OO-FSG: An Object-Oriented Approach to Mine Frequent Subgraphs

Bismita Srichandan **Rajshekhar Sunderraman**

Department of Computer Science
Georgia State University
Atlanta, USA
Email: bsrichandan1@student.gsu.edu, raj@cs.gsu.edu

Abstract

Frequent subgraph mining (FSG) has always been an important issue in data mining. Several frequent subgraph mining methods have been developed for mining graph data. However, most of these are main memory algorithms in which scalability is a bigger issue. A few algorithms have opted for a relational approach that stores the graph data in relational tables. However, relational databases have their own space as well computing constraints when it comes to storing large databases. Moreover, relational databases do not preserve semantic information as they represent simple entities and in order to preserve the relationship between two entities additional tables are necessary. Object-oriented databases, on the other hand, do not have these constraints. In this paper, we present an object-oriented database approach to mining frequent sub-graphs. We use Db4o, a popular open-source object database system, to store the input graph data as well as intermediate results. Db4o can save all the information about an entity in a single class in an object form. Application domains such as protein-protein interaction data, social network data, and chemical compound structure data require mining frequent subgraphs while preserving the meaning. This paper proposes a novel idea for using object oriented database db4o to store graph data, which can support large graph data as well as preserve semantic information.

Keywords: Data Mining, Frequent Subgraphs, Object Oriented Database

1 Introduction

There are many efficient algorithms for finding frequent itemsets in very large transaction databases (Agrawal et al. 1994, Agarwal et al. 1998, Zaki et al. 2001, Han et al. 2000). We can use these itemsets for discovering association rules, for extracting prevalent patterns that exist in the datasets, or for classification. However, we can't apply these techniques over datasets which are not itemsets. In recent years, there has been an increased interest in developing data mining algorithms that operate on graphs. Such graphs arise naturally in a number of different application domains, including computer networks, bioinformatics, semantic web, chemical compound and social networks. All these domains mentioned require mining of frequent subgraphs over large data sets. However, the algorithms developed so far are not scalable. Most works done on frequent subgraph mining have focused on algorithms that assume graph data is stored in main memory. Memory dependent algorithms could be very inefficient if the dataset is large. In this paper we propose to store the input graph data set as well as intermediate results in an object-oriented database and extend the FSG mining algorithms to work with object-oriented databases. This approach scales nicely for large data sets that cannot fit in main memory. We also show that using object-oriented databases over relational systems had an advantage in performance and scalability.

2 Related Work

We now discuss related work dealing with both in-memory as well as disk storage algorithms.

2.1 In-memory Methods

Cook et al. (2000) proposed SUBDUE to discover the best compressing structures. Inokuchi et al. (2003) proposed an Apriori based algorithm to discover all frequent substructures. Kuramochi et al. (2001) proposed FSG algorithm which represents graphs as sparse adjacency matrix and uses canonical labeling to determine subgraph isomorphism. Han et al.(2002) proposed gSpan algorithm which uses the depth-first search and generates lesser candidate items than FSG. Kuramochi et al. (2005) proposed an algorithm to find frequent patterns from a large sparse graph. Jiang et al. (2009) tried to find globally frequent subgraphs on a single labeled graph. All the above mentioned algorithms are not scalable because when the graph data is too large, it consumes the memory completely and reduces the efficiency.

2.2 In-disk method, DB-FSG

Chakravarthy et al. (2004) proposed an algorithm DB-Subdue which is the first attempt to implement the database approach for graph mining. Following DB-Subdue, Chakravarthy et al. (2008) proposed an SQL-based approach for frequent subgraph mining (DB-FSG). In their work, they have used relational tables to store graph data and subgraphs. Their approach is briefly as follows: the method has two tables to begin with: one for the vertices and other for the edges which contain individual vertices and edges. The individual tables are joined to obtain size-1 subgraphs. Each time the candidates are generated, the columns in the table grow depending on the size of the graphs. This will eventually place a limit on the size

of the maximum substructure that can be detected, as there is a limit on the number of columns a relation can have in a relational database. The algorithm described in DB-FSG can discover substructures of size 165 at the most. After implementing their algorithm, we figured out that it poses many difficulties for larger datasets, and efficiency is the major drawback when the dataset size is large.

Another issue related to relational database is storing semantic information. As graph databases, like chemical compounds and protein-protein interactions, have explicit relations between elements, semantic information must be taken care of by the data model.

We implemented the algorithm by Chakravarthy et al. (2008) using the same datasets to analyze the pros and cons of both approaches. The algorithm is as follows:

Algorithm 1 DB-FSG - Chakravarthy et al. (2008)

Input: A graph dataset Gs and min_*sup*
Output: The frequent subgraph set S
Method:

1. Create oneedge (instance_1) table by joining vertex table and edge table
2. Remove the edges with instance count less than support from the oneedge table
3. for n=2 to MaxSize do
   (a) Join instance (n-1) with oneedge table to generate instance n
   (b) Eliminate pseudo duplicates from instance n table
   (c) Canonically order instance n table on vertex labels
   (d) Project distinct vertex label, edge label and gid to obtain one instance per substructure for each graph and store in dist n table.
   (e) Group dist n table by vertex label and edge label to obtain substructures and its count
   (f) Retain only the instances of substructure satisfying support and store it in instance n table
   (g) If there are no instances of substructure satisfying support then stop.
4. end for

The rest of the paper is organized as follows. Section 3 describes the object-oriented approach. Section 4 describes the OO-FSG algorithm. Section 5 shows the experimental evaluation. Section 6 concludes the paper and points some of the future works in progress.

3 An OO-approach to Mine FSGs

We are proposing to use db4o, an object-oriented database (http://www.db4o.com/) to store the graph dataset. db4o stores everything as objects. The advantage of using db4o as the data storage is because it's highly scalable and do not put burden on memory. To begin with, our approach includes the following basic classes: Vertex, Edge, SingleEdge and Subgraph_1 shown in Fig. 1. The classes are extended as the size of subgraphs increase. For example, for size-2 subgraphs, TwoEdge and Subgraph_2 classes are used. Note that the paper focuses on directed labeled graphs where the direction is assumed to be from a smaller vertex number to the larger vertex number. For example, if the vertices are given the numbers as 0, 1, 2, 3 etc., then the direction of the edges are considered to be from 0 to 1, 1 to 2 or 2 to 3 but not 3 to 1. Hence our method does not need any specific field to keep track of direction between the vertices.

The Vertex class represents nodes in the graph. In the Vertex class, each object has a unique object identity which is 'VertexNo' and label as 'VertexLabel'. Fig. 2 shows a simple subgraph where the numbers 1 and 2 represent vertex numbers which are allocated for ease of use, but these do not have any significance for subgraph mining. A and B represent labels of the vertices and C is the edge label. Each vertex object represents a node of the given graph; the Edge class is similar to the Vertex class. Each edge object represents an edge of the given graph. The SingleEdge class is the combination of the Vertex and Edge classes. It contains all the single-edged subgraphs. The Vertex and Edge classes are constructed separately as the Edge class does not contain the label details of the vertices. The Subgraph_1 class includes all the subgraphs satisfying minimum support which is described in the subsequent sections. The Subscript '1' in Subgraph_1 represents the size-1 subgraphs. As the sizes of the subgraphs increase the subscript changes.

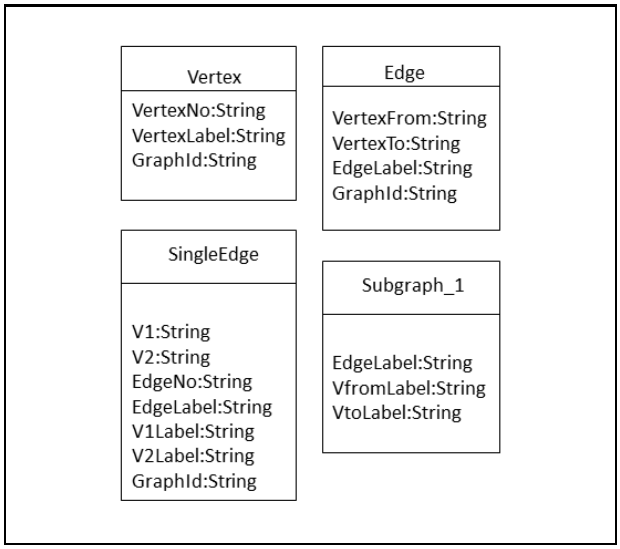

Figure 1: All major classes

Figure 2: An Example Subgraph

In the following sub-sections, we now present the subgraph construction and FSG determination, optimization techniques used in the object-oriented approach, and comparison of DB-FSG with OO-FSG with respect to implementation. Experimental comparison comes later in the paper.

3.1 Sub-graph Construction and FSG Determination

In this sub-section, we discuss the sub-graph construction process and determination of FSGs. We start with a definition.

Definition 1: A *labeled graph* is represented by a 4-tuple, G = (V, E, L, l), where

V is a set of vertices (or nodes)

E $\subseteq$ V $\times$ V is a set of edges, they can be directed or undirected

L is a set of labels

l: V $\cup$ E $\rightarrow$ L, l is a function assigning labels to the vertices and the edges

3.1.1 Sub-graph Support

In Fig. 3, three transaction graphs are shown. The numbers 0, 1, 2 and 3 are the numbers assigned for programming purpose. The labels (names) A, B, C, D, E, F and G are important to the algorithm. Let *nGraph* be the total number of graphs in the dataset and *nSubGraph* be the number of times a particular sub-graph appears in the dataset. Then, the support 'Sup' of a particular subgraph is defined as:

$$Sup = nSubGraph \div nGraph$$

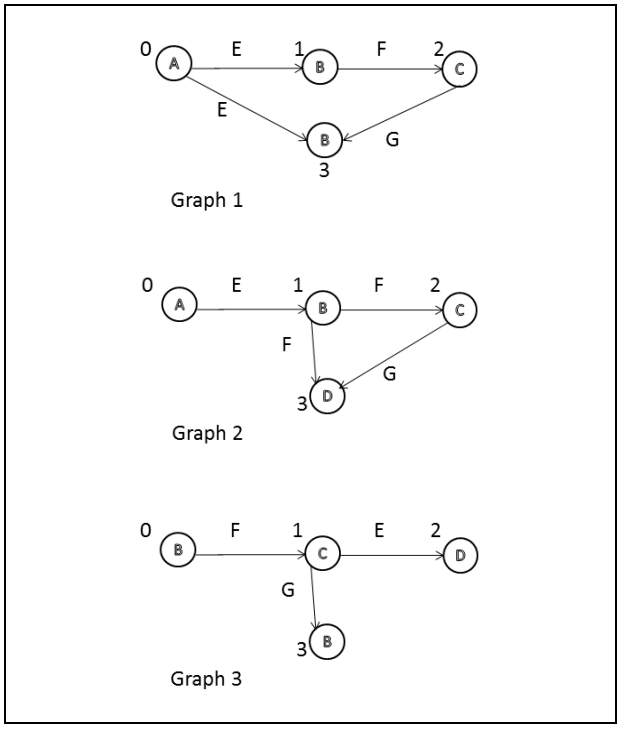

Figure 3: Representation of graphs in the dataset

Graph isomorphism problem needs to be tackled while counting the support of subgraphs in the dataset. Two instances are isomorphic if the vertex and edge labels are same and directions are same. In our experiment, the direction is assumed to be from the lower numbered vertex to the higher numbered vertex. For example, if we count the number of occurrences of subgraph A-E-B in the three graphs, the count is 3, but in reality it is 2. Graph 1 contains the subgraph A-E-B twice. That must be counted once. This problem is eliminated by finding the distinct subgraphs per graph. Note that though we count only one instance of the subgraph per graph, we do not discard the other instances before pruning. The problem could be the instance omitted might have significance in the discovery of the subgraph of size 2. If we remove the pair 0-1 (vertex numbers) from graph 1 instead of 0-3, then in the next level, construction of subgraph_2 would not generate the subgraph A-B-C (0-1-2). So we store the other instances too.

3.1.2 Subgraph construction

This section elaborates the process of subgraph construction. To begin with, we save the vertices and edges in different classes named Vertex and Edge classes. Since Edge class does not have information on the labels of the vertices, we join the Vertex and Edge classes based on the vertex numbers and the graph id to create the size-1 subgraphs stored in SingleEdge class. The graph id must be same during joining as the expansion happens in the same graph. SingleEdge class contains all the information on the vertices and edge labels. Considering the graphs shown in Fig. 4, the size-1 edges are shown in the diagrams. In order to provide a detailed view of the relational method (Chakravarthy et al., 2008) and object-oriented method, we have provided the tables along with the graph structures. Subsequent stages of construction are also shown in the figures. The edges are assigned a number to keep track of the edges joined during candidate generation. Actually, they are stored as objects in the db4o database.

Table 1: Vertex Table

VertexNo	VertexLabel	GraphId
0	A	1
1	B	1
2	C	1
3	B	1
0	A	2
1	B	2
2	C	2
3	D	2
0	B	3
1	C	3
2	D	3
3	B	3

Table 2: Edge Table

V1	V2	EdgeLabel	GraphId
0	1	E	1
0	3	E	1
1	2	F	1
2	3	G	1
0	1	E	2
1	2	F	2
1	3	F	2
2	3	G	2
0	1	F	3
1	2	E	3
1	3	G	3

Table 3: SingleEdge Table

V1	V2	ENo	ELabel	V1L	V2L	GId
0	1	1	E	A	B	1
0	3	2	E	A	B	1
1	2	3	F	B	C	1
2	3	4	G	C	B	1
0	1	5	E	A	B	2
1	2	6	F	B	C	2
1	3	7	F	B	D	2
2	3	8	G	C	D	2
0	1	9	F	B	C	3
1	2	10	E	C	D	3
1	3	11	G	C	B	3

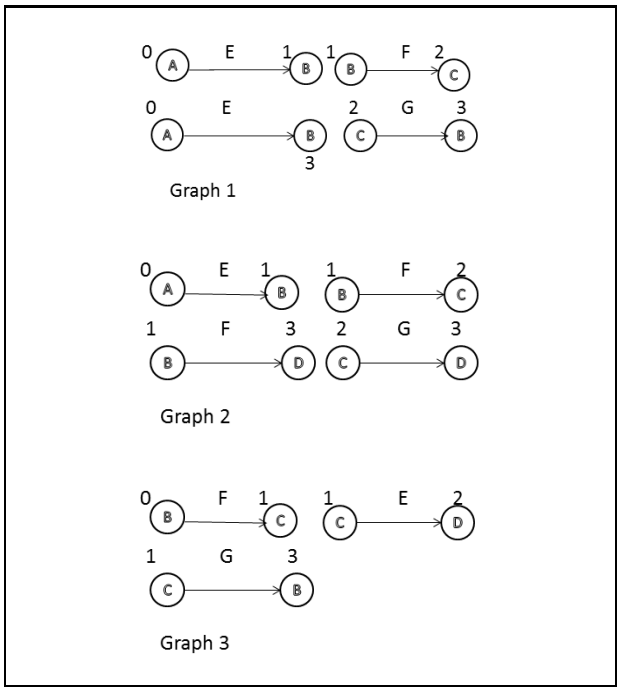

Figure 4: Objects of SingleEdge class

Table 1 contains all the objects in the Vertex class where each row represents the individual objects of the Vertex class. Vertex and Edge classes are not shown in graphical format. Similarly, Table 2 has all the objects which are individual edge objects. After joining the Vertex and Edge classes, we obtained the SingleEdge class shown in Fig. 4. In order to generate size-2 subgraphs, each object of SingleEdge class is joined with itself where V2 of the first edge is same as the V1 of the other object. In all cases, joining happens within the same graph. Unlike, relational databases where there is a defined join query using SQL; db4o does not have such join queries. Instead, it supports a query called 'Native Query' which constrains the class to be joined and has a keyword called 'descend' which goes down to the field level to query the data.

The TwoEdge class is shown in Fig. 5. The notations in the corresponding TwoEdge table are squeezed to fit more columns and are as follows: E1L-label of edge 1, E2L-label of edge 2, V1L-label of vertex 1, V2L-label of vertex 2, V3L-label of vertex 3. The ThreeEdge class is constructed from TwoEdge and SingleEdge classes. Note here that the SingleEdge, TwoEdge and ThreeEdge classes at this stage, do not contain the pruned data, and these are the possible sub-structures before considering pruning. An example of the states of the SingleEdge and TwoEdge classes after pruning is shown in section 4 under pruning. In order to construct the three-edge subgraphs, the query searches for the match for the V3 node from table 4 with the node V1 from table 3 with the same graph id. ThreeEdge class is shown in Fig 6.

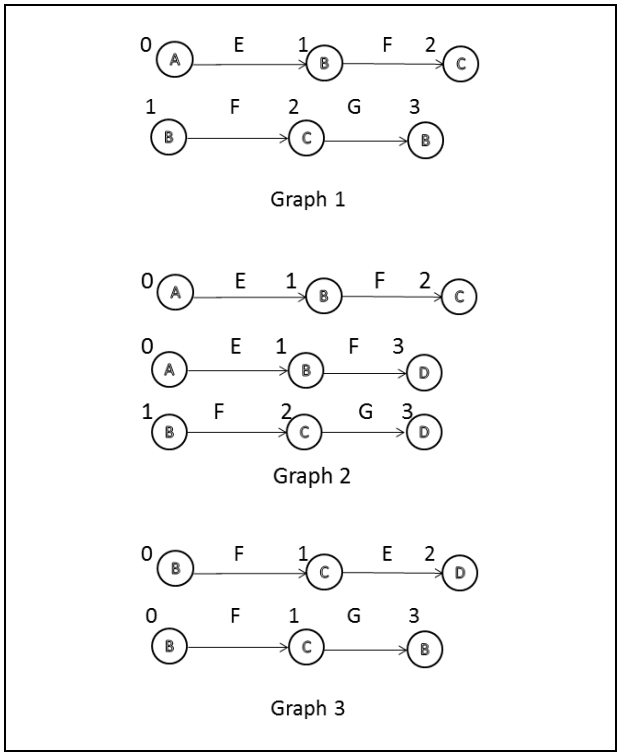

Figure 5: Objects of TwoEdge class

Table 4: TwoEdge Table

V1	V2	V3	E1L	E2L	V1L	V2L	V3L	GId
0	1	2	E	F	A	B	C	1
1	2	3	F	G	B	C	B	1
0	1	2	E	F	A	B	C	2
0	1	3	E	F	A	B	D	2
1	2	3	F	G	B	C	D	2
0	1	2	F	E	B	C	D	3
0	1	3	F	G	B	C	B	3

Table 5: ThreeEdge Table

v1	v2	v3	v4	e1L	e2L	e3L	v1L	v2L	v3L	v4L	GId
0	1	2	3	E	F	G	A	B	C	B	1
0	1	2	3	E	F	G	A	B	C	D	2

3.1.3 Determining frequent subgraphs

Frequent subgraphs are determined based on the number of times it appears in the whole dataset. If we consider the single-edge subgraphs shown in Fig 4, there are eleven of them, but AEB (Graphs 1 and 2), CGB (Graphs 1 and 3) and BFC (Graph 1, 2

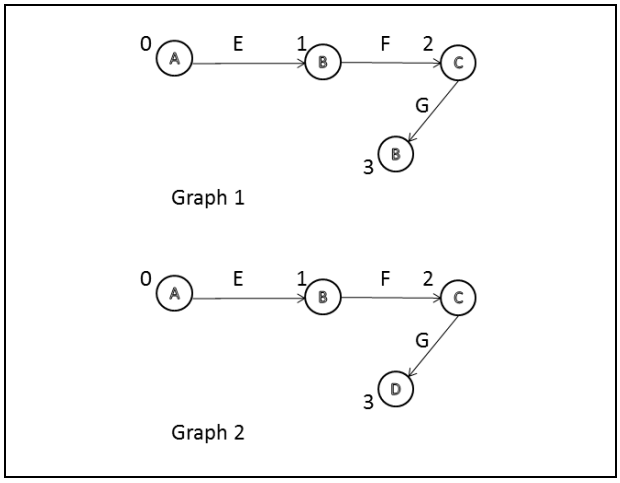

Figure 6: Objects of ThreeEdge class

and 3) appear more than once. Hence it is obvious that the other subgraphs except AEB, CGB and BFC are insignificant. Our purpose is to find the subgraphs which occur more than a specific number of times (min_*sup* provided by the user) in the dataset. Let's consider the minimum support as 2, which mean a subgraph must be appearing in at least two graphs. Subgraph_1 class contains the following size-1 subgraphs shown in the Fig 7. Notice that the Subgraph_1 class does not have the numbers of the vertices. Only significant details, the labels are stored. Fig. 8 shows the frequent subgraphs (minimum support-2) of size-2.

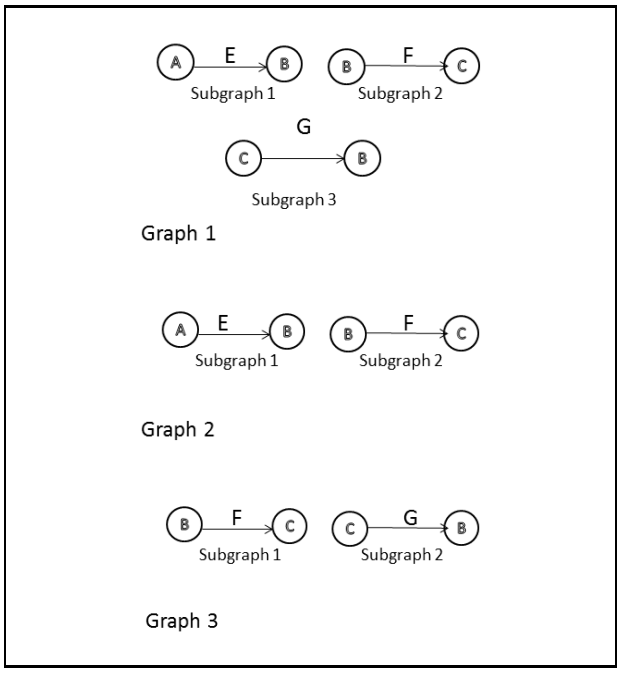

Figure 7: Objects of Subgraph_1 class

3.2 Optimization Techniques

This section discusses various optimization techniques used in object-oriented approach. We have used available data structures across the application to avoid frequent querying of the object database and hence increasing efficiency. Though data structures are used to make the processes faster, the applications are independent of each other, in other words, the graph dataset is always in db4o store. In order to retrieve the distinct instances (to tackle graph isomorphism), we used the data structure "hash sets". In many places, common Java data structures are used to make application process faster. Subgraph counting time has been dramatically improved by using "MultiKey" and "MultiValueMap" common collections data structure available from apache.org. MultiKey can store the same sub-structure instances more than once; in other words, the keys do not need to be unique. For example, considering Fig. 7 we can save the sub-structure A-E-B from both the graphs 1 and 2.

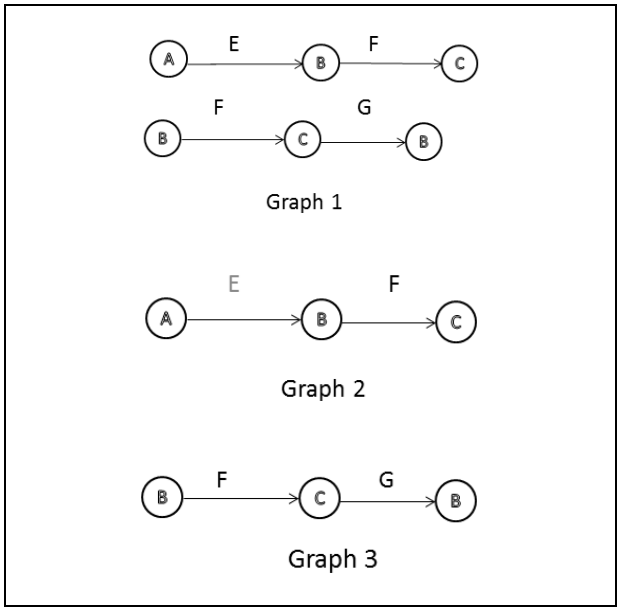

Figure 8: Objects of Subgraph_2 class

3.3 DB-FSG vs OO-FSG: Implementation

The coding of Algorithm 1 was done in Java using Oracle 11g. The tables have the same name as the classes in db4o. We tried to optimize the relational method as much as possible by using indexes, and prepared statements for the insert statements. We noticed a significant time delay while inserting millions of records, whereas in db4o approaches it takes significantly less time. Initial data loading was quite time consuming, so we used Perl script to minimize the time by separating raw input data to Vertex, Edge and SingleEdge files to load into the relational database. For small sized datasets, the efficiency of relational and db4o approach are nearly same, but as the dataset size increases, the performance of db4o over relational increases dramatically. The only problem with db4o approach is it needs strong programming skills whereas relational approach solves things with simple queries. But, at the same time manipulating millions of database records through queries has a huge drag on efficiency. Also the join queries of SQL get messy when we join more than 2 tables. The queries of db4o database are quite simple. A comparison of both queries is given below.

The following is an example of a query used to join the matching vertices of the SingleEdge class/table with itself in order to obtain TwoEdge class/table objects/rows. In the SODA (db4o query) query the "vertexFrom" from the SingleEdge class is joined with the matching "vertexTo" of the same SingleEdge

class.

In the second statement of the code snippet the keyword "constrain" constrains the SingleEdge class. In the third statement the "descend" keyword means starting from the class level the query goes down to one level to the "VertexFrom" field and "constrain" keyword is used to match the "VertexTo" of the SingleEdge class. VertexFrom and VertexTo are the fields in the SingleEdge class and they are named so to indicate the direction of the edges. The last statement executes the query which retrieves all the matching objects in the class. We have also shown the SQL version of the query.

db4o version of a join query

```
query = db.query();
query.constrain(DB4OSingleEdge.class);
query.descend("VertexFrom")
         .constrain(VertexTo)
         .and(GraphId.constrain(GraphId));
query.execute();
```

SQL version of the same query

```
select distinct
  I1.VertexFrom as V1,
  I1.VertexTo as V2,
  I2.VertexTo as V3,
  I1.EdgeNo as E1No,
  I2.EdgeNo as E2No,
  I1.EdgeLabel as E1Label ,
  I2.EdgeLabel as E2Label,
  I1.VfromLabel as V1Label,
  I1.VtoLabel as V2Label,
  I2.VtoLabel as V3Label,
  I1.GraphId
from
  oneedge I1, oneedge I2
where
  I1.VertexTo = I2.VertexFrom and
  I1.GraphId = I2.GraphId;
```

4 Details of OO-FSG Algorithm

OO-FSG algorithm has two major aspects. One is generating candidates and another one is pruning the insignificant edges from the graphs. Each step of the algorithm is discussed in detail. In the algorithm, first step is for the construction of SingleEdge class from Vertex and Edge classes. In the second step, the distinct single edges are separated to get rid of isomorphic structures and stored in Subgraph_1 class. Counting of the distinct edges is done using MultiKey and MultiValueMap on the whole dataset with the user provided minimum support(min_*sup*). In the third step, we remove the edges which fail to satisfy the minimum support value from the SingleEdge class. Step 4 is the looping condition, looping occurs from steps 4 (a) through 4 (e) until size-n which is 5 for our experiment. Step 4 (a) combines the SingleEdge class with itself based on the matching vertices and graph id. Step 4 (b) removes the redundant subgraphs to find the distinct instances and stores in the temporary class Subgraph_*Distinct*_2 class. In Step 4 (c), we count the subgraphs. When we say subgraphs, means only the edge labels and vertex labels not the numbers given to the nodes and edges. Steps 4 (d) and 4 (e) are self-explanatory. In the second iteration of the loop, we combine TwoEdge class with SingleEdge class and follow the steps accordingly. We keep repeating the loop until we get a subgraph of size-5.

Algorithm 2 OO-FSG Algorithm

Input: A graph dataset Gs and min_*sup*
Output: The frequent subgraph set S
Method:

1. construct SingleEdge class by joining Vertex and Edge class.
2. select distinct single edges and store the subgraphs which satisfies min_*sup* in Subgraph_1 class.
3. remove the edges with count less than the min_*sup* from SingleEdge.
4. repeat steps a through e until a candidate subgraph of size-N with min_*sup* is generated.
   (a) join (N-1)Edge class with SingleEdge class to generate *(N)Edge.
   (b) eliminate the redundant subgraphs from (N)Edge and store the size-N subgraphs in Subgraph_*Distinct*_*N* class.
   (c) count the unique vertex and edge labels in the Subgraph_*Distinct*_*N* class.
   (d) eliminate the subgraphs from Subgraph_*Distinct*_*N* with count less than min_*sup* and store it in Subgraph_*N* class.
   (e) remove the edges with count less than min_*sup* from (N)Edge class.
5. end loop.

*(N)Edge: represents the TwoEdge, ThreeEdge, FourEdge and FiveEdge classes etc.

Candidate generation : this process is same as the subgraph construction described in section 3. First time the SingleEdge class is combined with itself. In subsequent iterations it is combined with TwoEdge, ThreeEdge and FourEdge classes as we are running the loop until size-5 subgraphs are generated.

Frequency counting and Pruning: The subgraphs from the Subgraph_*Distinct*_1 class are searched for frequency counting on the vertex labels, edge labels. The subgraphs which meet the support value (user defined) are stored in a class called Subgraph_1. Edges are retained in the SingleEdge class where there is a matching; all other edges are pruned from the SingleEdge class. An example of pruning would be good here to understand the process. If we consider the Figures 4 and 5, the states are shown before pruning. Let's assume the minimum support provided by the user is 2, i.e. a sub-structure must be appearing in at least two graphs. The states of the SingleEdge and TwoEdge classes after pruning are shown in Fig. 9 and 10 respectively.

5 Experimental Details

The experiments were conducted on a Linux machine with 2 GB memory. The OO-FSG algorithm was coded in Java. The experimental results are shown in Table 6 as well as in graphical format. The graphs in the figures 11, 12, 13 and 14 show the efficiency comparison between DB-FSG and OO-FSG w.r.t minimum supports 1, 3, 5 and 7 respectively. We observed that using db4o database, efficiency is much

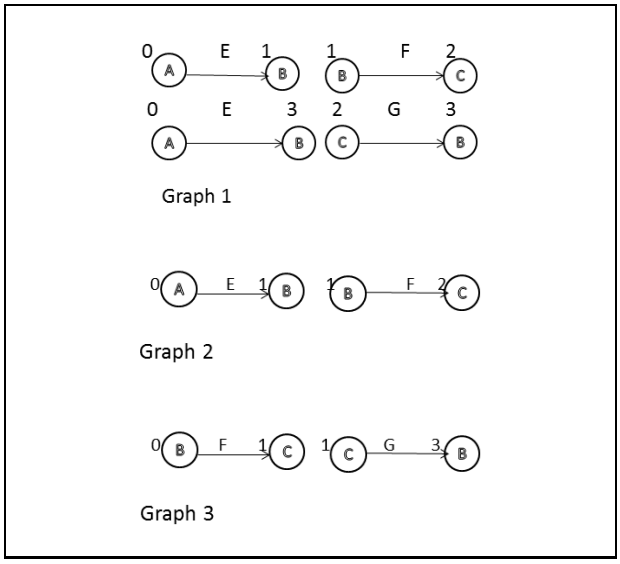

Figure 9: Objects of SingleEdge After Pruning

higher than relational database. Also, scalability of db4o is higher than relational database.

For the comparison of both the methods, we performed the experiments on datasets containing 50,000 to 400,000 graphs. These are transaction graphs. Each graph contains 30-50 edges and 30-50 vertices. Tests were conducted with varying minimum support values 1% , 3%, 5% and 7%. The maximum size of the sub-structures is taken as five. Datasets have millions of records. For example, 50K dataset has approximately 2 million size-1(single) edges. We observed that DB-FSG and OO-FSG performs the same for 50K data (min_*sup*: 1%). Hence, we ignored to evaluate further comparisons with other minimum support values. But as the dataset size grows big, the performance improved dramatically. For datasets of size 100K, the improvement is around 84% and for 400K it is around 230% for minimum support 1%.

Table 6: DB-FSG vs OO-FSG Performance

Dataset size	Min_*sup*	DB-FSG	OO-FSG
50K	1%	357	353
100K	1%	1349	731
100K	3%	1220	656
100K	5%	1061	563
100K	7%	827	484
200K	1%	2439	1331
200K	3%	2002	1206
200K	5%	1717	1117
200K	7%	1622	1030
300K	1%	5887	2221
300K	3%	5394	2141
300K	5%	5137	2019
300K	7%	4164	1863
400K	1%	9502	2879
400K	3%	8228	2457
400K	5%	7156	2426
400K	7%	6962	2313

Note: All run-times in the DB-FSG and OO-FSG columns are shown in seconds.

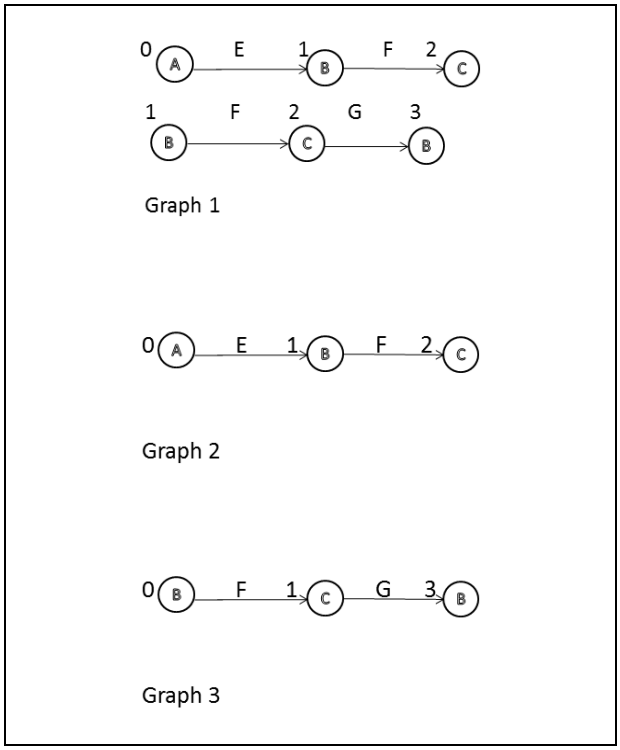

Figure 10: Objects of TwoEdge After Pruning

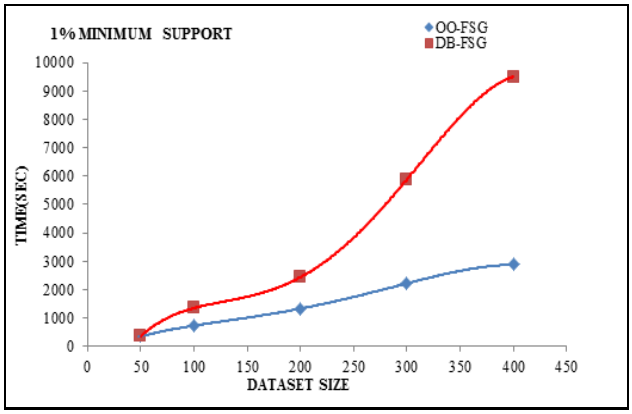

Figure 11: Comparison with 1% minimum support

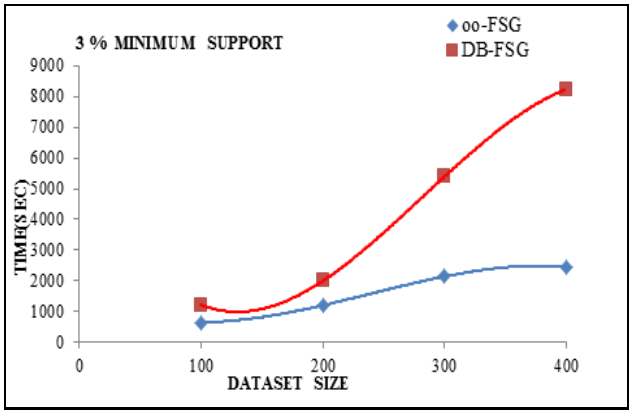

Figure 12: Comparison with 3% minimum support

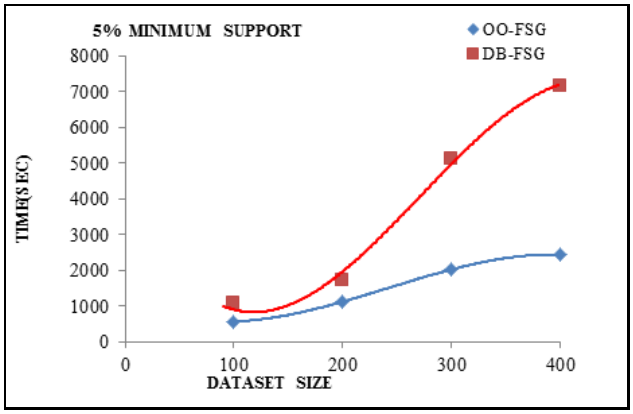

Figure 13: Comparison with 5% minimum support

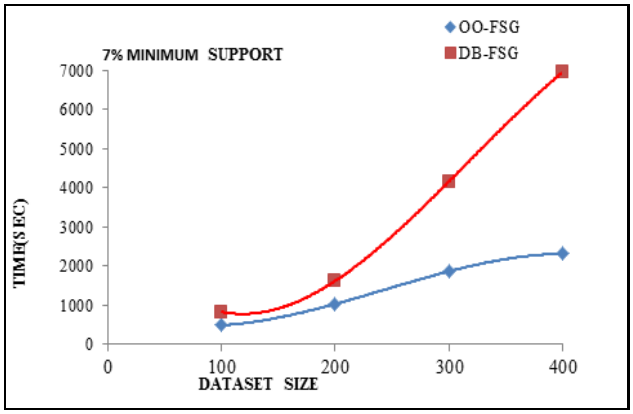

Figure 14: Comparison with 7% minimum support

6 Conclusion and Future Work

In this paper we proposed a new approach for mining frequent subgraphs by storing the graph data sets in an object-oriented database, db4o. Using db4o, large graph data sets can be stored, thus eliminating the constraint of memory resident graph data sets. Also, db4o overcomes the space constraints of relational databases. Furthermore, retaining the semantic information is an added advantage of db4o. To the best of our knowledge, our method is the fastest amongst the entire frequent subgraph mining methods so far. Currently, we are implementing our algorithm on both directed and undirected graph data sets. Our work is in progress to resolve the graph isomorphism problem and storing the ontology in undirected graphs (which this paper does not support currently) in an efficient manner.

7 Acknowledgments

This research has been supported by The Molecular Basis of Disease Program, Georgia State University, USA.

References

Cook, D.J., Holder, L.B. & (2000), Graph-based data mining. *in* 'IEEE Intelligent Systems', 15(2), 32-41.

Inokuchi, A., Washio, T. & Motoda, H. (2003), Complete mining of frequent patterns from graphs. *in* 'Mining graph data. Mach. Learn.', 50(3).

Kuramochi, M., Karypis, G. & (2001), Frequent subgraph discovery. *in* ' ICDM 2001: Proc. of the 2001IEEE International Conference on Data Mining, Washington, DC, USA.', IEEE Computer Society, Los Alamitos pp. 313-320.

Han, J., Yan, X. & (2002), gSpan: Graph-based substructure pattern mining. *in* 'ICDM 2002:Proc. of the 2002 IEEE Int. Conf. on Data Mining', pp. 721-731.

Kuramochi, M., Karypis, G. & (2005), Finding frequent patterns in a large sparse graph. *in* 'Data Min. Knowl. Discov.' 11 (3) (2005) 243-271.

Jiang, X., Xiong, H. & Wang, C., Tan, A. (2009), Mining Globally Distributed Frequent Subgraphs in A Single Labeled Graph. *in Data and Knowledge Discovery*, 68(10), pp: 1034-1058 2209.

Chakravarthy, S., Beera, R. & Balachandran, R. (2004), Database approach to graph mining. *in* 'PAKDD Proceedings', Sydney, pp. 341-350.

Chakravarthy, S., Pradhan, S. & (2008), DB-FSG: An SQL-Based Approach for Frequent Subgraph Mining. *in* 'DEXA 2008', LNCS 5181, pp. 684-692.

Agrawal, R., Srikant, R. & (1994),Fast algorithms for mining association rules. *in* 'J. B. Bocca, M. Jarke, and C. Zaniolo, editors, Proc. of the 20th Int. Conf. on Very Large Databases (VLDB)', pages 487-499. Morgan Kaufmann.

Agarwal, R. C., Aggarwal, C. C. & Prasad, V. V. V., Crestana, V. (1998), A tree projection algorithm for generation of large itemsets for association rules. *in* 'IBM Research Report RC21341'.

Zaki, M. J., Gouda, K. & (2001),Fast vertical mining using diffsets. *in* 'Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute'.

Jiang, X., Xiong, H. & Wang, C., Tan, A. (2000), Mining frequent patterns without candidate generation. *in* 'Proc. of ACM SIGMOD Int. Conf. on Management of Data', 68(10), Dallas, TX.