ISSN No. 0976-5697

Volume 4, No. 1, January 2013 (Special Issue)

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Query Processing in Graphical Data

Santosh Kumar Sahu*
Department of Information Technology
Rajiv Gandhi College of
Engineering & Research
sksahu.ngp.india@gmail.com

Mujeeb Rahaman
Department of Computer Engineering
St. Vincent Pallotto College of Engineering & Tehnology
Nagpur, India
Mujeebrahman1984@gmail.com

Abstract: Graph data are becoming increasingly more ubiquitous in today's networked world. Examples include social networks such as MySpace and Facebook as well as cell phone networks and blogs. The network routing across the Internet is another example of graph data, as is the hyperlinked structure of the World Wide Web (WWW). Bioinformatics, especially systems biology, deals with understanding interactions networks between various types of biomolecules, such as protein-protein interactions, metabolic networks, gene networks, and so on. Another example comes from semi-structured data, say in the form of XML documents. Given a graph query, it is desirable to retrieve graphs quickly from a large database via graph-based indices. We are using of frequent substructure as the basic indexing feature. Frequent substructures are ideal candidates since they explore the intrinsic characteristics of the data and are relatively stable to database updates. To reduce the size of index structure, we used techniques, size-increasing support discriminative fragments.

Keywords: Graph indexing, frequent fragments, discriminative fragments, gSpan, index construction

I. INTRODUCTION

The goal of graph mining is to extract interesting subgraphs from a single large graph (e.g., the WWW), or from a database of many graphs. In different applications we may be interested in different kinds of subgraph patterns, such as subtrees, complete graphs or cliques, bipartite cliques, dense subgraphs, and so on. These may represent, for example, communities in a social network, hub and authority pages on the WWW, a protein modules involved in similar biochemical functions, and so on. Often, we may want to mine all the frequent subgraphs that appear in a database, without specifying a particular type of subgraph of interest.A graph is a pair G = (V,E) where V is a set of vertices, and E is a set of edges, where an edge is an unordered pair of distinct vertices. A graph G'= (V',E') is said to be a subgraph of G if V' Є V and E' Є E. In many applications of data mining, we are specifically interested in only connected subgraphs, i.e., when V' Є V, E' Є E, and for any x, y Є V', there exists a path from x to y in G'.

A. *Graph isomorphism:*

Find a mapping f of the vertices of G1 to the vertices of G2 such that G1 and G2 are identical; i.e. (x,y) is an edge of G1 iff (f(x),f(y)) is an edge of G2. Then f is an isomorphism, and G1 and G2 are called isomorphic[3,4].
□□ No polynomial-time algorithm is known for graph isomorphism.
Neither is it known to be NP-complete.

B. *Subgraph isomorphism:*

Subgraph isomorphism asks if there is a subset of edges and vertices of G1 that is isomorphic to a smaller graph G2. Subgraph isomorphism is NP-complete. The formatter will need to create these components, incorporating the applicable criteria that follow.

II. SUBGRAPH MINING

To mine all the frequent subgraphs, we have to search over the space of all possible graph patterns, which is enormously large. For example, if we consider graphs with m vertices, then there are

$$\binom{m}{2} = O(m^2)$$

Possible edges. The number of possible subgraphs with m nodes is then 2O(m2), since we may either decide to include to exclude each of the O(m2) edges. Many of these will not be connected, but we can still use this worst case bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more. Assume that |ΣV | = | Σ E | = *l*, then there are Im possible ways to label the vertices and there are *I*m2 ways to label the edges. Thus the number of possible labeled subgraphs with m vertices is

$$2^{O(m2)}l^{m}l^{m2} = O((2 \cdot l)^{m2})$$

This is the worst case pessimistic bound, since many of these subgraphs will be isomorphic to each other, so the number of distinct subgraphs will be much less[3].

A. *Select Subgraph Mining Algorithms:*

For Subgraph Mining, there are two types of approach:

Apriori-based approach
o FSG
Pattern growth approach
o SUBDUE
o gSpan

a. Apriori-based approach: The Apriori algorithm uses prior knowledge of frequent itemsets to generate the candidates for larger frequent itemsets[3]. It relies on relationships between itemsets and subsets. If an itemset is frequent, then all of its subsets must also be frequent. But generating candidates and checking their support at each level of iteration can become costly. The Apriori Algorithms an influential algorithm for mining frequent item sets for Boolean association

CONFERENCE PAPER
National Level Conference on
"Trends in Advanced Computing and Information Technology (TACIT - 2012)"
on 2nd September 2012
Organized by
St. Vincent Pallotti College of Engineering and Technology, Nagpur, India

rules. Most algorithms follow the general principle of the Apriori algorithm for association rule mining.

(a). *FSG:* Kuramochi and Karypis [2001] propose the FSG algorithm which also has the same flavor: starting with frequent subgraphs of 1 and 2 nodes, it successively generates larger subgraphs which still occur frequently in the graph. The algorithm expects a graph with colored edges and nodes; our graphs are a special case where all nodes and edges have only one color. However, it also needs to solve the graph and subgraph isomorphism problems repeatedly, and this is very slow and inefficient for graphs with only one color.

Challenges of Apriori-based approach: The Apriori-like algorithms meet two challenges:

***a) Candidate Generation*:** The generation of size & subgraph candidates from size 6 frequent subgraphs is more complicated and costly than that of itemsets.

b) Pruning false positives: Subgraph isomorphism test is an NP complete problem, thus pruning false positives is costly.

b. Pattern growth approach: Pattern growth introduces a different approach here. Instead of generating the candidates, it compresses the database into a compact tree form, known as the FP-tree, and extracts the frequent patterns by traversing the tree.

a) SUBDUE is a graph-based knowledge discovery system that finds structural, relational patterns in data representing entities and relationships. SUBDUE represents data using a labeled, directed graph in which entities are represented by labeled vertices or subgraphs, and relationships are represented by labeled edges between the entities. SUBDUE uses the minimum description length (MDL) principle to identify patterns that minimize the number of bits needed to describe the input graph after being compressed by the pattern. SUBDUE can perform several learning tasks, including unsupervised learning, supervised learning, clustering and graph grammar learning. SUBDUE has been successfully applied in a number of areas, including bioinformatics, web structure mining, counter-terrorism, social network analysis, aviation and geology.

i. Disadvantage of SUBDUE:

Compression is lossy from the point of view of minimal description length (MDL), this is not very nice length.

Not well in line with existing, well--studied, graph grammars studied, grammars.

a) gSpan: Unlike FSG, gSpan discovers frequent substructures without candidate generation and false positives pruning. It builds a lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan adopts the depth first search strategy to mine frequent connected subgraphs[1]. gSpan, which targets to reduce or avoid the challenges of Apriori based approaches and overcome to disadvantage of SUBDUE. If the entire graph dataset can not in main memory, gSpan can be applied directly; otherwise, one can first perform graph based data projection as in and then apply gSpan. To the best of our knowledge, gSpan is the first algorithm that explores depth first search (DFS) in frequent subgraph mining. Two techniques, DFS lexicographic order and minimum DFS code, are used here, which form a novel canonical labeling system to support DFS search. gSpan discovers all the frequent subgraphs without candidate generation and false positives pruning. It combines the growing and checking of frequent subgraphs into one procedure, thus accelerates the mining process.

b) DFS Subscripting: When performing a depth first search [3] in a graph, we construct a DFS tree. One graph can have several different DFS trees. For example, graphs in Fig.1(b), Fig.1(c) and Fig.1(d) are isomorphic to that in Fig.1(a). The thickened edges in Fig.1(b)-(d) represent three different DFS trees for the graph in Fig.1(a). The depth first discovery of the vertices forms a linear order. We use subscripts to label this order according to their discovery time [3]. i < j means vi is discovered before vj . We call v0 the root and v1the rightmost vertex. The straight path from v0 to vn is named the rightmost path. In Fig.1(b)-(d), three different subscripting are generated for the graph in Fig.1(a). The right most path is (v0, v1, v4) & in Fig.1(b), (v0, v4) in Fig.1(c), and (v0, v1, v2, v4) in Fig.1(d). We denote such subscripted G as GT.

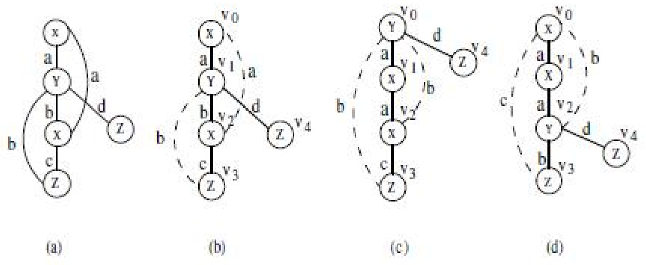


Figure 1. Depth First Search Tree.

Table I. 1 DFS code for Figure.1(b) – (d)

Edge	Fig.1(b) α	Fig.1(c) β	Fig.1(c) γ
0	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, X, a, X)
1	(0, 2, Y, b, X)	(1, 2, X, a, X)	(1, 2, X, a, Y)
2	(2, 0, X, a, X)	(2, 0, X, b, Y)	(2, 0, Y, b, X)
3	(2, 3, X, c, Z)	(2, 3, X, c, Z)	(2, 3, Y, b, Z)
4	(3, 1, Z, b, Y)	(3, 0, Z, b, Y)	(3, 0, Z, c, X)
5	(1, 4, Y, d, Z)	(0, 4, Y, d, Z)	(2, 4, Y, d, Z)

Given a graph G, we perform a depth first search (DFS) over its vertices, and create a DFS spanning tree, i.e., one that covers or spans all the vertices, which was obtained by starting at v1 and then choosing the vertex with the smallest index at each step. Edges that are included in the DFS tree are called forward edges, and all other edges are called backward edges. Backward edges create cycles in the graph.

c) DFS Code Lexicographic:

We write

$$\langle v_i, v_j, L(v_i), L(v_j), L(v_i v_j)\rangle < \langle v_x, v_y, L(v_x), L(v_y), L(v_x v_y)\rangle$$

i) $\langle v_i, v_j\rangle <_e \langle v_x, v_y\rangle$, or

ii) $\langle v_i, v_j \rangle = \langle v_x, v_y \rangle$ and $\langle L(v_i), L(v_j), L(v_iv_j) \rangle <_l \langle L(v_x), L(v_y), L(v_xv_x) \rangle$

If either:

Here <e is an ordering on the edges and <*l* is an ordering on the vertex and edge labels. The label order <*l* is the standard lexicographic order on the vertex and edge labels. For example the label tuple <a,a,r> <*l* <a,b,q> since if we compare the two tuples in an element-wise manner, we find that L(vj) = a < b = L(vy). The edge order <e is more involved. Let ei j = (vi,vj) and exy = (vx,vy). We write eij <e exy if the following conditions are met:

i) If eij and exy are both forward edges, then a) j < y or, b) j
= y and i > x.

ii) If eij and exy are both backward edges, then a) i < x or b) i = x and j < y.

iii) If eij is a forward and exy is a backward edge, then i <
y.

iv) If eij is a backward and exy is a forward edge, then j *
x.

Given the DFS codes for any two graphs, we can compare them tuple by tuple to check which is smaller.

III. FREQUENT FRAGMENT

Given a graph database D, |Dg| is the number of graphs in D where g is a subgraph. |Dg| is called (absolute) support, denoted by support(g). A graph g is frequent if its support is no less than a minimum support threshold, minSup. As one can see, frequent graph is a relative concept. Whether a graph is frequent depends on the setting of minSup[2]. We use the term "fragment" to refer to a small subgraph (i.e., substructure) existing in graph databases and query graphs. Figure 3 shows two frequent fragments in the sample database with minSup = 2.

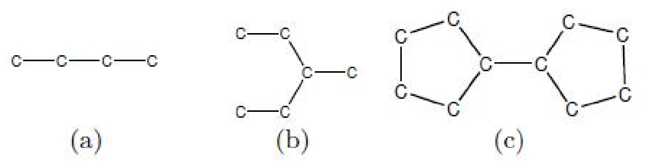


Figure 2. A Sample Database

any graph with q embedded must contain q's sub-graphs, Df is a candidate answer set of query q. If minSup is low, it is not expensive to verify the small number of graphs in Df in order to find the query answer set. Therefore, it is feasible to index frequent fragments for graph query processing[4].

IV. DISCRIMINATIVE FRAGMENT

Do we need to index every frequent fragment? Let's have some analysis. If two similar frequent fragments, f1 and f2, are contained by the same set of graphs in the database, i.e., Df1 = Df2 , it is probably wise to include only one of them in the feature set[2]. Generally speaking, among similar fragments with the same support, it is often sufficient to index only the smallest common fragment since more query graphs may contain the smallest fragment. That is to say, if f', a supergraph of f, has the same support as f, it will not be able to provide more information than f if both are selected as indexing features. Thus f' should be removed from the feature set. In this case, we say f' is not more discriminative than f.

All the graphs in the sample database (Figure 2) contain carbon-chains: c, c-c, c-c-c, and c-c-c-c. Fragments c-c, c-c-c, and c-c-c-c do not provide more indexing power than fragment c. Thus, they are useless for indexing.

Let us examine the query example in Fig.4. As shown below, carbon chains, c - c, c - c - c, and c - c - c - c, are redundant and should not be used as indexing features in this dataset. The carbon ring (Fig.5(c)) is a discriminative fragment since only graph (c) in Figure 2 contains it while graphs (b) and (c) in Figure 2 have all of its subgraphs. Fragments (a) and (b) in Fig.5 are discriminative too.

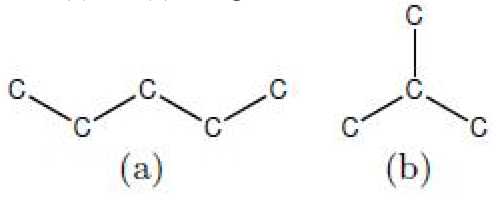


Figure 3. Frequent fragment

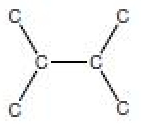


Figure 4. A Sample Query

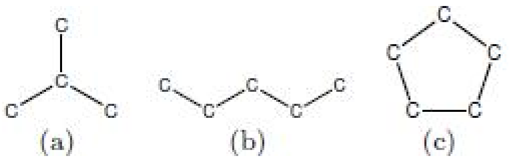


Figure 5. Discriminative Fragment

Frequent fragments expose the intrinsic characteristic of a graph database. Suppose all the frequent fragments with minimum support minSup are indexed. Given a query graph q, if q is frequent, the graphs containing q can be retrieved directly since q is indexed. Otherwise, q probably has a frequent subgraph f whose support may be close to minSup. Since

V. INDEX CONSTRUCTION

Once discriminative fragments are selected, we use efficient data structures to store and retrieve them. Each fragment is associated with an id list: the ids of graphs containing this fragment. We present the details of index construction.

Steps for Index construction:

a. Find the DFS code of each discriminative fragment.
b. Find the each DFS fragments id list associated with them.
c. Convert each DFS fragment into its equal graphic hash value by using **Graphic Hash Code** function. (We can map any graph to an integer by hashing its canonical label).

Graphic Hash Code can help quickly locating fragments in the index structure. 9. *C*q = □ *Si*

Algorithm 1: Index Construction

10. return *C*q

a. ***Input:*** Graph Database *D* (Discriminative fragment with DFS code).

b. **Output:** Index Structure with hash_value and its

associated id list.
- ***c.*** For each Discriminative fragment do
- ***d.*** Convert hash_ value by using Graphic Hash Code
- ***e.*** *P*i = hash_value with it associated id list

/* *P*i is the Data Structure which is store the hash_value and its associated id list */
- ***f.*** end do

VI. HASH CODE GENERATION

Given a sequence hash function h and a graph g, h(dfs(g)) is called graphic hash code. We treat the graphic hash code as the hash value of a graph. If two graphs g and g0 are isomorphic, then h(dfs(g)) = h(dfs(g0)).

Algorithm 2: Graphic Hash Code
- ***a.*** ***Input:*** Graph Database *D* (Discriminative fragment with DFS code).
- ***b.*** ***Output:*** hash_value (unsigned long integer) of each DFS code.
- ***c.*** for each Discriminative fragment do
- ***d.*** hash_value = convert unsigned long to each DFS fragment.
- ***e.*** Store hash_value to appropriate Data Structure
- ***f.*** end do

VII. SEARCHING

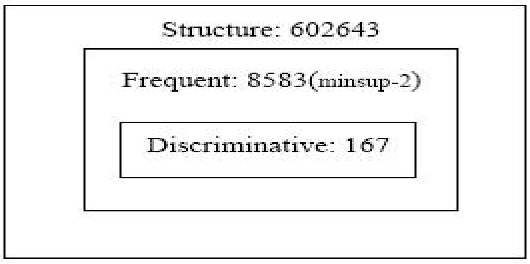
Given a query q, we enumerate all its fragments and locate them in the index. Then it intersects the id lists associated with these fragments.

Algorithm 3: Searching
- ***a.*** ***Input:*** Graph database *D* with id list, Query q.
- ***b.*** ***Output:*** Candidate answer set *C*q.
- ***c.*** let *C*q = NULL
- ***d.*** for each fragment *x* □q do
- ***e.*** if x □*Di*
- ***f.*** *S*i = id list of *D*i /* *S*i is the Data Structure which is store the id list */
- ***g.*** end if
- ***h.*** end do.

VIII. EXPERIMENTAL RESULT

We implement our project work for 60 different types of graph data in Intel core2 Duo processor with 2GB RAM. The entire 60 graph is in above described input format. Each graph stored in different text file, and Entire 60 graphs stored in an another text file.

- ***a.*** Discriminative fragment reduce the index search space. see table:



- ***b.*** Time Comparison between Hash Code and String Matching.

Table: 2

Support	No. of frequent fragment	No. Of Discrinative fragment	String Matching (in seconds)	Hash code (in seconds)
2	8583	167	340	315
3	4374	124	230	220
4	2472	100	150	145
5	1312	80	100	96
6	968	66	95	92

- ***c.*** ***Query Perfection:*** If we work with minimum support then our query result is more accurate.

Table: 3

Search Graph	Found with support 2	Found with support 3	Found with support 4	Found with support 5
1	1	1	1	1
2	2	2	2+3+4	2+3+4
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5+10+14+24+52
6	6	6	6	6
7	7	7	7	7+57
8	8	8	8	8
9	9	9	9	9
10	10	10	10	10

IX. CONCLUSION

Graph indexing plays a critical role at effiient query processing in graph databases which have gained increasing popularity in bioinformatics, Web analysis, and other applications involving complex structures. Previous graph indexing approaches take paths as indexing features and suffer from overly large index size and substantial query processing overhead.

In this paper, we have explored a rather different approach to graph indexing: indexing based on frequent subgraph structures and discriminative subgraph structures. Our performance study shows that our graph indexing method performs better and consumes less space.

X. REFERENCES

[1] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In Proc. 2002 Int. Conf. on Data Mining (ICDM'02), pages 721-724, Maebashi, Japan,Dec. 2002.

[2] X. Yan P. S. Yu and J. Han. Graph Indexing: A frequent structure based approach. In Proc. Of SIGMOD, 2004.

[3] Jiawei Han and Micheline Kamber 2008, Data Mining Concept and Technique. Morgan Kaufmann Publishers.

[4] D.V. Janardhan Rao 2007, A study of Graph mining Algorithms, ppt, Nov 1, 2007.