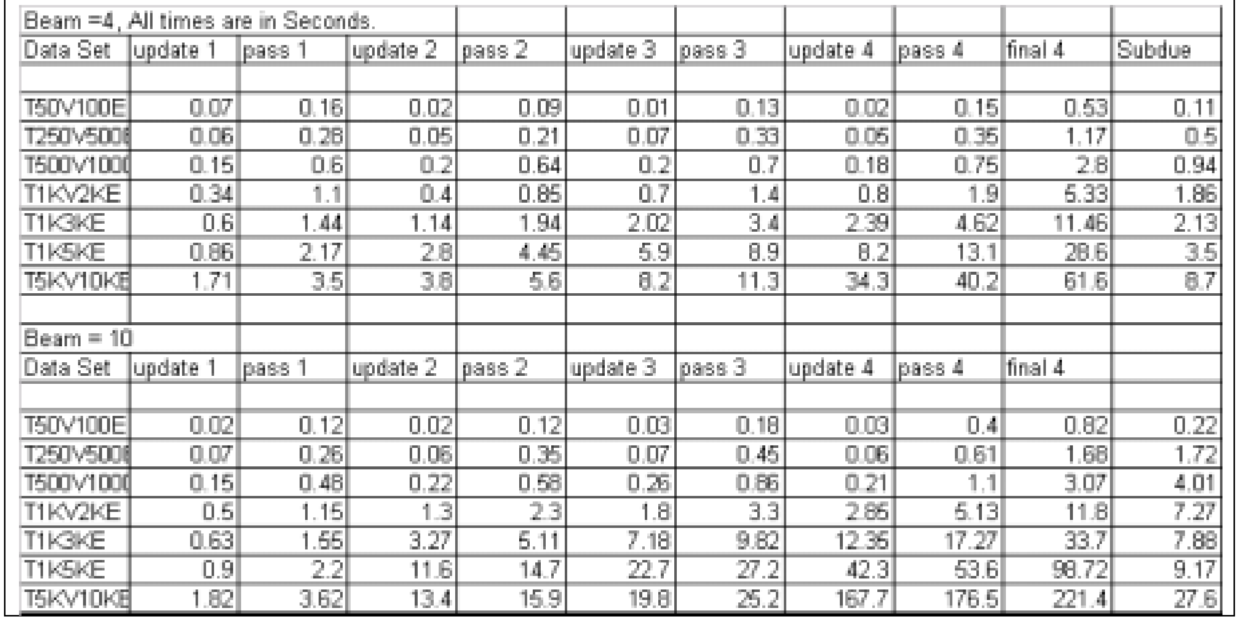
As discussed above, the correlated query is a very costly operation. The challenge is to remove the correlated query but still achieve the functionality. Instead of deleting tuples from the table Joined_n, we create a new table Joined_beam_n. This table is used to store the tuples, which are the instances of the substructures in the table Frequent_beam_n. The EdgeN_subs table contains all the substructures, which do not participate in future extensions. So when EdgeN_subs is joined with the Joined_n table it will result in all the instances of the substructures that do not participate in the future extensions. If we define a set, which contains all the instances of the substructures that are not going to participate in the future extensions then we can subtract this set of tuples from the original set of tuples (Joined_n) to result in instances of the substructures that are going to participate in the future extensions. The **Minus** operator in DB2 can achieve this functionality of the subtract operation. So after subtracting we can store the result set in the table Joined_beam_n. The query to do this is shown below. This avoids the correlated queries.

```
Insert    into Joined_beam_n  ATTRIBUTES
          (
               Select    ATTRIBUTES
               From      Joined_n
               MINUS
               Select    ATTRIBUTES
               From      Joined_n,edgeN_subs f
               Where     f.vertex1name=j.vertex1name and
                         f.vertex2name=j.vertex2name…
                         f.vertexN+1name=j.vertexN+1name
                         and  f.edge1=j.edge1 and
                         f.edge2=j.edge2… f.edgeN=j.edgeN
                         and  f.ext1=j.ext1 …f.extN-
                         1=j.extN-1
          )
```

The program is again tested on the same inputs. The results are tabulated in table 6-3.

The results show that there has been a considerable improvement in run time of the algorithm. If we take the data set T1KV5KE then the time taken for update in the fourth pass for the previous approach was 77.74 seconds compared to 13.1 seconds for this approach. Although this approach does not perform as well as the main memory approach, there has been considerable improvement in time over the previous approach. If we look at the time taken for update 4 then it constitutes more than 75% of the overall time taken. So effort must be taken to still improve the performance of this query. The best substructure discovered by Subdue for the data set T1KV5KE is shown in the Figure 6-2. The best four-edge substructures discovered by this database approach are shown in Figure 6-3.

Table 6-3 Test results using the Minus Operator



Beam =4, All times are in Seconds.										
Data Set	update 1	pass 1	update 2	pass 2	update 3	pass 3	update 4	pass 4	final 4	Subdue
T50V100E	0.07	0.16	0.02	0.09	0.01	0.13	0.02	0.15	0.53	0.11
T250V500	0.06	0.28	0.05	0.21	0.07	0.33	0.05	0.35	1.17	0.5
T500V100	0.15	0.6	0.2	0.64	0.2	0.7	0.18	0.75	2.8	0.94
T1KV2KE	0.34	1.1	0.4	0.85	0.7	1.4	0.8	1.9	5.33	1.86
T1K3KE	0.6	1.44	1.14	1.94	2.02	3.4	2.39	4.62	11.46	2.13
T1K5KE	0.86	2.17	2.8	4.45	5.9	8.9	8.2	13.1	28.6	3.5
T5KV10K	1.71	3.5	3.8	5.6	8.2	11.3	34.3	40.2	61.6	8.7
Beam = 10										
Data Set	update 1	pass 1	update 2	pass 2	update 3	pass 3	update 4	pass 4	final 4	
T50V100E	0.02	0.12	0.02	0.12	0.03	0.18	0.03	0.4	0.82	0.22
T250V500	0.07	0.26	0.06	0.35	0.07	0.45	0.06	0.61	1.68	1.72
T500V100	0.15	0.48	0.22	0.58	0.26	0.86	0.21	1.1	3.07	4.01
T1KV2KE	0.5	1.15	1.3	2.3	1.8	3.3	2.85	5.13	11.8	7.27
T1K3KE	0.63	1.55	3.27	5.11	7.18	9.82	12.35	17.27	33.7	7.88
T1K5KE	0.9	2.2	11.6	14.7	22.7	27.2	42.3	53.6	98.72	9.17
T5KV10K	1.82	3.62	13.4	15.9	19.8	25.2	167.7	176.5	221.4	27.6

6.5 Indexing Techniques

An *index* is an access method that can be created on a table, using one or more columns of the table as the key columns of the index. An index provides a fast way to find rows of the table. Indexes can greatly improve the performance of queries that search for a particular column value or range of values, as well as for joining. An index always provides a logical ordering on the rows of the table. The ordering property of an index is useful in processing queries with ORDER BY and GROUP BY clauses, and is some kind of join algorithms. An example for creating index is shown below.

```
Create Index v1 on vertices (vertexno,vertexname);
```

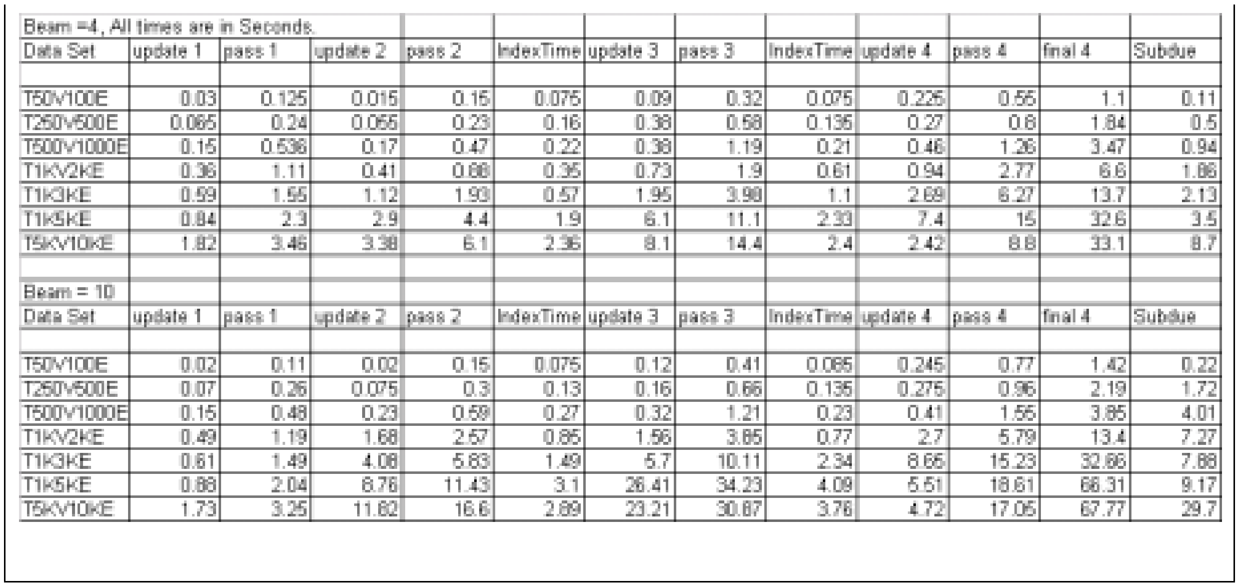
The above statement creates an index v1 on the table vertices with column names vertexno and vertexname. There is always an overhead of creating and maintaining an index. Whenever a tuple is inserted or deleted from a table the corresponding operation has to be done on the index also.

Since there is a join operation involved in the query we are trying to optimize it by creating an index on those attributes involved in the where clause. The index is created in the following way on table Joined_n

```
Create Index j_N
          On Joined_N    (vertex1name..vertexN+1name,
                         edge1..edgeN,ext1..extN-1);
```

The indexes are created after loading the table because otherwise there would be an overhead of updating the index every time a new tuple is inserted into the table. The same tests were re run to see the effect on the run time. The results have been tabulated in the table 6-4. Indexes were created only on the tables Joined_3 and the Joined_4, because the run time for these operations formed the major part of the overall time taken by the program.

Table 6-4 Timings comparison with Indexing



Beam =4, All times are in Seconds.											
Data Set	update 1	pass 1	update 2	pass 2	IndexTime	update 3	pass 3	IndexTime	update 4	pass 4	final 4	Subdue
T50V100E	0.03	0.125	0.015	0.15	0.075	0.09	0.32	0.075	0.225	0.55	1.1	0.11
T250V500E	0.065	0.24	0.065	0.23	0.16	0.38	0.58	0.135	0.27	0.8	1.84	0.5
T500V1000E	0.15	0.536	0.17	0.47	0.22	0.38	1.19	0.21	0.46	1.26	3.47	0.94
T1KV2KE	0.36	1.11	0.41	0.88	0.35	0.73	1.9	0.61	0.94	2.77	6.6	1.86
T1K3KE	0.59	1.55	1.12	1.93	0.57	1.95	3.98	1.1	2.69	6.27	13.7	2.13
T1K5KE	0.84	2.3	2.9	4.4	1.9	6.1	11.1	2.33	7.4	15	32.6	3.5
T5KV10KE	1.82	3.46	3.38	6.1	2.36	8.1	14.4	2.4	2.42	8.8	33.1	8.7
Beam = 10												
Data Set	update 1	pass 1	update 2	pass 2	IndexTime	update 3	pass 3	IndexTime	update 4	pass 4	final 4	Subdue
T50V100E	0.02	0.11	0.02	0.15	0.075	0.12	0.41	0.085	0.245	0.77	1.42	0.22
T250V500E	0.07	0.26	0.075	0.3	0.13	0.16	0.66	0.135	0.275	0.96	2.19	1.72
T500V1000E	0.15	0.48	0.23	0.59	0.27	0.32	1.21	0.23	0.41	1.55	3.85	4.01
T1KV2KE	0.49	1.19	1.68	2.57	0.85	1.56	3.85	0.77	2.7	5.79	13.4	7.27
T1K3KE	0.61	1.49	4.08	5.83	1.49	5.7	10.11	2.34	8.66	15.23	32.86	7.88
T1K5KE	0.88	2.04	8.76	11.43	3.1	26.41	34.23	4.09	5.51	18.61	66.31	9.17
T5KV10KE	1.73	3.25	11.82	16.6	2.89	23.21	30.87	3.76	4.72	17.05	67.77	29.7

In the initial data sets there was not much of an improvement because there is an overhead of creating the index, which is shown in the column Index Time. The first Index time is the time taken to create the index on the Joined_3 table and the second one is the time taken to create the index on Joined_4 table. The timings show a considerable improvement in the run time of the algorithm, especially the update operation of the Joined table. For example, for the data set T5KV10KE and beam =10, the time taken by the update operation is 3.76 seconds compared to 167.7 seconds taken by the previous approach. We can also see that the index on the Joined_3 table did not have much of an improvement on the algorithm. Although there has been a huge improvement in the performance from the previous approach, the performance of Subdue's main memory algorithm is still better than the database approach. For example, the overall time taken by the data set T5KV10KE using beam size 4 for Subdue is just 8.7 seconds, compared to the 33.1 seconds taken by the database algorithm.

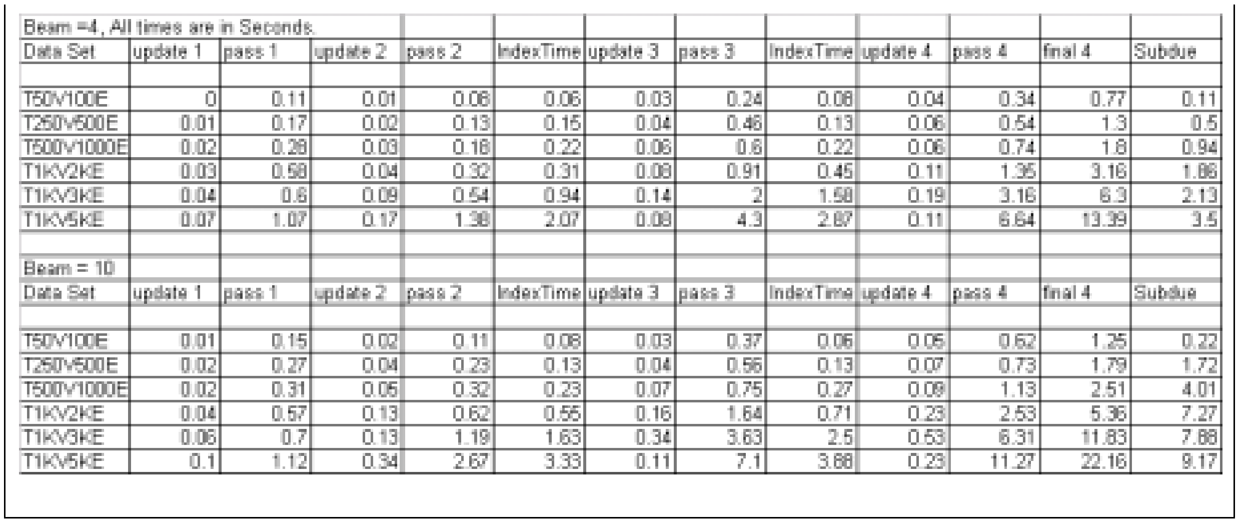
6.6 Updating without the Minus operator

This approach is the one discussed in chapter 5. The assumption while implementing this approach is that the beam number is a very small number compared to the number of substructures. Instead of creating the table EdgeN_subs that have all the instances of the substructures that do not participate in the future extensions, we create a table Frequent_beam_n table that contains the substructures that are going to participate in the future extensions. Hence, the Frequent_beam_n table will have at most beam number of tuples, which would be much smaller than the table Frequent_n itself. It would be just enough to gather all the instances of the substructures, which are in the table Frequent_beam_n table. Since the Joined_n table contains all the instances of the substructures of size n, we can gather the instances of the substructures in Frequent_beam_n table by joining the Frequent_beam_n table with Joined_n table. The query to achieve this is shown below:

```
Insert    into Joined_beam_n  ATTRIBUTES
          (
               Select    ATTRIBUTES
               From      Joined_n j, Frequent_beam_n f
               Where     f.vertex1name=j.vertex1name and
                         f.vertex2name=j.vertex2name…
                         f.vertexN+1name=j.vertexN+1name
                         and f.edge1=j.edge1 and
                         f.edge2=j.edge2… f.edgeN=j.edgeN
                         and  f.ext1=j.ext1 …f.extN-
                         1=j.extN-1
          )
```

The run times of this approach are shown in table 6-5.

Table 6-5 Timings without using the Minus operator



Beam =4, All times are in Seconds.												
Data Set	update 1	pass 1	update 2	pass 2	IndexTime	update 3	pass 3	IndexTime	update 4	pass 4	final 4	Subdue
T50V100E	0	0.11	0.01	0.08	0.06	0.03	0.24	0.08	0.04	0.34	0.77	0.11
T250V500E	0.01	0.17	0.02	0.13	0.15	0.04	0.46	0.13	0.06	0.54	1.3	0.5
T500V1000E	0.02	0.28	0.03	0.18	0.22	0.06	0.6	0.22	0.06	0.74	1.8	0.94
T1KV2KE	0.03	0.58	0.04	0.32	0.31	0.08	0.91	0.45	0.11	1.35	3.16	1.86
T1KV3KE	0.04	0.6	0.09	0.54	0.94	0.14	2	1.58	0.19	3.16	6.3	2.13
T1KV5KE	0.07	1.07	0.17	1.38	2.07	0.08	4.3	2.87	0.11	6.64	13.39	3.5
Beam = 10												
Data Set	update 1	pass 1	update 2	pass 2	IndexTime	update 3	pass 3	IndexTime	update 4	pass 4	final 4	Subdue
T50V100E	0.01	0.15	0.02	0.11	0.08	0.03	0.37	0.06	0.05	0.62	1.25	0.22
T250V500E	0.02	0.27	0.04	0.23	0.13	0.04	0.56	0.13	0.07	0.73	1.79	1.72
T500V1000E	0.02	0.31	0.05	0.32	0.23	0.07	0.75	0.27	0.09	1.13	2.51	4.01
T1KV2KE	0.04	0.57	0.13	0.62	0.55	0.16	1.64	0.71	0.23	2.53	5.36	7.27
T1KV3KE	0.06	0.7	0.13	1.19	1.63	0.34	3.63	2.5	0.53	6.31	11.83	7.88
T1KV5KE	0.1	1.12	0.34	2.67	3.33	0.11	7.1	3.88	0.23	11.27	22.16	9.17

There is a definite decrease in the time taken by this approach compared to the previous approach. For example, consider the pass 3 of the data set T1KV5KE; the time taken for by the previous approach is 30.87 seconds compared to the 7.1 seconds taken by this approach. The improvement has been in the update times, which is just 0.11 seconds taken by this approach compared to 23.21 seconds of the previous approach. The graphical comparison graphs for the overall timings for all the approaches are shown in the table 6-6 and 6-7. The timings are shown on the logarithmic scale, as the range is very large.

Table 6-6 Comparing the Final times

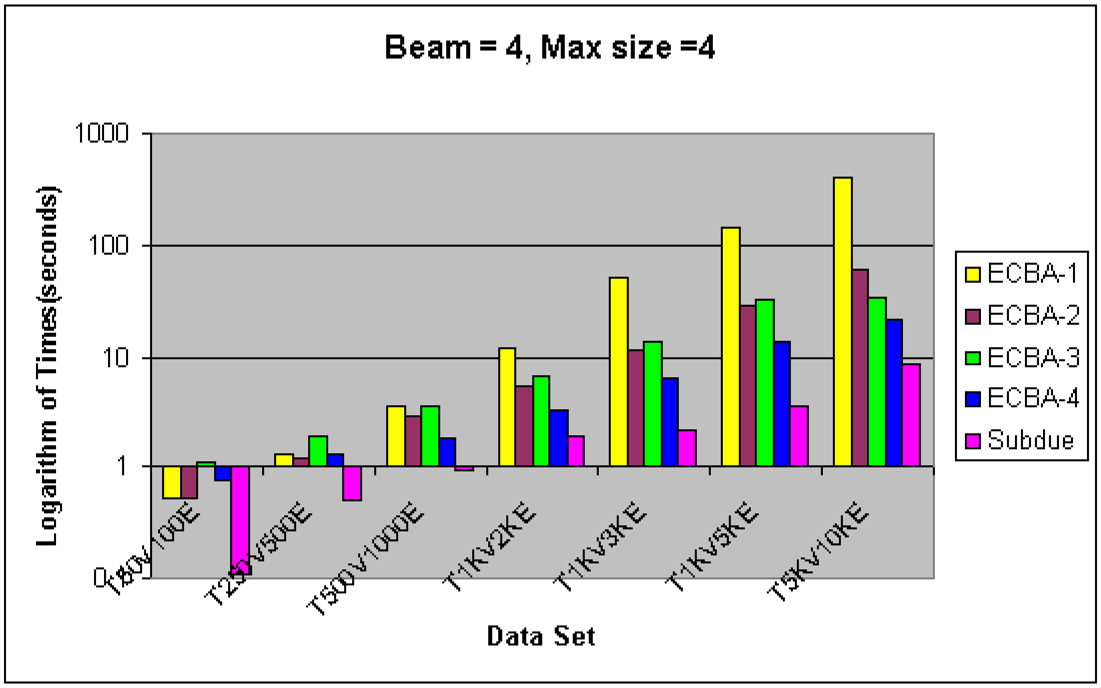
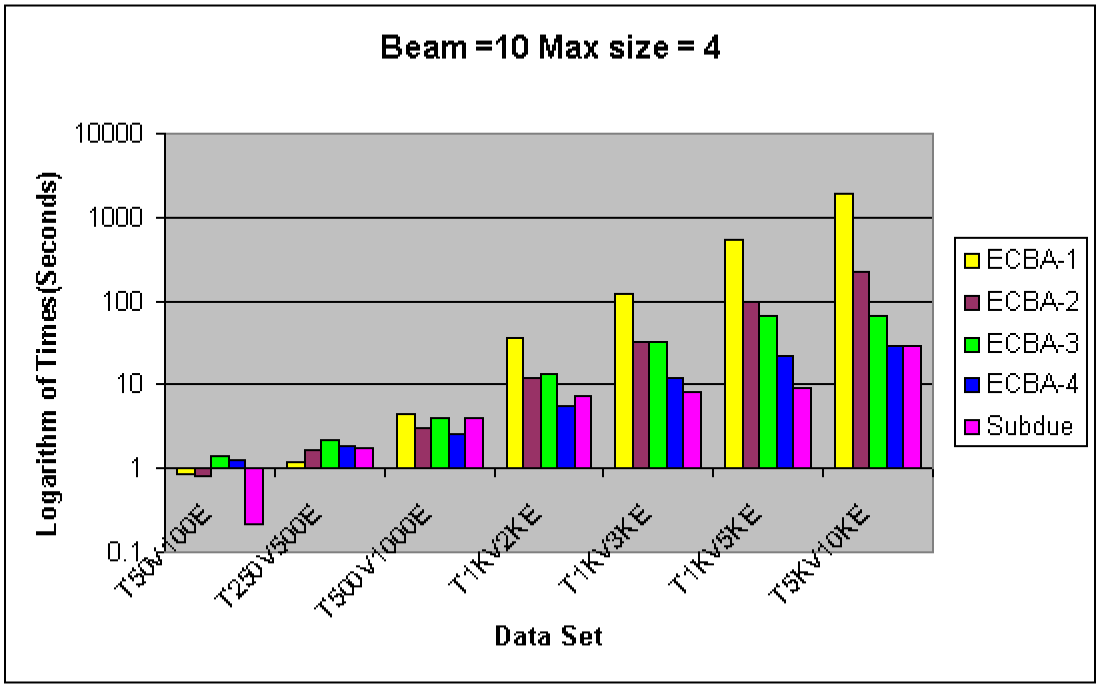


Table 6-7 Comparison of overall times

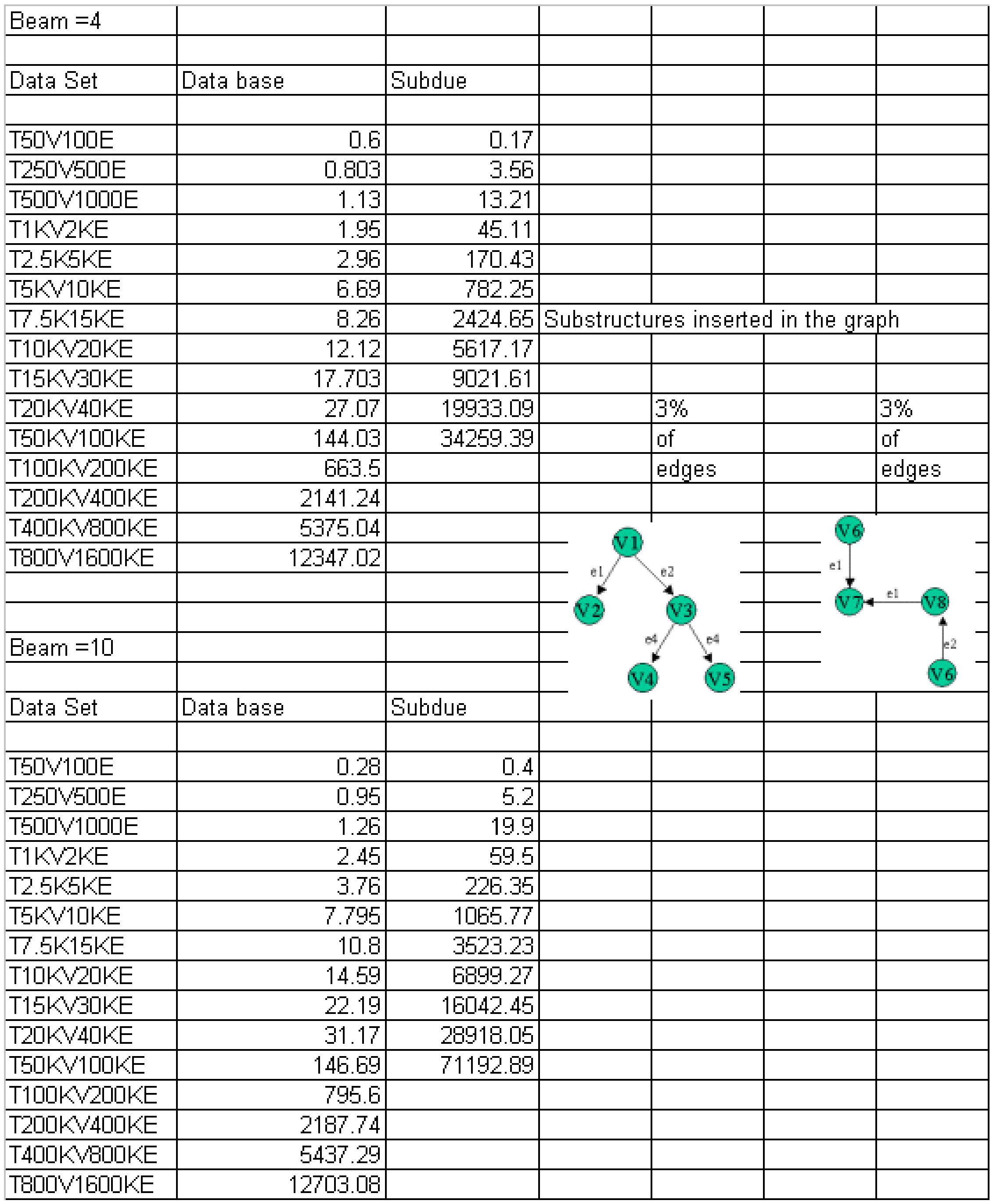


6.7 Achieving Scalability

The above tests could only help in making a comparison within the approaches but could not establish any proof of scalability of the approaches. The code used for generating the test cases could not create test cases large enough to test for higher data sets. Another graph generator described in section 3.3.4 has been used for testing the algorithm further. The generator was able to create data sets of size 1,600,000 edges and 800,000 vertices. Tests have been done on these data sets on main memory as well as the last approach discussed above. The results are shown in table 6- 8.

The final timings and their comparison with Subdue are shown. The number of instances of the substructures discovered by both the algorithms turned out to be the same. The crossover point for the algorithm timing takes place at as low as 100 edges. The main memory algorithm took more than 60,000 seconds to initialize the T400KV1600KE graph and took more than 20 hours to initialize the T800KV1600E data set. We could not go beyond the T800KV1600KE data set for testing because the graph generator could not produce the required graph. The run time for the data set T800KV1600KE using the database version with a beam size of 10 is 12,703 seconds. The time taken by the Subdue algorithm for a data set of T50KV100KE and a beam size of 10 is 71,192 seconds. One of the main reasons for such improvement has been using pure SQL statements to achieve the functionality. The graphical comparison of the approaches is shown in tables 6-9 and 6-10.

Table 6-8 Comparison of Timings



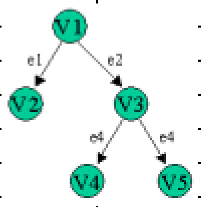
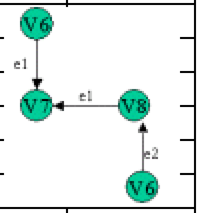
Beam =4						
Data Set	Data base	Subdue				
T50V100E	0.6	0.17				
T250V500E	0.803	3.56				
T500V1000E	1.13	13.21				
T1KV2KE	1.95	45.11				
T2.5K5KE	2.96	170.43				
T5KV10KE	6.69	782.25				
T7.5K15KE	8.26	2424.65	Substructures inserted in the graph			
T10KV20KE	12.12	5617.17				
T15KV30KE	17.703	9021.61				
T20KV40KE	27.07	19933.09		3%		3%
T50KV100KE	144.03	34259.39		of		of
T100KV200KE	663.5			edges		edges
T200KV400KE	2141.24					
T400KV800KE	5375.04					
T800V1600KE	12347.02					
Beam =10						
Data Set	Data base	Subdue				
T50V100E	0.28	0.4				
T250V500E	0.95	5.2				
T500V1000E	1.26	19.9				
T1KV2KE	2.45	59.5				
T2.5K5KE	3.76	226.35				
T5KV10KE	7.795	1065.77				
T7.5K15KE	10.8	3523.23				
T10KV20KE	14.59	6899.27				
T15KV30KE	22.19	16042.45				
T20KV40KE	31.17	28918.05				
T50KV100KE	146.69	71192.89				
T100KV200KE	795.6					
T200KV400KE	2187.74					
T400KV800KE	5437.29					
T800V1600KE	12703.08					

Table 6-9 Graphical comparison of the approaches

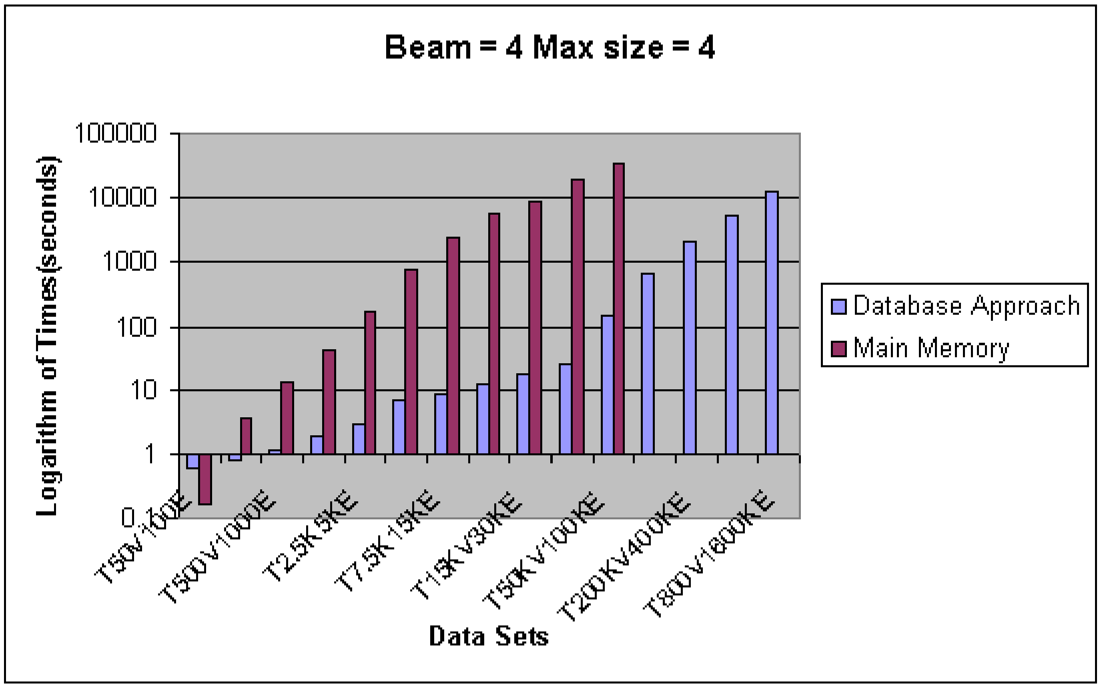
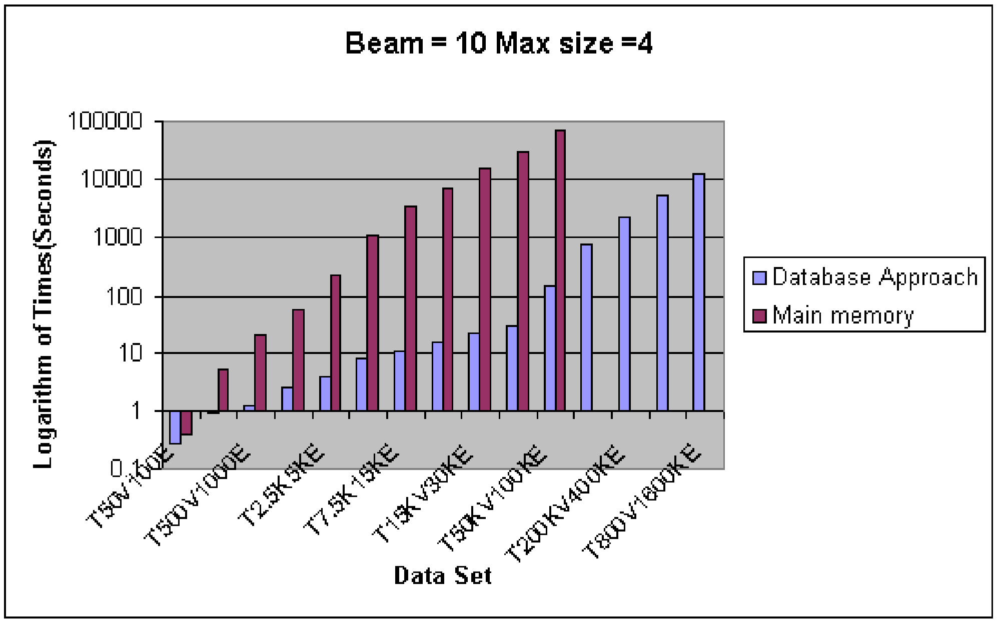


Table 6-10 Graphical Comparison of the Approaches



CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusion and Future Work

In this thesis we have developed several algorithms for graph-based data mining using relational databases. The idea behind graph-based data mining was to find interesting and repetitive substructures in a graph. Initial efforts included mapping graphs into a database and achieve the functionalities of Subdue. One of the main challenges was the representation of the substructures in a database and discovering these substructures. Our first algorithm developed achieved the functionality required but lacked scalability. The next challenge is to achieve the scalability and apply the concept of beam. The ECBA implements the graph discovery in pure SQL statements and uses indexing techniques to improve the performance of the algorithm. We were able to run graph mining on data sets that have 800K vertices and 1600K edges. The algorithms developed were able to achieve the functionality desired (scalability). The algorithms were able to find substructures and their number of instances correctly. Much functionality like beam and max size was successfully implemented in this thesis.

We are still in the process of implementing the concept of overlap. For considering the issues involved in overlap the instances of the substructure have to be carefully analyzed to see if there exists a common vertex between the two instances. One of the important concepts while dealing with graphs is detecting cycles. The current algorithm though detects the cycles correctly but after detecting the cycles it still loops within the cycle. Cycles need to be carefully handled and reported. We are also in a process of implementing a concept like MDL, which would report compression achieved by a substructure in a database system.

REFERENCES

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

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

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

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

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

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

7. Chamberlin, D., *A Complete Guide to DB2 Universal Database*. 1998: Morgan Kaufmann Publishers, Inc.

8. Noble, C., *Graphgen.*

9. Holder, L.B., *Subgen*.

BIOGRAPHICAL INFORMATION

Ramji Beera was born on May 28, 1979 in Hyderabad, India. He received his Bachelor of Technology degree in Computer Science and Engineering from Indian Institute of Technology, Madras, India in May 2000. In the Fall of 2000, 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 August 2003. His research interests include graph based mining and Business Intelligence.