Subgraph Matching with Set Similarity Using Apache Spark

Nisha yadav1 Anil kumar2
1,2Department of Computer Science and Engineering
1,2MRK Institute of Technology and Management Rewari, Haryana, India

Abstract— Graph databases have been widely used as important tools to model and query complex graph data with the emergence of many real applications such as social networks, Semantic Web, biological networks, and so on, wherein each vertex in a graph usually contains information, which can be modeled by a set of tokens or elements. Liang Hong, Lei Zou, Xiang Lian, Philip S. Yu proposed the method for subgraph matching with set similarity (SMS2) query over a large graph database, which retrieves subgraphs that are structurally isomorphic to the query graph, and meanwhile satisfy the condition of vertex pair matching with the (dynamic) weighted set similarity. To handle the above problem, efficient pruning techniques are proposed by considering both vertex set similarity and graph topology. Frequent patterns are determined for the element sets of the vertices of data graph, and lightweight signatures for both query vertices and data vertices are designed in order to process the SMS2 query efficiently. Based on the determined frequent patterns of elements sets of vertices and signatures, an efficient two-phase pruning strategy including set similarity pruning and structure-based pruning is proposed to facilitate online pruning. Finally, an efficient dominating-set-based subgraph match algorithm guided by a dominating set selection algorithm is proposed to find subgraph matches in order to achieve better query performance. All the techniques are designed for the distributed systems using Apache Spark in order to offer a better price/performance ratio than centralized systems and to increase availability using redundancy when parts of a system fail.

Key words: Graph database, Pruning techniques, Graph topology, Selection algorithm, Data mining

I. INTRODUCTION

With the enormous amount of data stored in files, databases, and other repositories, it is increasingly important, if not necessary, to develop powerful means for analysis and perhaps interpretation of such data and for the extraction of interesting knowledge that could help in decision-making.

Data Mining, also popularly known as Knowledge Discovery in Databases (KDD), refers to the nontrivial extraction of implicit, previously unknown and potentially useful information from data in databases. The automated, prospective analyses offered by data mining move beyond the analyses of past events provided by retrospective tools typical of decision support systems. Data mining tools can answer business questions that traditionally were too time consuming to resolve. They scour databases for hidden patterns, finding predictive information that experts may miss because it lies outside their expectations.

Data mining techniques are the result of a long process of research and product development. Data mining is ready for application in the business community because it is supported by three technologies that are now sufficiently mature:

- Massive data collection
- Powerful multiprocessor computers
- Data mining algorithms

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

A. Scope Of Data Mining:

Data mining derives its name from the similarities between searching for valuable business information in a large database— for example, finding linked products in gigabytes of store scanner data — and mining a mountain for a vein of valuable ore. Both processes require either sifting through an immense amount of material, or intelligently probing it to find exactly where the value resides.

Fig. 1: Architecture of Data Mining System

Given databases of sufficient size and quality, data mining technology can generate new business opportunities by providing these capabilities:

- Automated prediction of trends and behaviors:
  Data mining automates the process of finding predictive information in large databases. Questions that traditionally required extensive hands-on analysis can now be answered directly from the data — quickly. A typical example of a predictive problem is targeted marketing. Automated discovery of previously unknown patterns: Data mining tools sweep through databases and identify previously hidden patterns in one step. An example of pattern discovery is the analysis of retail sales data to identify seemingly unrelated products that are often purchased together.
The most commonly used techniques in data mining are:
- Artificial neural networks
- Decision trees
- Genetic algorithms
- Nearest neighbor method
- Rule induction

B. Graph Theory:
Graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects in mathematics and computer science.

1) Graph: Basic Notation and Terminology:
A graph $G = (V,E)$ in its basic form is composed of vertices and edges. $V$ is the set of vertices (also called nodes or points) and $E \subseteq V \times V$ is the set of edges (also known as arcs or lines) of graph $G$. The difference between a graph $G$ and its set of vertices $V$ is not always made strictly, and commonly a vertex $u$ is said to be in $G$ when it should be said to be in $V$. The order (or size) of a graph $G$ is defined as the number of vertices of $G$ and it is represented as $|V|$ and the number of edges as $|E|$. If two vertices in $G$, say $u,v \in V$, are connected by an edge $e \in E$, this is denoted by $e = (u,v)$ and the two vertices are said to be adjacent or neighbors. Edges are said to be undirected when they have no direction, and a graph $G$ containing only such types of graphs is called undirected. When all edges have directions and therefore $(u,v)$ and $(v,u)$ can be distinguished, the graph is said to be directed. Usually, the term arc is used when the graph is directed, and the term edge is used when it is undirected. In this dissertation we will mainly use directed graphs, but graph matching can also be applied to undirected ones. In addition, a directed graph $G = (V,E)$ is called complete when there is always an edge $(u,v) \in E = V \times V$ between any two vertices $u,v$ in the graph.

Graph vertices and edges can also contain information. When this information is a simple label (i.e. a name or number) the graph is called labelled graph. Other times, vertices and edges contain some more information. These are called vertex and edge attributes, and the graph is called attributed graph. More usually, this concept is further specified by distinguishing between vertex-attributed (or weighted graphs) and edge-attributed graphs.

2) Graph Database:
In computing, a graph database is a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data.

Most graph databases are NoSQL in nature and store their data in a key-value store or document-oriented database. In general terms, they can be considered to be key-value databases with the additional relationship concept added. Relationships allow the values in the store to be related to each other in a free form way, as opposed to traditional relational database where the relationships are defined within the data itself. These relationships allow complex hierarchies to be quickly traversed, addressing one of the more common performance problems found in traditional key-value stores.

Structure Graph databases are based on graph theory. Graph databases employ nodes, properties, and edges. Nodes represent entities such as people, businesses, accounts, or any other item you might want to keep track of.

Properties are pertinent information that relate to nodes. For instance, if Wikipedia were one of the nodes, one might have it tied to properties such as website, reference material, or word that starts with the letter w, depending on which aspects of Wikipedia are pertinent to the particular database.

C. Query Processing:
Query processing is a 3-step process that transforms a high-level query (of relational calculus/SQL) into an equivalent and more efficient lower-level query (of relational algebra).

Distributed query processing: Transform a high-level query (of relational calculus/SQL) on a distributed database (i.e., a set of global relations) into an equivalent and efficient lower-level query (of relational algebra) on relation fragments.

In graph theory, Query processing differs between two principle kinds of graph databases. In the first kind, which consists of a collection of small to medium size graphs, query processing involves finding all graphs in the collection that are similar to or contain similar subgraphs to a query graph (SUBGRAPH MATCHING). In the second case, the database consists of a single large graph, and the goal of query processing is to find all of its subgraphs that are similar to the given query graph.

1) Graph Matching Problems:
Many fields such as computer vision, scene analysis, chemistry and molecular biology have applications in which images have to be processed and some regions have to be searched for and identified. When this processing is to be performed by a computer automatically without the assistance of a human expert, a useful way of representing the knowledge is by using graphs.

Graphs have been proved as an effective way of representing objects. When using graphs to represent objects or images, vertices usually represent regions (or features) of the object or images, and edges between them represent the relations between regions. Graph Matching that is finding whether two graphs are equivalent has been an interesting problem from the day graph theory as a discipline has emerged. The many different solutions proposed have been able to check isomorphism (graph matching) on a particular class of graphs only. Two graphs G1 and G2 are isomorphic if and only if there is a permutation of the labeling of the vertices such that the two graphs are equivalent.

In this work, we will consider query-based pattern recognition problems, where the model is represented as a graph (the query graph, $Q$), and another graph (the data graph, $D$) represents the image where recognition has to be performed. The latter graph is built from a segmentation of the image into regions. Graphs can be used for representing objects or general knowledge, and they can be either directed or undirected. When edges are undirected, they simply indicate the existence of a relation between two vertices. On the other hand, directed edges are used when relations between vertices are considered in a not symmetric way. Different variants of graph matching: -
In this work, we will study about Attributed Sub Graph Matching: Sub-graph matching on weighted/cost graphs is referred to as attributed sub- graph matching. Gallagher (2006), describes attributed sub graph matching as an important variant of inexact graph matching having applications in computer vision, electronics, computer aided design etc. Kriege and Mutzel (2012), propose graph kernels for sub graph matching.

2) Subgraph Matching with Set Similarity:
In this work, we focus on a variant of the subgraph matching query, called subgraph matching with set similarity (SMS2) query, in which each vertex is associated with a set of elements with dynamic weights instead of a single label. The weights of elements are specified by users in different queries according to different application requirements or evolving data. Specifically, given a query graph Q with n vertices ui (i=1,...,n), the SMS2 query retrieves all the subgraphs X with n vertices vj (j = 1,...,n) in a large graph G, such that (1) the weighted set similarity between S(ui) and S(vj) is larger than a user specified similarity threshold, where S(ui) and S(vj) are sets associated with ui and vj, respectively; (2) X is structurally isomorphic to Q with ui mapping to vj.

Fig. 3: An Example of finding groups of cited papers in DBLP that match with the query citation graph
It is challenging to utilize both dynamic weighted set similarity and structural constraints to efficiently answer SMS2 queries. Due to the inefficiency of existing methods, an efficient SMS2 query processing approach is proposed by Liang Hong. This approach adopts a —filter-and-refine framework.

3) Applications of Subgraph Matching:
In automata theory: multiple uses, mainly to show that some two languages are equal.
In parallel processing, to reason about behaviour of complex systems.
In verification of many things: computer programs, logic proofs, or even electronic circuits.
In any search engine that can use formulas more sophisticated than words, e.g. in chemistry, biology, but I guess also music and other arts fall into that category as well.
Security, i.e. fingerprints scanners, facial scanners, retina scanners and so on. Any system that performs clustering would benefit from fast graph-isomorphism algorithm, e.g. o linking two facebook's accounts of the same person, o recognizing web users based on their behavior, o recognizing plagiarism in students solutions.
Analysis of social structures (special cases may include schools, military, parties, events, etc.), big part of it is search again.
Analysis of business relations

D. Big Data Processing with Apache Spark:
This work of Sub graph matching with set similarity is carried out in a distributed environment which requires the use of Apache Spark.

1) Apache Spark Overview:
Apache Spark is a cluster computing platform designed to be fast and general-purpose. On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools.

Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries. It also integrates closely with other Big Data tools. In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

2) Programming Model:
To use Spark, developers write a driver program that implements the high-level control flow of their application and launches various operations in parallel. Spark provides two main abstractions for parallel programming: resilient distributed datasets and parallel operations on these datasets (invoked by passing a function to apply on a dataset).
a) Resilient Distributed Datasets (RDDs)
A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail. In Spark, each RDD is represented by a Scala object. Spark lets programmers construct RDDs in four ways:
- From a file in a shared file system, such as the Hadoop Distributed File System (HDFS).
- By parallelizing a Scala collection (e.g., an array) in the driver program, which means dividing it into a number of slices that will be sent to multiple nodes.
- By transforming an existing RDD.
- By changing the persistence of an existing RDD.

b) Parallel Operations:
Several parallel operations can be performed on RDDs:
- reduce: Combines dataset elements using an associative function to produce a result at the driver program.
- collect: Sends all elements of the dataset to the driver program.
- foreach: Passes each element through a user provided function.

c) Shared Variables:
Programmers invoke operations like map, filter and reduce by passing closures (functions) to Spark. As is typical in functional programming, these closures can refer to variables in the scope where they are created.

3) GraphX:
In this work, we also make use of one of the components of Apache Spark—GraphX.

GraphX is a library for manipulating graphs (e.g., a social network’s friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API. The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication. By incorporating recent advances in graph-parallel systems, GraphX is able to optimize the execution of graph operations.

4) Apache Spark Vs Hadoop Mapreduce:
The difference in Spark from Hadoop is that it performs in-memory processing of data. This in-memory processing is a faster process as there is no time spent in moving the data/processes in and out of the disk, whereas MapReduce requires a lot of time to perform these input/output operations thereby increasing latency. Spark uses more RAM instead of network and disk I/O its relatively fast as compared to hadoop. But as it uses large RAM it needs a dedicated high end physical machine for producing effective results. Spark claims to process data 100x faster than MapReduce, while 10x faster with the disks. Hadoop is used in batch processing and Spark is used in real-time processing.

E. Organisation of The Report:
Chapter 2 documents various subgraph matching (isomorphism) algorithms developed over the time, and some selected isomorphism algorithms. Chapter 3 states the problem statement of project; and Chapter 4 explains proposed system design, including all design modules and their constraints. Chapter 5 include implementation details of all modules. Chapter 6 presents the results, analysis and comparison of the results with standard algorithms.

II. PRUNING

A. Anti-Monotone Pruning:
Given a query vertex u, for each accessed frequent pattern P in the inverted pattern lattice, if UB(S(u), P) < 1, all vertices in the inverted list L(P) and L(P’ can be safely pruned, where P’ is a descendant node of P in the lattice. 4.2.1.2.2 Vertical Pruning: Vertical pruning is based on the prefix filtering principle [18]: if two canonicalized sets are similar, the prefixes of these two sets should overlap with each other.

Finding maximum prefix length: The first p elements in the canonicalized set S(u) is denoted as the p-prefix of S(u). We find the maximum prefix length p such that if S(u) and S(v) have no overlap in p-prefix, S(v) can be safely pruned, because they do not have enough overlap to meet the similarity threshold [18].

To find p-prefix of S(u): Each time we remove the element with the largest weight from S(u), we check whether the remaining set S’(u) meets the similarity threshold with S(u). We denote L1 norm of S(u) as I{S(u)}1 = Σ. If I{S(u)}1 I{S(u)}1, the removal stops. The value of p is equal to |S’(u)| - 1, where |S’(u)| is the number of elements in S’(u). For any set S(v) that does not contain the elements in S’(u)’s p-prefix, we have sim(S(u), S(v)) < 1, so S(u) and S(v) will not meet the set similarity threshold.

Theorem: Given a query set S(u) and a frequent pattern P in the lattice, if P is not a one-frequent pattern (or its descendant) in S’(u)’s p-prefix, all vertices in the inverted list L(P) can be safely pruned.

B. Horizontal Pruning:
In the inverted pattern lattice, each frequent pattern P is a subset of data vertices (i.e., element sets) in P’s inverted list. Suppose we can find the length upper bound for S(u) (denoted by LU(u)). If the size of P is larger than LU(u), (i.e., the sizes of all data vertices in P’s inverted list are larger than LU(u)) then P and its inverted list can be pruned. Due to dynamic element weights, we need to find S(u)’s length interval on the fly. We find LU(u) by adding elements in (U-S(u)) to S(u) in an increasing order of their weights. Each time an element is added, a new set S’(u) is formed. We calculate the similarity value between S(u) and S’(u) if sim(S(u), S’(u)) holds, we continue to add elements to S’(u). Otherwise, the upper bound LU(u) equals to |S’(u)| - 1.

All frequent patterns under Level LU(u) will be pruned.

C. Putting All Pruning Techniques Together:
We Apply All the Set Similarity Pruning Techniques and Obtain Candidates for a Query Vertex.
For each query vertex \( u \), we first use vertical pruning and horizontal pruning to filter out false positive patterns and jointly determine the nodes (i.e., frequent patterns).

Then, we traverse them in a breadth-first manner. For each accessed node \( P \) in the lattice, we check whether \( UB(S(u), P) \) is less than. If yes, \( P \) and its descendant nodes are pruned safely.

**D. Structure Based Pruning:**

A matching subgraph should not only have its vertices (element sets) similar to corresponding query vertices, but also preserve the same structure as \( Q \). We design lightweight signatures for both query vertices and data vertices to further filter the candidates after set similarity pruning by structural constraints.

1) **Structural Signatures:**

Two structural signatures are defined -

- Query Signature \( \text{Sig}(U) \),
- Data Signature \( \text{Sig}(V) \),

For Each Query Vertex \( U \) And Data Vertex \( V \), Respectively.

To encode structural information - \( \text{Sig}(u) = \text{Sig}(v) \) should contain the element information of both \( u=v \) and its surrounding vertices.

Procedure:

- First sort elements in element sets \( S(u) \) and \( S(v) \) according to a predefined order (e.g., alphabetic order).
- Based on the sorted sets, we encode the element set \( S(u) \) by a bit vector, denoted by \( \text{BV}(u) \), for the former part of \( \text{Sig}(u) \).
- Each position \( \text{BV}(u)[i] \) in the vector corresponds to one element \( ai \), where \( 1 \leq i \leq |U| \) and \( |U| \) is the total number of elements in the universe \( U \). If an element \( aj \) belongs to set \( S(u) \), then in bit vector \( \text{BV}(u) \), we have \( \text{BV}(u)[j]=1 \), otherwise \( \text{BV}(u)[j]=0 \) holds.
- Similarly, \( \text{Sig}(v) \) is also encoded using the above technique.

- For the latter part of \( \text{Sig}(u) \) and \( \text{Sig}(v) \) (i.e., encoding surrounding vertices), we propose two different encoding techniques for \( \text{Sig}(u) \) and \( \text{Sig}(v) \), respectively.

**(Query Signature):** Given a vertex \( u \) with \( n \) adjacent neighbor vertices \( u_i \) \((i=1,...,m)\) in a query graph \( Q \), the query signature \( \text{Sig}(u) \) of vertex \( u \) is given by a set of bit vectors, that is, \( \text{Sig}(u)=[\text{BV}(u),\text{BV}(u_1),...,\text{BV}(u_m)] \), where \( \text{BV}(u) \) and \( \text{BV}(v) \) are the bit vectors that encode elements in set \( S(u) \) and \( S(u_i) \), respectively.

**(Data Signature):** Given a vertex \( v \) with \( n \) adjacent neighbor vertices \( v_i \) \((i=1,...,n)\) in a data graph \( G \), the data signature, \( \text{Sig}(v) \), of vertex \( v \) is given by: \( \text{Sig}(v)=[[\text{BV}(v),V]\), where \( v \) is a bitwise OR operator, \( \text{BV}(v) \) is the bit vector associated with \( v \), \( V \) is called a union bit vector, which equals to bitwise-OR over all bit vectors of \( v \)’s one-hop neighbors.

2) **Signature Based Dht(Distributed Hash Table):**

To enable efficient pruning based on structural information, we use Distributed Hash Table to hash each data signature \( \text{Sig}(v) \) into a signature bucket.

3) **Structural Pruning:**

a) Finding Similarity Using Jaccard Similarity:

Given: Bit vectors \( \text{BV}(u) \) and \( \text{BV}(v) \)

To Find : similarity between \( \text{BV}(u) \) and \( \text{BV}(v) \) 

\[
\text{sim}(\text{BV}(u),\text{BV}(v)) = \sum_{u \in U} \sum_{v \in V} \text{UB}'(\text{BV}(u),\text{BV}(v)) < \theta 
\]

where \( \wedge \) is a bitwise AND operator and \( \lor \) is a bitwise OR operator, \( a \in (\text{BV}(u) \wedge \text{BV}(v)) \) means the bit corresponding to element \( a \) is 1, \( W(a) \) is the assigned weight of \( a \).

For each \( \text{BV}(ui) \), we need to determine whether there exists a \( \text{BV}(vj) \) so that \( \text{sim}(\text{BV}(ui),\text{BV}(vj)) \) holds. To this end, we estimate the union similarity upper bound between \( \text{BV}(ui) \) and \( \text{BV}(v) \), which is defined as follows.

\[
\text{UB}'(\text{BV}(ui),\text{BV}(v)) = \sum_{u \in U} \sum_{v \in V} \text{UB}'(\text{BV}(ui),\text{BV}(v)) < \theta 
\]

Based on above definitions we have Aggregate Dominance Principle described as follows- Given a query signature \( \text{Sig}(u) \) and a data signature \( \text{Sig}(v) \), if \( \text{UB}'(\text{BV}(ui),\text{BV}(v)) < \theta \) then for each one-hop neighboring \( v_j \) of \( v \), \( \text{sim}(\text{BV}(ui),\text{BV}(vj)) < \theta \).

Based on aggregate dominance principle, we have the following Lemmas – Lemma 1. Given a query signature \( \text{Sig}(u) \) and a bucket signature \( \text{Sig}(B) \), assume bucket \( B \) contains \( n \) data signatures \( \text{Sig}(vt) \) \((t=1,2,...,n)\) , if \( \text{UB}'(\text{BV}(ui),\text{BV}(vt)) < \theta \) then there exists at least one neighboring vertex \( u_i \) \((i=1,...,m)\) of \( u \) such that \( \text{UB}'(\text{BV}(ui),\text{BV}(vt)) < \theta \), then all data signatures in bucket \( B \) can be pruned.

Lemma 2. Given a query signature \( \text{Sig}(u) \) and a data signature \( \text{Sig}(v) \), if \( \text{sim}(\text{BV}(u),\text{BV}(v)) < \theta \) then there is at least one neighboring vertex \( u_i \) \((i=1,...,m)\) of \( u \) such that \( \text{UB}'(\text{BV}(ui),\text{BV}(v)) < \theta \) can be pruned.

The aggregate dominance principle guarantees that structural pruning will not prune legitimate candidates.

E. **Dominating-Set-Based Subgraph Matching:**

An efficient dominating-set-based subgraph matching algorithm (denoted by DS-Match) facilitated by a dominating set selection method is proposed.

1) **DS-Match Algorithm (For The Distributed Systems):**

It first finds matches of a dominating query graph \( \text{DGQ}(QD) \) \((\text{QD} \text{ defined later}) \) formed by the vertices in dominating set \( \text{DS}(Q) \), then verifies whether each match of \( \text{QD} \) can be extended as a match of \( Q \).

DSMatch is motivated by two observations:

- First, compared with typical subgraph matching over vertex-labeled graph, the overhead of finding candidates in SMS2 queries is relatively higher, as the computation cost of set similarity is much higher than that of label matching. We can save filtering cost by only finding candidate vertices for dominating vertices rather than all vertices in \( Q \).
- Second, we can speed up subgraph matching by only finding matches of dominating query vertices. The candidates of remaining (non-dominating) query vertices can be filled up by the structural constraints between dominating vertices and non-dominating vertices. In this way, the size of intermediate results during subgraph matching can be greatly reduced.

**(Dominating Set):** Let \( Q=(V,E) \) be a undirected, simple graph without loops, where \( V \) is the set of vertices and \( E \) is the set of edges. A set \( \text{DS}(Q) \) \( V \) is called a dominating set for \( Q \) if every vertex of \( Q \) is either in \( \text{DS}(Q) \), or adjacent to some vertex in \( \text{DS}(Q) \).

Based on above Definition, we have the following Theorem:
III. THEOREM

Assume that u is a dominating vertex in Q’s dominating set DS(Q). If |DS(Q)| 2. Then, there exists at least one vertex u’ ∈ DS(Q) such that Hop(u,u’) 3, where Hop(…) is the minimal number of hops between two vertices. The dominating vertex u’ is called a neighboring dominating vertex of u.

IV. DOMINATING QUERY GRAPH:

Given a dominating set DS(Q), the dominating query graph QD is defined as by (V(QD),E(QD)) , and there is an edge (ui,uj) in E(QD) iff at least one of the following conditions holds: 1) ui is adjacent to uj in query graph Q (Fig. a) 2)|N1(ui) N1(uj)| > 0 (Fig. b) 3) |N1(ui) N2(uj)| > 0 (Fig. c) Figure 4.2
4) |N2(ui) N1(uj)| > 0 (Fig. d)

To transform a query graph Q to a dominating query graph QD, we first find a dominating set DS(Q) of Q. Then for each pair of vertices ui,uj in DS(Q), we determine whether there is an edge (ui,uj) between them and the weight of (ui,uj) according to above rules.

To find matches of dominating query graph, we propose the distance preservation principle.

V. DISTANCE PRESERVATION PRINCIPLE

Given: a subgraph match XD of QD in data graph G, XD and XD have n vertices (u1,...,un) and (v1,...,vn) respectively, where vi ∈ (ui)... Considering an edge (ui,uj) in XD, we have the following distance preservation conditions hold: 1) if the edge weight is 1, then vi is adjacent to vj. 2) if the edge weight is 2, |N1(ui) N1(uj)| > 0 or |N2(ui) N1(uj)| > 0 or |N2(uj) N1(uj)| > 0.

A. Dominating Set Selection:

A query graph may have multiple dominating sets, leading to different performance of SMS2 query processing. Motivated by such observation, in this subsection, we propose a dominating set selection algorithm to select a cost-efficient dominating set of query graph Q, so that the cost of answering SMS2 query can be reduced.

To problem of finding a cost-efficient dominating set is actually a Minimum Dominating Set (MDS) problem. MDS problem is equivalent to Minimum Edge Cover problem [2], which is NP-hard. As a result, we use a best effort Branch and Reduce algorithm [2]. The algorithm recursively select a vertex u with minimum number of candidate vertices from query vertices that have not been selected, and add to the edge cover an arbitrary edge incident to u. The overhead of finding the dominating set is low, because the query graph is usually small.

REFERENCES

[19] Liang Hong, Member, IEEE, Lei Zou, Member, IEEE, Xiang Lian, Member, IEEE, and Philip S. Yu, Fellow, IEEE, “ Subgraph Matching with Set Similarity in a Large Graph Database”, IEEE transactions on knowledge and data engineering, vol. 27, no. 9, September 2015