International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:11, 2009

An Efficient Graph Query Algorithm Based on Important Vertices and Decision Features

Xiantong Li, Jianzhong Li

***Abstract*—**Graph has become increasingly important in modeling complicated structures and schemaless data such as proteins, chemical compounds, and XML documents. Given a graph query, it is desirable to retrieve graphs quickly from a large database via graph-based indices. Different from the existing methods, our approach, called VFM (Vertex to Frequent Feature Mapping), makes use of vertices and decision features as the basic indexing feature. VFM constructs two mappings between vertices and frequent features to answer graph queries. The VFM approach not only provides an elegant solution to the graph indexing problem, but also demonstrates how database indexing and query processing can benefit from data mining, especially frequent pattern mining. The results show that the proposed method not only avoids the enumeration method of getting subgraphs of query graph, but also effectively reduces the subgraph isomorphism tests between the query graph and graphs in candidate answer set in verification stage.

***Keywords*—**Decision Feature, Frequent Feature, Graph Dataset, Graph Query.

I. Introduction

GRAPHS, as a general data structure, provide a powerful and primitive tool to model the data in a variety of applications, e.g. social or information networks, biological networks, 2D/3D objects in pattern recognition, wired or wireless interconnections, chemical compounds or protein networks. Nodes in graphs usually represent real world objects and edges indicate relationships between the objects. For example, in computer vision, graphs are used to represent complex relationships, such as the organization of entities in images. In chemical informatics and bio-informatics, graphs are employed to denote compounds and proteins. In order to accurately describe the characters of the data, the labeled graphs are often applied. The nodes and edges are associated with attributes in a labeled graph. With the tremendous amount of structured or networked data accumulated in large databases, how to efficiently support the scalable graph query processing becomes a challenging research issue in database area.

Given a graph dataset *GD* and a query graph *q*, the target of graph query is to get all supergraphs of *q* in *GD*, or $Q=\{g|g\in GD\wedge q\subseteq g\}$. For example, in chemistry, the subgraph can be used to retrieve the compounds containing the given special substructure. Graph query can also be used in biology identification.

Recently, graph query has been received much more concern and been well studied in [1]-[13]. The indices of [1]-[4] are founded on frequent features, such as frequent paths, subtrees, or subgraphs, which are generated from graph mining methods. Reference [5] and [6] are two similarity graph query algorithm based on [1]. In [7]-[11], the strategies use other feature set to construct the indices, such as graph closure, directed acyclic graph decomposition, special components in graph, and so on. In some special applications, some other more efficient algorithms are provided, such as [12], [13] in XML.

Including precise and similar methods, the methodology of the pressed algorithms is founded on *Filter-Verification*. The 'filter' stage erases fake answers as much as possible in order to form a smallest candidate answer set. In 'verification' stage, each graph in candidate answer set is verified by taking subgraph isomorphism test with query graph *q*. The subgraph isomorphism is NP-complete problem [14] whereas the candidate answer set should be as small as one can. The *inclusion logic* is used while filtering takes place: Give two labeled graphs g_1 and g_2. And g' is a subgraph of g_1 ($g'\subseteq g_1$). If g_1 is a subgraph of g_2, then g' is a subgraph of g_2 too, or $((g_1\subseteq g_2)\Rightarrow(g'\subseteq g_2))$. On the other side, if g' is *not* a subgraph of g_2, it can be drawn that g_1 is not a subgraph of g_2 too, or $((g'\nsubseteq g_2)\Rightarrow(g_1\nsubseteq g_2))$.

The efficiency of graph query is restricted by two operations. One is the subgraphs enumeration of query graph q. After query graph *q* is given, the algorithm enumerates all subgraphs of *q*, in order to find out the indexed feature set F_q which are subgraphs of *q*, or $F_q=\{f|f\subseteq q\wedge f\in F\}$, where F is the indexed feature set. The other part is subgraph isomorphism between candidate graphs and query graph in verification stage. Subgraph isomorphism is NP-complete problem. The most important issue of graph query research is to reduce the number of subgraph isomorphism tests.

In this paper, a vertex-feature mapping based index structure VFM (Vertex to Frequent Feature Mapping) is proposed to handle graph query problem. There are two mappings in VFM. One maps important vertex array (VA) to its neighbor composed array (CA) and the other maps (VA, CA) to a decision feature set (DF). Through the two mappings, VFM

Xiantong Li is with the School of Computer Science and Technology at Harbin Institute of Technology, Harbin, Heilongjiang, China 150001(Corresponding author. phone: 13936394192; e-mail: lxt@hit.edu.cn).
Jianzhong Li is with Harbin Institute of Technology, Harbin, Heilongjiang, China. 150001 (e-mail: lijzh@hit.edu.cn).

builds the frequent feature set F_q which q contains in polynomial time. The main contributions of this paper are:

1. VFM builds the indexed frequent feature set in which are contained by q. This set is formed by VFM-Index through the mappings. In this way, the enumeration method is avoided to decompose q and the complexity is reduced indeed.

2. Through the two mappings in VFM, a more precise location of frequent indexed features is realized and produces a much smaller candidate answer set.

The rest of this paper is organized as follows. In section 2, the preliminaries of graph query are given. Index construction algorithm and query algorithm are given in section 3 and the efficiency of VFM is analyzed in section 4. In section 5, the experimental results are given. The conclusion of this paper is addressed in section 6.

II. Preliminaries

Definition 2-1 A **labeled graph** G is a 5-tuple $G = \{V,E,\Sigma V,\Sigma E,L\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of undirected edges. ΣV and ΣE are the sets of vertex labels and edge labels respectively. The labeling function L defines the mappings $V \rightarrow \Sigma V$ and $E \rightarrow \Sigma E$.

Definition 2-2 Given a pair of labeled graphs $G = \{V,E,\Sigma V,\Sigma E,L\}$ and $G' = \{V',E',\Sigma V',\Sigma E',L'\}$, G is a **subgraph** of G' iff

- $V \subseteq V'$,
- $\forall u \in V, (L(u)= L'(u))$,
- $E \subseteq E'$,
- $\forall (u, v) \in E, (L(u, v)= L'(u, v))$

Here, G' is also referred to as a supergraph of G.

Definition 2-3 Graph isomorphism is a bijection $f: V(G)\leftrightarrow V(G')$. For given two labeled graphs $G=\{V,E,\Sigma V,\Sigma E,L\}$ and $G'=\{V',E',\Sigma V',\Sigma E', L'\}$, if they are isomorphic, they satisfied such conditions:

1). $\forall u\in V, L(u)=L'(f(u))$ and

2). $\forall u,v\in V,((u,v)\in E)\Leftrightarrow((f(u),f(v))\in E')$, and

3). $\forall (u,v)\in E, L(u,v)=L'(f(u),f(v))$.

If G and G' are graph isomorphic, it noted as $G\equiv G'$.

The bijection f is an isomorphism between G and G'. We also say that G is isomorphic to G' and vice versa.

Definition 2-4 A labeled graph G is **subgraph isomorphic** to a labeled graph G', denoted by $G\subseteq G'$, iff there exists a subgraph G'' of G' such that G is isomorphic to G''.

Here, G'' is an embedding of G in G'.

It is proven in [14] that subgraph isomorphism is NP-complete. In the process of graph query, subgraph isomorphism is unavoidable. How to reduce the number of subgraph isomorphism test is the key character about efficiency in graph query.

Definition 2-5 Given a graph dataset GD and query graph q, the answer set of **graph query** is Q. The graphs in Q satisfy the following condition:

$$Q = \{g \mid g\in GD \wedge q\subseteq g\}.$$

In this paper, unless specific statement, the size of a graph is the edges it has. Given a labeled graph G, the size of G is noted as $|G|$. The algorithm proposed in this paper is used in undirected labeled graph set, though it can be fit in directed graph set with slight adjusted. For avoiding the confusions, the graphs in graph dataset are named *target graph*.

III. VFM-Index

The naïve method of graph query is compared query graph q and every target graph in graph dataset. It is very inefficient. For reducing subgraph isomorphism computation in graph query, the *inclusion logic* is introduced to filter out a smaller candidate answer set from graph dataset. The inclusion logic is: Given two graphs q and g. If q is a subgraph of g, then all subgraphs of q are subgraphs of g, too. Otherwise, even if one subgraph of q is not a subgraph of g, q is not a subgraph of g.

According to inclusion logic, the *filter-verification* method of graph query is: 1) forming a subset of indexed features which are subgraphs of query graph q, 2) getting the support set of those features, 3) intersecting the support sets to build the candidate set C_q, and 4) verifying the target graphs in C_q one bye one.

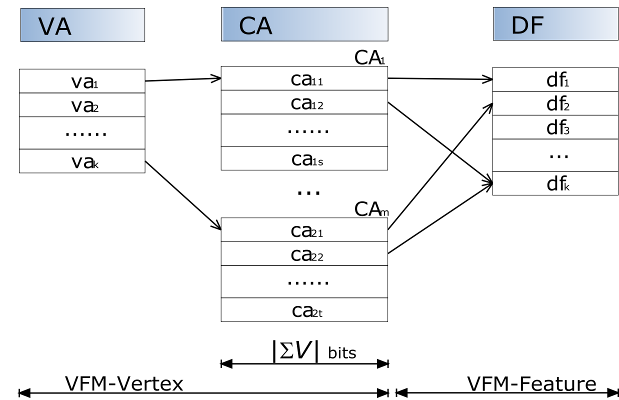


Fig. 1 Data structure of VFM

The data structure of VFM-Index is shown in fig.1. It can be divided into two parts, VFM-Vertex and VFM-Feature. VFM-Vertex is composed by two types of arrays, VA (Vertex Array) and CA (Composed Array). *VA* brings the information of vertex label, vertex degree (neighbors), and neighbor relationship. Each *VA* points to a set of *CA* array. *CA* array has $|\Sigma V|$ bits, which represents the neighbors' labels. Here, '1' is positive and '0' is negative. A serial of *DF* array is VFM-Feature, which is pointed by *CA*. Each *DF* entry is a set of decision features which have the important node *VA* and *CA* described.

VFM-Index is organized as follow.

1. Off-line Index Construction. In this stage, *VA*, *CA* and *DF* are built according to *GD*. At the same time, every entry of *DF* should record its own support set.

2. Searching Stage. When query graph q is given, VFM-Index gets important vertices of q. According to the

values of each vertex of q, it locates DF entries through the mappings between VA, CA.

3. Filtering Stage. VFM-Index construct candidate answer set C_q by intersecting the decision features support set which are contained in DF array in stage 2. Or, $C_q = \bigcap_{df} D_{df} (df \in DF \wedge df \subseteq q \wedge (df \subseteq g \wedge g \in GD))$.

4. Verification Stage. To return the correct answers, VFM-Index verifies every target graph in C_q.

The rest of this section will discuss the construction of VFM-Index and query process.

A. VFM-Vertex

Vertices are the basic part of a labeled graph. If the graph index is built on vertices, the naïve method is forming a relationship between vertices and target graphs. But, vertex brings no structure information. Its filtering ability is poor and not suitable for *Filter-Verification* method. However, the size of such index structure can be totally controlled. Vertices index size has a linear relationship with $|\Sigma V|$.

VFM-Vertex describes the important vertices in a graph by a 3-tuple: *VA (L, D, NC)*, *L* is the label of the vertex, *D* is the degree (or neighbors) of the vertex, *NC* is the status of the neighbor connections, which records the number of connections between all its neighbors.

There are various ways of measuring the importance of a vertex in a graph. For simplicity, we use the degree centrality measure in this paper. In this measure, vertices with high degrees are considered more important than vertices with low degrees. Note that the definition of "importance" is flexible in VFM-Vertex and customizable for specific application needs. VFM-Vertex can be easily extended to use other measures of node importance, such as *closeness*, *betweenness*, and *eigenvector centralities*.

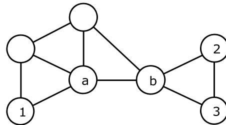


Fig. 2 Labeled graph G

For example, fig.2 shows a labeled graph *G*. For simplicity of describing, we erase all edge labels and part vertex labels. 'a' and 'b' are labels of vertices and the numbers are serial number of vertices. Vertex a and vertex b can be described by *VA*, VA_a: (*a*, 4, 3), VA_b: (*b*, 4, 2).

There is a linear relationship between *VAs*.

Definition 3-1 Given two vertices *a*, *b* in a labeled graph. The relationship of *VA* between them is:

1) $VA_a = VA_b$, iff $(L_a = L_b) \wedge (D_a = D_b) \wedge (NC_a = NC_b)$

2) $VA_a \prec VA_b$, iff

   a) $(L_a = L_b) \wedge (D_a < D_b) \wedge (NC_a < NC_b)$ or

   b) $(L_a = L_b) \wedge (D_a = D_b) \wedge (NC_a < NC_b)$ or

   c) $(L_a = L_b) \wedge (D_a < D_b) \wedge (NC_a = NC_b)$

3) $VA_a \succ VA_b$, others.

For a given query graph *q* in the process of graph query, the way to pick frequent features *q* contained is a complicated part. First, it enumerates all subgraphs of *q*. Then, subgraph isomorphism tests take place between the subgraphs of *q* and frequent features to find out which indexed features are contained by *q*. In this paper, the frequent feature set of *q* is forming by *VA*. The features in $F_q = \{f \mid f \in F \wedge (T_q = T_f \vee T_q \prec T_f)\}$ is much less than indexed frequent features. VFM-Vertex avoids the process of enumerating subgraphs of *q* and main part of subgraph isomorphism between subgraphs of *q* and indexed frequent features.

A number of vertices have the same value of *VA*. For example, in fig.2, suppose the label of v_1, v_2 and v_3 is *c*. The value of *VA* of these vertices is (*c,* 2, 1). While, if the information of neighbor labels are appended to *VA*, the vertices can be classified more detailed. Such as above example, the neighbor labels separate v_1 from v_2 and v_3.

VFM-Vertex appends *CA* (Composed Array) into *VA* to classify vertices much detailed. For a given graph *G* and important vertex *v* in *G*, the node array of *v* is va_n which points to a list of Composed Array, or *CA*. Every entry of *CA* is a $|\Sigma V|$ bits array, which every bit represents a label in *GD*. If a label belongs to one neighbor of *v*, the value of that bit is '1'. But, if the value of $|\Sigma V|$ is too big to fit in main memory, the space which *CA* used should be very large. If this condition occurs, a method of [15] can be introduced to handle it. In this way, the labels of a vertex can be controlled in S_{bit} bits, and S_{bit} is a value which the user defined.

VFM-Vertex is composed by two parts. One part is the basic information of a vertex, or the triple of *VA*. The other part is a list of *CA*, which are connected with *VA* by structure information mapping.

In the coming subsection, the frequent feature part, VFM-Feature, is introduced.

B. VFM-Feature

Definition 3-2 The **support** of a graph *g* in graph dataset *GD* is the percentage of *g's* supergraphs in *GD*, or:

$$support(g, GD) = \frac{|\{g_i \mid g_i \in GD \wedge g \subseteq g_i\}|}{|GD|}$$

The definition of frequent feature is given about the concept of support.

Definition 3-3 Given a graph dataset $GD=\{g_1,g_2,\ldots,g_n\}$. A graph feature *g* is frequent about *GD*, if and only if its *support* is no less than a given support threshold *min_sup*, or: $support(g, GD) \geq min_sup$.

The *support set* of a frequent feature *g* is the set of all its supergraphs in *GD*. It is noted as $D_g=\{g_i|g_i \in GD \wedge g \subseteq g_i\}$. The definition of support can be transformed to its support set form: $support(g,GD)=|D_g|/|GD|$.

Frequent features expose the intrinsic characteristic of a graph database. Suppose all the frequent features with minimum support *min_sup* 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 *min_sup*. Since any graph with q embedded must contain q's subgraphs, D_f is a candidate answer set of query q. If *min_sup* is low, it is not expensive to verify the small number of graphs in D_f in order to find the query answer set. Therefore, it is feasible to index frequent fragments for graph query processing.

A further examination helps clarify the case where query q is not frequent in the graph database. We sort all q's subgraphs in the support decreasing order: $f_1, f_2, \ldots, f_n$. There must exist a boundary value i where $support(f_i,GD) \geq min_sup$, and $support(f_{i+1},GD) < min_sup$. Since all the frequent fragments with minimum support *min_sup* are indexed, the graphs containing $f_k (1 \leq k \leq i)$ are known. Therefore, we can compute the candidate answer set C_q by $\bigcap_{1 \leq k \leq i} D_{f_k}$, whose size is at most $support(f_i,GD)$. For many queries, $support(f_i,GD)$ is likely to be close to *min_sup*. Hence the intersection of its frequent fragments, $\bigcap_{1 \leq k \leq i} D_{f_k}$, leads to a small size of C_q. Therefore, the cost of verifying C_q is minimal when *min_sup* is low.

Frequent features can be gained from graph mining algorithms. During the process of mining, depth first search (DFS) is chosen for the high efficiency. At the beginning of mining, it always selects one frequent edge for expanding. It expands one edge in one round until it reaches the biggest subgraph. Then it feeds back to the upper layer to mine continuously. Fig.3 shows a frequent feature mining space. The fewer edges the graph has, the shorter distance it has from root to it. The distance from root to node is the number of edges the subgraph has which the node represents.

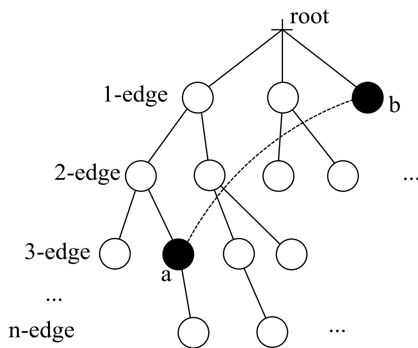


Fig. 3 Space of frequent feature mining

However, there are lots of redundancies between frequent features. Obviously, the index composed with redundant frequent features is too inefficient to answer graph query efficiently. For example, there are two paths $p_1 = v_1\text{-}v_2\text{-}v_3$ and $p_2 = v_1\text{-}v_2\text{-}v_3\text{-}v_4$ in frequent feature set. If $D_{p1} = D_{p2}$, then p_1 and p_2 are redundant to each other. When such situation takes place, keeping one of p_1, p_2 in the index is enough. In the general resolution, it always keeps the smaller feature for the reason that the probability of smaller features contained by q is much higher than bigger ones.

The elimination of features with same support is far from enough. In [1], the author proposes a discriminative ratio δ. When the ratio between the support of feature f and the intersection of all its sub-features' support is less than δ, f is redundant. The shortcoming of this factor is that the calculation has the process of enumerating subgraphs of f. It is very expensive. In this paper, a concept of *decision feature* is given to replace discriminative ratio to reduce the redundancy in frequent features.

The decision features (DF) occurs where the support jumps happen.

Definition 3-4 Given graph dataset *GD*, support threshold *min_sup* and decision factor σ. The frequent feature set is get from depth first mining method. Feature f' is a one edge extending supergraph of f, or f' is a son of f in DFS tree. If $support(f, GD)/support(f', GD) \geq \sigma$, there is a **support jump** between f and f'.

$support(f, GD)$ is always no less than $support(f', GD)$. For this reason, $\sigma \geq 1$, otherwise, it is meaningless.

The meaning of *support jump* is that there is a jump of support between f and its one edge supergraph. For example, in chemical compounds set, the frequency of benzene ring is very high. Suppose that benzene ring is composed by $e_1\ e_2 \ldots\ e_6$. In the process of extending from e_1 to $e_1\ e_2\ e_3\ e_4\ e_5\ e_6$, the value of support is almost equal. But, if extra edge introduced into benzene ring, the support jump occurs. The feature where support jump occurs is *decision feature* (DF). The number of DF is decided by decision factor σ.

In graph mining space, there is a path from root to leaf f_n: $rf_1f_2 \ldots f_n$, in which f_i is a frequent feature with i edges. The decision features on this path are $f_{1'}, f_{2'}, \ldots, f_{k'}$, and $|f_{i'}| \leq |f_{i'+1}|$. According to definition 3-4, $support(f_{i'},GD) > support(f_t,GD) \geq support(f_{i'+1},GD)$ where $i' \leq t < i'+1$. If the value of decision factor σ is given seemly, $support(f_t,GD) \approx support(f_{(i+1)'},GD) \cdot \sigma$. If the graph query index is built on decision features, the redundancy should be controlled well.

However, though the depth first search is selected to be the mining order to avoid duplication works, there are some re-mining works. For example, in fig.3 which is a typical depth first search tree (DFS tree), node a and b are two features in mining process. In this mining process, it might be happen that $g_b \subseteq g_a$. If the mining method discovers g_b which is a subgraph of g_a, and the support ratio between g_b and g_a satisfies the definition of decision feature, then g_b is a decision feature no matter whether g_a is a decision feature or not.

For quickly locating the decision features q has, VFM maps VFM-Vertex to a list of decision features (*DF* in fig.1) which satisfy the condition of *VA* and *CA*. Every *DF* in this list contains those important vertices which are constrained by the attributes of *VA* and *CA*. Through this mapping, VFM outputs the features contained by q in polynomial time, without enumeration process.

C. Algorithm of index construction

The VFM-Index is composed by VFM-Vertex and VFM-feature. VFM-Index has two mappings. One is the mapping between *VA* and *CA*. The other is the mapping between VFM-Vertex to a list of decision features (or *DF*). In actually situation, all one edge features are added into decision features to ensure the completeness of answers.

The construction algorithm of VFM-Index is given in *algorithm 1*.

VFM-Index can be divided into three parts.

The major target of first part (line 1-4) is to produce the frequent features by graph mining algorithm. In this stage, the mining space of *DFS* tree is constructing. All one edge features are put into the feature set (line 2-3). The index is constructed off-line, graph mining algorithm can be chosen freely.

Algorithm 1. VFM-Index

Input: Graph dataset GD, decision factor σ
Output: VFM-Index

1. $F \leftarrow \Phi$
2. for *all single edges in GD* do
3. $F \leftarrow e$
4. *mining frequent feature and forming DFS tree*
5. for *all nodes V_i in DFS tree* do
6. if $\dfrac{support(f_k, GD)}{support(f_{k-1}, GD)} \geq \sigma$
7. $F \leftarrow F \cup f_k$
8. if $f_k \subseteq V_i$
9. if $\dfrac{support(f_k, GD)}{support(\hat{f}_k, GD)} \geq \sigma$
10. $F \leftarrow F \cup f_k$
11. for *each important vertices set V in GD* do
12. $v_i \leftarrow va_v$
13. *organize v_i by descending order*
14. for *each v_i* do
15. $c_j \leftarrow ca$
16. *mapping(v_i, c_j)*
17. for *each c_j* do
18. $F_k \leftarrow \{f | f \in F \wedge (v_i c_j) \subseteq f\}$
19. *organize F_k by descending order $|F_k|$*
20. *mapping(c_j, F_k)*
21. return *V, C, F*

The second part (line 5-10) is executed paralleled with part 1. During the mining process, the supports of feature f_k and its son are compared to find out whether f_k is a decision feature about the given σ. If it does, f_k is added into feature set *F*.

The third part (line 11-20) is the mapping formation stage. It gets all important vertices of *GD* and calculates its *VA*. After the mapping between *VA* and *CA* is built, the mapping of (*VA*, *CA*) to *DF* is computed in line 17-20.

The set of *V*, *C* and *F* are returned in line 21.

D. Query algorithm

The graph query algorithm VFM-Query on VFM-Index is given in algorithm 2.

At the beginning of VFM-Query, the whole set of *GD* forms the candidate answer set C_q. After all features contained by *q* is pick out through the two mappings, the support sets of all these features are intersected with C_q to improve the precise of query.

When the algorithm begins, it visit *q* to pick out all its important features v_i. Then, decision features contained by *q* is calculated through the two mappings (line 5). These features are verified in line 7-9. At the end, the candidate answer set C_q is return by intersecting the support sets of those features.

Algorithm 2. VFM-Query

Input: Graph dataset *GD*, *VFM-Index*, query graph *q*
Output: candidate answer set C_q

1. $C_q \leftarrow GD$
2. $F_q \leftarrow \Phi$
3. $V \leftarrow$ *{all important vertices in q}*
4. for *each entry v_i in V* do
5. $F_i \leftarrow$ *mapping(mapping(v_i,c_i),n_i)*
6. $F_q \leftarrow F_q \cup F_i$
7. for *each DF in F_q* do
8. if $f_i \not\subseteq q$ do
9. $F_q \leftarrow F_q - f_i$
10. for *every feature f in F_q* do
11. $C_q \leftarrow C_q \cap D_f$
12. return C_q

IV. Effectiveness of VFM

While a query graph *q* is issued, the response time of the query processing algorithm VFM can be estimated by $T_{query} = T_{filter_nodes} + T_{filter_features} + T_{isoCand} * |C_q|$, where T_{query} is the query response time, T_{filter_nodes} is the time of filtering the features through important nodes, $T_{filter_features}$ is the time of the filtering of decision features, $T_{isoCand}$ is the time for subgraph isomorphism test between candidate graph and query graph *q*, and $|C_q|$ is the size of candidate graphs set. In VFM, the features contained by *q* is formalized by the two mappings, the complexity of this process is polynomial. For this reason, T_{filter_nodes} is much faster than other enumerating methods. The mappings between *VA* to *CA* and (*VA*, *CA*) to *DF* ensure a smaller frequent feature set. The smaller the features are, the fewer the subgraph isomorphism tests are taken. VFM-Vertex produces not only a cheap T_{filter_nodes}, but also a cheap $T_{filter_features}$.

On the other hand, the other part of high complicated is $T_{isoCand} * |C_q|$. Subgraph isomorphism is NP-complete, the effect factor is $|C_q|$. The experimental results given in section 5 show that VFM formulate a smaller candidate answer set than other algorithm does.

V. Experimental Study

In this section, we will report our experimental results that

validate the effectiveness and efficiency of the VFM algorithm. The performance of VFM is compared with that of TreePi, which outperforms gIndex in [2].

We use two types of datasets in our experiments: one real dataset and a series of synthetic datasets. The real dataset is an AIDS antiviral screen dataset containing more than 43,000 classified chemical molecules. This dataset is available publicly on the web site of the Developmental therapeutics Program. The synthetic data generator is the same as that in [16]. The generator allows the users to specify the number of graphs, their average size, the number of seed graphs, the average size of seed graphs, and the number of distinct labels.

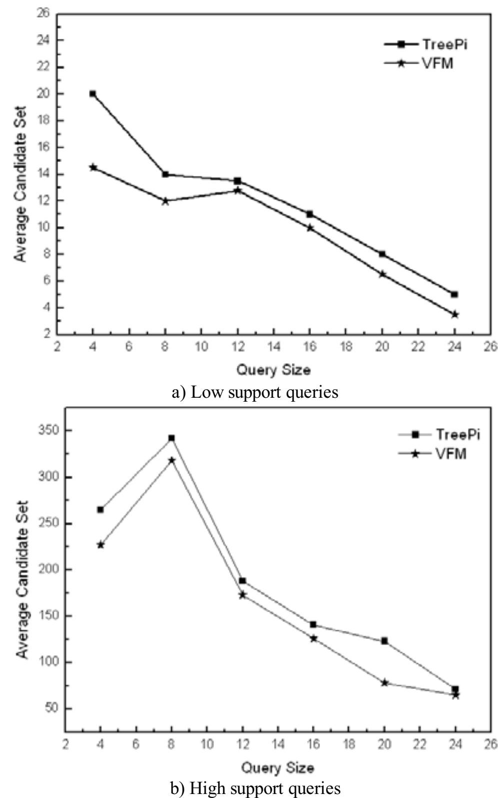
All our experiments were performed on a 3.2GHZ, 512GB memory, Intel PC running on Windows XP Professional with Service Package III. Both TreePi and VFM are compiled with gcc/g++.

A. AIDS Antiviral Screen Dataset

In this subsection, we report the experimental results on the antiviral screen dataset. The following parameters are set for TreePi. We set $\alpha = 5$, $\beta = 2$, $\eta = 10$ for the support threshold function, and $\gamma = 1.5$.As we now use a randomized algorithm to partition the query graph, we set $\delta = |q|$, which is relatively large. The same maximum size of features and equivalent number of features are chosen in TreePi and VFM so that a fair comparison between them can be performed.

In this experiment, $\Gamma_{10,000}$ is selected as the test dataset. During the tests, six query sets are tested, each of which has 1,000 queries. We randomly select 1,000 graphs from the antiviral screen dataset and then extract a connected m edge subgraph from each graph randomly. These 1,000 subgraphs are taken as query set, denoted by Q_m. m is selected from 4 to 24. The query sets are divided into two groups, low support group if its support is less than 50 and high support group otherwise. Since frequent patterns are used as index structures, it is crucial to show the index algorithm can perform well on both high support and low support queries.

Fig.4(a) and fig.4(b) present the pruning performance of TreePi and VFM on low support queries and high support queries, respectively. We also plot the average size of the actual support set of the query graphs, which is the optimal performance of a pruning algorithm. As shown in the figures, VFM surpasses TreePi in all query sets of different sizes.



a) Low support queries

b) High support queries

Fig. 4 Pruning Performance

Fig.5 presents the running time in constructing index patterns in both TreePi and VFM. The database size varies from 2,000 to 10,000 and the index is constructed from scratch for each database. The index construction time of the two methods is both approximately proportional to the database size, while VFM is relatively faster due to the following reasons: (1) the calculation of important vertices is much cheaper than subgraphs or subtrees. (2) The two mapping method is more efficient than center distance constraints.

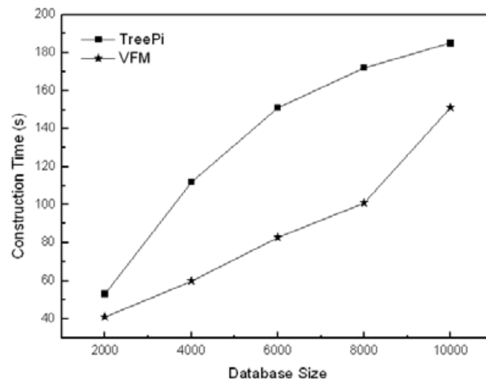


Fig. 5 Index Construction Time

In fig.6, the running time for query processing in both TreePi and VFM is presented. The X-axis represents the edge size of the query graph, and the Y-axis represents the running time to find the corresponding support sets of query graphs. For each edge size, we randomly generate 1000 graphs as query graphs and a graph database of size 6000 is used. Obviously VFM is

much faster than TreePi, since it reduces more than half of the running time for the query graphs of almost all the edge sizes. The reasons of the promotion of efficiency are: (1) the vertices calculation is cheaper than subtrees and center distance constrains. (2) The candidate answer set, which is tested in fig.4, of VFM is smaller than TreePi.

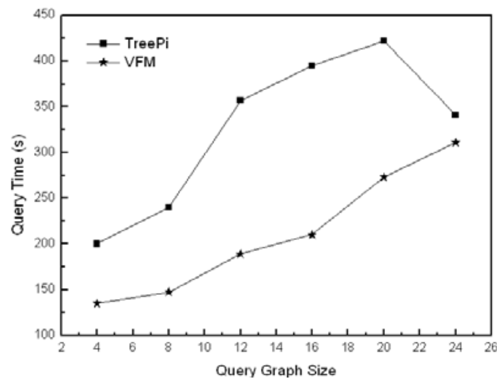


Fig. 6 Query Processing Time

B. Synthetic dataset

In this subsection, we present the performance comparison on synthetic datasets. The synthetic graph dataset is generated as follows: First, a set of seed fragments are generated randomly, whose size is determined by a Poisson distribution with Mean I. The size of each graph is a Poisson random variable with mean T. Seed fragments are then randomly selected and inserted into a graph one by one until the graph reaches its desired size. More details about the synthetic data generator are available in [9]. A typical dataset may have the following settings: it has 8,000 graphs and uses 1,000 seed fragments with 40 distinct labels. On average, each graph has 20 edges and each seed fragment has 10 edges. This dataset is denoted by D8kI10T20S1kL40.

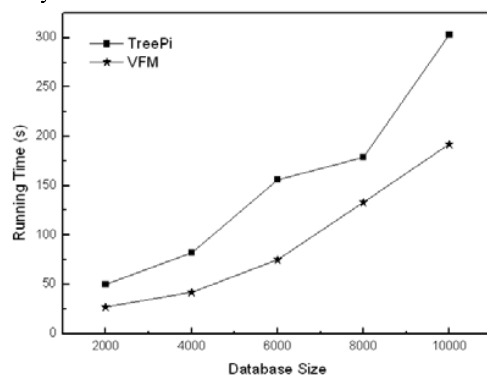


Fig. 7 Index Construction Time

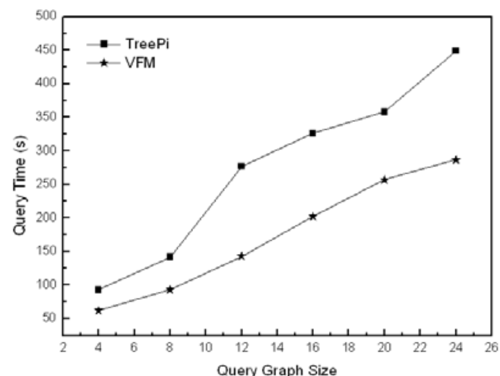


Fig. 8 Query Processing Time

Fig.7 and fig.8 show the comparison of index construction time and query processing time on synthetic dataset respectively. The comparisons show the similar results about on real data set.

VI. CONCLUSION

In this paper, a node to decision feature two-step mapping based graph query algorithm VFM is proposed. VFM is a *filter-verification* method. It is different from other graph query algorithms, VFM-Index is composed by *VA*, *CA* and *DF*. For avoiding the enumeration of subgraphs of *q*, VFM formulates all decision features that *q* has in polynomial time through mappings *VA* to *CA* and (*VA*, *CA*) to *DF*. The size of candidate answer set is reduced by the introduction of decision feature and promotes query efficiency a lot. Efficiency analysis and experimental results show that VFM has a better executing efficient and more suitable to be developed than TreePi.

REFERENCES

[1] X. Yan, P. S. Yu, et al. "Graph Indexing: A Frequent Structure based Approach", in Proc. SIGMOD'04, 2004, p. 335-346.
[2] S. Zhang, M. Hu, et al. "TreePi: A Novel Graph Indexing Method", in Proc. ICDE'07, 2007, p. 966-975.
[3] J. Cheng, Y. Ke, et al. "FG-Index: Towards Verification Free Query Processing on Graph Databases", in Proc. SIGMOD'07, 2007, p. 857-872.
[4] P. Zhao, J. X. Yu, et al. "Graph Indexing: Tree + Delta >= Graph", in Proc. VLDB'07, 2007, p. 938-949.
[5] X. Yan, P. S. Yu, et al. "Substructure Similarity Search in Graph Databases", in Proc, SIGMOD'05, 2005, p.766-777.
[6] X. Yan, F. Zhu, et al. "Searching Substructures with Superimposed Distance", in Proc. ICDE'06, 2006, p.88.
[7] D. Shasha, J. T. Wang, et al. "Algorithmics and Applications of Tree and Graph Searching", in Proc. PODS'02, 2002, p. 39-52.
[8] H. He, A. K. Singh. "Closure-Tree: An Index Structure for Graph Queries", in Proc. ICDE'06, 2006, p. 38.
[9] D. W. Williams, J. Huan, et al. "Graph Database Indexing Using Structured Graph Decomposition", in Proc. ICDE'07, 2007, p. 976-985.
[10] H. Jiang, H. Wang, et al. "GString: A Novel Approach for Efficient Search in Graph Databases", inProc. ICDE'07, 2007, p. 566-575.
[11] L. Zou, L. Chen, et al. "A novel spectral coding in a large graph database", in Proc. EDBT'08, 2008, p. 181-192.
[12] K. Gupta, D. Suciu. "Stream Processing of XPath Queries with Predicate"s, in Proc. SIGMOD, 2003, p. 419-430.
[13] P. Bohannon, W. Fan, et al. Narayan, "Information Preserving XML Schema Embedding", in Proc. VLDB, 2005, p. 85-96.
[14] M. Garey, D. Johnson, Javier Bezos. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1979.
[15] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970.
[16] M. Kuramochi, and G. Karypis. Frequent subgraph discovery. ICDE, 2001

Xiantong Li (1973.7-) is a Ph.D candidate of School of Computer Science and Technology at Harbin Institute of Technology, China. His current inteeresting is about Data Mining, Graph Mining, and Graph Query.

He has worked in the Harbin Institute of Technology Software Development Ltd. His articles which are published is: An Efficient Frequent Subgraph Mining Algorithm. Xiantong Li, Jianzhong Li, Hong Gao. Journal of Software, Vol.18, No.10, October 2007, pp.2469−2480. And, DAG Decomposition Based Algorithm of Graph Similarity Containment Query. Xiantong Li, Jianzhong Li. Journal of Harbin Institute of Technology. Vol.41, No.4, April, 2009.