DB-FSG: An SQL-Based Approach for Frequent Subgraph Mining*

Sharma Chakravarthy and Subhesh Pradhan

IT Laboratory & Department of Computer Science and Engineering
The University of Texas at Arlington, Arlington, TX 76019
{sharma@cse.uta.edu,subhesh_kp}@yahoo.com

Abstract. Mining frequent subgraphs (FSG) is one form of graph mining for which only main memory algorithms exist currently. There are many applications in social networks, biology, computer networks, chemistry and the World Wide Web that require mining of frequent subgraphs. The focus of this paper is to apply relational database techniques to support frequent subgraph mining. Some of the computations, such as duplicate elimination, canonical labeling, and isomorphism checking are not straightforward using SQL. The contribution of this paper is to efficiently map complex computations to relational operators. Unlike the main memory counter parts of FSG, our approach addresses the most general graph representation including multiple edges between any two vertices, bi-directional edges, and cycles. Experimental evaluation of the proposed approach is also presented in the paper.

1 Introduction

Frequent subgraphs (FSG) is one form of graph mining. However, for FSG mining there currently exist only main memory algorithms [4]. There are many applications in social networks, biology, computer networks, chemistry and the World Wide Web that require mining of frequent subgraphs over large data sets. These main memory algorithms do not scale very well for large data sets. Hence, there is a need for developing scalable algorithms for frequent subgraph mining. An SQL-based approach [9,5] is one way of doing that by exploiting the buffer management and optimization techniques already provided and fine tuned in a RDBMS. However, applying limited representation and computations provided by a RDBMS for graph mining is not trivial. Representation of a graph, generation of larger subgraphs, checking for exact and inexact matches of subgraphs using relational representation and operators is one of the contributions of this paper.

The remainder of the paper is organized as follows. The different graph mining algorithms that motivated the development of a SQL-based approach for frequent subgraph mining is discussed in section 2. An overview of DB-FSG algorithm

* This work was supported, in part, by Air Force grant F30602-01-2-0570 and NSF (grants EIA-0216500, IIS-0326505, MRI 0421282, and IIS 0534611).

for FSG is provided in section 3. The design issues related to DB-FSG algorithm is detailed in section 4. Experimental results are discussed in section 5. Finally, conclusions and future work are discussed in section 6.

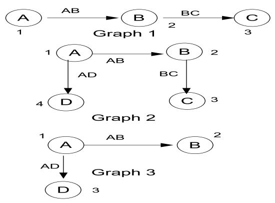
2 Related Work

Subdue [2] is one of the earliest graph mining algorithms that detects the best substructure using the minimum description length principle [8]. It also mines for interesting concepts, detect anomalies, and similarities between graph structures. FSG [4] and others [10,3] are main memory algorithms that mine graph sets to discover frequent subgraphs. FSG uses canonical labeling to determine subgraph isomorphism. It considers an undirected graph representation without multiple edges (between two vertices) or cycles. Hence, FSG cannot mine over general forms of directed graphs, graphs with multiple edges, and cycles. gSpan [11] is another frequent subgraph mining approach which uses depth first search and generates lesser candidate items than FSG. The depth-first traversal and book keeping requires special data structures and is not clear how it can be mapped using relational operators.

DB-Subdue [1] and HDB-Subdue [6] (SQL-based versions of Subdue) detect interesting subgraphs that compress a graph (or a forest) maximally using the minimum description length (or MDL) principle. HDB-Subdue handles multiple edges, cycles, and hierarchical reduction to deal with a general graph. However, HDB-Subdue did not support mining over a set of input graphs to discover frequent subgraphs.

3 Overview of DB-FSG

Normally, graphs are represented as a set of edges and vertices. DB-FSG represents graphs using two relations: i) a vertex table and ii) an edge table which store the vertices and the edges of the graph, respectively. For the set of graphs shown in Figure 1(a), the corresponding *vertex* and *edge* tables are shown in Figures 1(b) and 1(c), respectively. Graph Id (in short GID) attribute in the tables helps to identify the edges and vertices belonging to the same graph.



(a) Example Graph

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

(b) Vertex Table

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

(c) Edge Table

Fig. 1. Sample Graph and Corresponding Vertex and Edge Table

Vertex 1	Vertex 2	Edge No.	Edge Label	Vertex 1 Name	Vertex 2 Name	Graph Id
1	2	1	AB	A	B	1
2	3	2	BC	B	C	1
1	2	3	AB	A	B	2
2	3	4	BC	B	C	2
1	4	5	AD	A	D	2
1	2	6	AB	A	B	3
1	3	7	AD	A	D	3

Fig. 2. Oneedge Table

As the edge table does not contain information about vertex labels, tuples of edge table cannot represent substructures of size one. Hence, we create a new relation called *oneedge* by joining the vertex and the edge tables as shown in Figure 2. The *oneedge* table will contain all the instances of substructures of size one as tuples. For a one-edge substructure, the edge direction is always from the first vertex to the second vertex. Hence, there are no attributes in the *oneedge* table which specify the direction. For a higher edge substructure, we introduce connectivity attributes to denote the direction of edges between the vertices of the substructure. The *oneedge* table is the base table that will be used for generating higher size substructures. For each edge in the *oneedge* table, we assign a unique identifier called the edge number.

We need to systematically generate subgraphs of increasing size in all the input graphs and obtain the count for the isomorphic substructures across the graphs. To expand a one-edge substructure to a two-edge substructure, we join *oneedge* relation with itself on matching vertices. To ensure that the expansion is done within the same graph, we impose a constraint that the GID (each graph has an id termed GID) of both one-edge substructures should be same. We term the resulting two-edge substructure table as instance_2. In general, substructures of size i are generated by joining instance_(i-1) relation with *oneedge* relation. In order to avoid expansion of instances on edges that are already present (remember that our approach unlike FSG handles multiple edges and cycles), we impose the rule that the new edge being added should not have the same edge number as the edge already present in the substructure instances. In case of substructures that have two or more edges, we would need attributes to denote the direction of the edges. The From and To (F and T for short) attributes in the instance_n table serve this purpose. An n-edge substructure is represented by n+1 vertex numbers, n+1 vertex labels, n edge numbers, n edge labels, and n From and To pairs. In general, 6n+3 attributes are needed to represent an n-edge substructure. Though edge numbers are part of every instance_n table, owing to the space constraint, we will be showing it only in sections where they are necessary.

4 Details Of DB-FSG Approach

DB-FSG algorithm is shown below. Some of the major aspects - *new substructure instance representation*, *unconstrained expansion and pseudo duplicate* *elimination*, *canonical label ordering* and *frequency counting and substructure pruning* are eleborated further. For a comprehensive description, refer to [7].

Algorithm 1. DB-FSG Algorithm

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

The algorithm starts by creating one-edge substructure instance by joining vertex table and edge table as shown in the step 1 of algorithm 1. The *oneedge* instances with frequency less than the user specified support value are pruned. The remaining one edge instances are expanded to two-edge instances and the two-edged substructure instances having frequency less than the support value are pruned. As shown in steps 3 to 11 of the algorithm 1, the expansion and pruning of sub-graphs continues till user specified MaxSize is reached or until the subgraphs cannot be expanded any further. Due to the unconstrained expansion, the same substructure may be generated in many ways. Hence, pseudo duplicate elimination is required to remove such duplicates as mentioned in step 5 of the algorithm 1. Also, due to unconstrained expansion similar substructure instances may be generated in different order. Hence, canonical ordering is performed in step 6 of the algorithm 1 to identify such substructures instances. Similarly, to get the correct frequency of the substructures, substructure counting and pruning is done in steps 7, 8 and 9 of the algorithm 1.

DB-FSG [7] represents a substructure as a tuple of a relation. The repeating vertex number of a cycle or a multiple edge is marked by '0' and the corresponding vertex label is marked by '-', respectively. The marking of repeating vertices avoids redundant expansion on the same vertex. This form of representation can represent most general forms of a graph including cycles and multiple edges.

However, this representation is not sufficient to represent a set of graphs. For example, this representation cannot represent a set of graphs shown in In DB-FSG, we need to distinguish between graphs in which the same substructure

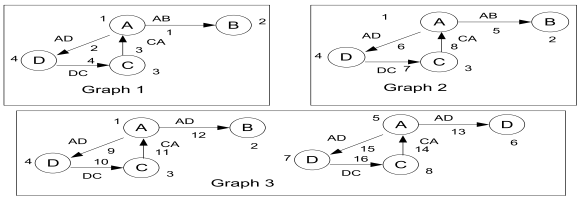


Fig. 3. DB-FSG graph representation

appears. Hence, we have added one more attribute (Graph ID or GID) to denote which graph a substructure belongs to. Each graph is assigned a unique GID and all the substructures belonging to same graph will have the same GID. New substructure instance representation of size two for Figure 3 is shown in table 1.

Table 1. DB-FSG instances

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

Unconstrained expansion generates all possible substructures in an arbitrary graph input. However, this unconstrained expansion also results in the same instance to be generated in different order (will be termed pseudo duplicates). In order to identify same instances that grew in different order, we have implemented pseudo duplicate elimination by constructing an edge code that is unique to an instance. We have introduced a new attribute called ecode in instance_n table. This attribute will store edge code of each instance in the table. Then by comparing ecodes, we can identify and remove pseudo duplicates more efficiently. Details can be found in [7].

4.1 Canonical Ordering

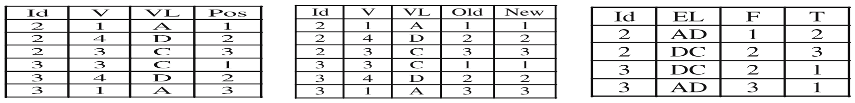
In order to identify two similar substructure instances, vertex labels and the connectivity attributes need to be used (unlike vertex numbers or edge numbers for pseudo duplicate elimination). If two instances have same vertex labels and edge directions, then they can be identified as similar (or isomorphic) instances. In SQL, we can identify similar substructures only if the vertex labels and connectivity map of each tuple is canonically ordered. Since databases do not allow rearrangement of columns (only rows by using group by and order clauses), to obtain canonical ordering, we have to transpose the rows of each substructure into columns, sort and reconstruct them to get the canonical order. To facilitate construction of canonically ordered instance_n table, we introduce an additional attribute called *ID* in unordered instance_n table. Each instance (tuple) in the instance_n table should have unique ID for which rownum is used as ID value. Table 2 shows instance_2 table for Fig 3 before canonical ordering.

Owing to the table space constraints, canonical ordering of only the second and third instance are shown below. We project the vertex numbers and vertex names from the instance table and insert them row wise into a relation called unsorted as shown in Fig 4(a). We also include the position in which the vertex occurs in the original instance. To differentiate between the vertices of different instances, we

Table 2. Before Canonical Ordering

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

carry the Id value from the instance table onto the unsorted table. Next, we sort the table on Id and vertex label and insert it into a table called Sorted as shown in Fig 4(b) with its New attribute pointing to the new position of the vertex and the attribute Old pointing to the old position of the vertex. Similarly, the



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

(a) Unsorted Table

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

(b) Sorted Table

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

(c) Old_Ext Table

Fig. 4. Canonical Ordering Intermediate Tables

connectivity attributes are also transposed into a table called Old_Ext as shown in Fig 4(c). Since the sorting on vertex numbers has changed its position, we need to update the connectivity attributes to reflect this change. Therefore, we do a 3 way join of Sorted and Old_Ext tables on the Old attribute of the Sorted table to get the updated connectivity attributes which we call New_Ext as in Tab 3. Next, we sort the New_Ext table on Id and the attributes F (From vertex) and T (Terminating vertex). Since, we also need ecode and GID attributes for

Table 3. New_Ext

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

Table 4. Sorted_Ext

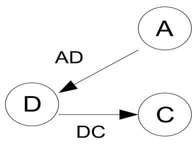
Id	EL	F	T
2	AD	1	2
2	DC	3	2
4	AD	1	3
4	DC	3	2

expansion of the instances, the ecode and GID attribute were also transposed to tables called label_ecode_n and label_GID_n. To differentiate between the GID and ecode of different instances, we carry the Id value from the instance table onto the respective tables.

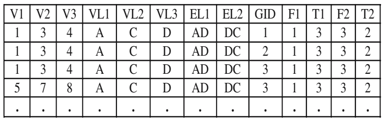
Now, we have the ordered vertex as well as connectivity map tables. Hence, we can do a *2n+3* way join (where *n* is current substructure size) of *n+1* Sorted tables, label_GID_n table, label_ecode_n table and *n* Sorted_Ext tables to reconstruct the original instance in a canonical order. Table 5 shows the substructures after canonically ordering vertex numbers and connectivity attributes.

Table 5. Instance table - After canonical ordering

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



(a) Graph with multiple instances



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

(b) Instance table with multiple instances

Fig. 5. Multiple Instances of same substructure

4.2 Frequency Counting and Substructure Pruning

A graph may have many instances of the same substructure. For example, if we consider substructure in Fig 5(a), it has two instance in graph 3 of Fig 3. Fig 5(b) shows the instances of substructure in Fig 5(a). If we count the frequency of the substructure from the instance table, it will give a count of four. Even though, the correct frequency count across the graph set is three. Hence, to obtain the correct frequency of a substructure in the graph set, we need to include only one instance per substructure within a graph. However, we need to preserve all instances of a substructure that satisfy the support condition for future expansion. In order to get one instance per substructure of size n, we project *distinct* vertex labels, edge labels, connectivity map and GID and store it into dist_n table. Then, a GROUP BY operation on vertex labels, edge labels and connectivity map in dist_n table will provide the correct frequency of each substructure. Since, the substructures having less frequency than support value will not contribute to future expansion, we can store only those substructures that satisfies the support value in sub_fold_n table. Then, we can prune the instance_n table by removing instances of substructures that are not in sub_fold_n table.

5 Experimental Analysis

The experiments were conducted on a Linux machine (running on dual processors with 2.4 GHz CPU speed and 2 GB memory) of the Distributed and Parallel Computing Cluster at UTA (DPCC@UTA) using Oracle 10g.

For the comparison of DB-FSG with FSG, we performed experiments on data sets containing 100K - 300K graphs, with each graph containing 30 to 50 edges and 30 to 50 vertices. The support value was set to 1% and the maximum substructure size (MaxSize) was set to 5. When we tried to compare DB-FSG with FSG, it was observed that FSG was not able to detect all the frequent patterns in the input graphs and failed to detect all the embedded subgraphs. Also, FSG does not handle multiple edges and cycles. Hence, the results are not discussed here. Other main memory FSG systems

Other set of experiments were performed to analyze the performance of the DB-FSG algorithm for different data sets, for various support values, and for different types of graphs (that is, simple graph without cycles and multiple edges, graph with cycles, and graph with multiple edges). Each graph in the data set has 40 edges and 40 vertices. The data sets are varied from 50K of graphs (2 million vertices and 2 million edges in the data set) to 300K of graphs (12 million vertices and 12 million edges in the data set). The frequent substructures embedded in the data set had support value of 3% and 4%. The parameters used for the set of experiments were MaxSize - 5, Support - 1%. Fig 6(a) gives the

	Time (seconds)		
Graph \ Type	Simple	Cycles	Multiple Edges
50K	391.23	431.74	560.8
100K	1510.4	1516.365	1735.8
150K	2572.61	2313.04	2639.49
200K	3680.08	3233.4	3535.39
250K	4663.78	4387.89	4590.78
300K	5692.28	5297.8	5604.93

(a) Performance of DB-FSG on different graphs

	Time (seconds)		
Support \ Graph	100K	200K	300K
1%	1892.89	3912.11	6088.9
3%	1679.05	3697.68	5722.07
5%	1516.64	3280.12	5374.15
7%	1064.64	2224.03	4323.82

(b) Performance of DB-FSG for varying support

Fig. 6. Summary of Experiments

processing time required by DB-FSG on data sets containing graphs without cycles and multiple edges, graphs with cycles and graphs with multiple edges. The experimental results showed that the processing time of algorithm increases linearly as the size of the data set grows. The number of substructures instances discovered in data set containing graph with cycles are lesser than the graphs without cycles and graphs with multiple edges. Hence, the processing time of the data sets containing cycles was less than the graphs without cycles and graphs with multiple edges.

Then, we conducted experiments to analyze the performance of the algorithm for varying support value. The frequent substructures embedded in the data sets had support value of 1%, 3%, 5% and 7%. We varied the support value from 1% to 7% (keeping the MaxSize as 5) in order to evaluate the performance of the DB-FSG on those data sets. Figure 6(b) gives shows the relation of support value with the processing time. The experimental results showed that the processing time decreased as the support value increased. For greater support value, more substructures will be pruned in earlier iterations of the algorithm. Hence, less processing time is required. The number of substructure instances retained for 7% of support value will be lesser than the number of instances retained for support value of 1% in each expansion iteration. Hence, the processing time for each steps DB-FSG like substructure expansion, pseudo duplicate elimination, canonical ordering and substructure counting and pruning will require lesser time for user defined support value of 7% than for 1%.

6 Conclusions

In this paper, for the first time, we have applied relational database approach for frequent subgraph mining. The graph representation used in this paper can represent the most general form of graph including graphs with cycles and multiple edges (between two vertices). Our approach addresses all aspects of frequent subgraph mining – from candidate generation to pseudo duplicate elimination to canonical ordering – all using standard SQL. Our experimental results show that this approach is highly scalable for very large data sets whereas main memory approaches are likely to fail. Currently, we are further optimizing the efficiency of the algorithm.

References

1. Chakravarthy, S., Beera, R., Balachandran, R.: Database approach to graph mining. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 341–350. Springer, Heidelberg (2004)
2. Cook, D.J., Holder, L.B.: Graph-based data mining. IEEE Intelligent Systems 15(2), 32–41 (2000)
3. Inokuchi, A., Washio, T., Motoda, H.: Complete mining of frequent patterns from graphs: Mining graph data. Mach. Learn. 50(3) (2003)
4. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: ICDM 2001: Proc. of the 2001 IEEE International Conference on Data Mining, Washington, DC, USA, pp. 313–320. IEEE Computer Society, Los Alamitos (2001)
5. Mishra, P., Chakravarthy, S.: Performance evaluation and analysis of k-way join variants for association rule mining. In: James, A., Younas, M., Lings, B. (eds.) BNCOD 2003. LNCS, vol. 2712, pp. 95–114. Springer, Heidelberg (2003)
6. Padmanabhan, S.: HDB-Subdue, a relational database approach to graph mining and hierarchial reduction. Master's thesis, CSE Dept., U T Arlington (2004)
7. Pradhan, S.: A Relational Database Approach to Frequent Subgraph (FSG) Mining. Master's thesis, The University of Texas at Arlington (August 2006), `http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Pra06MS.pdf`
8. Rissanen, J.: Stochastic Complexity in Statistical Inquiry Theory. World Scientific Publishing Co., Singapore (1989)
9. Sarawagi, S., Thomas, S., Agrawal, R.: Integrating mining with relational database systems: Alternatives and implications. In: SIGMOD Conference, pp. 343–354 (1998)
10. Washio, T., Motoda, H.: State of the art of graph-based data mining. SIGKDD Explor. Newsl. 5(1), 59–68 (2003)
11. Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: ICDM 2002: Proc. of the 2002 IEEE Int. Conf. on Data Mining, pp. 721–731 (2002)