A Novel Approach for Efficient Supergraph Query Processing on Graph Databases

Shuo Zhang, Jianzhong Li, Hong Gao, Zhaonian Zou
Department of Computer Science and Technology, Harbin Institute of Technology
92 West Dazhi Street, PO Box 750, Harbin, China
{zhangshuocn, lijzh, honggao, znzou}@hit.edu.cn

ABSTRACT

In recent years, large amount of data modeled by graphs, namely graph data, have been collected in various domains. Efficiently processing queries on graph databases has attracted a lot of research attentions. *Supergraph query* is a kind of new and important queries in practice. A *supergraph query*, q, on a graph database D is to retrieve all graphs in D such that q is a supergraph of them. Because the number of graphs in databases is large and subgraph isomorphism testing is NP-complete, efficiently processing such queries is a big challenge. This paper first proposes an optimal compact method for organizing graph databases. Common subgraphs of the graphs in a database are stored only once in the compact organization of the database, in order to reduce the overall cost of subgraph isomorphism testings from stored graphs to queries during query processing. Then, an exact algorithm and an approximate algorithm for generating significant feature set with optimal order are proposed to construct indices on graph databases. The optimal order on the feature set is to reduce the number of subgraph isomorphism testings during query processing. Based on the compact organization of graph databases, a novel algorithm of testing subgraph isomorphisms from multiple graphs to one graph is presented. Finally, based on all these techniques, a query processing method is proposed. Analytical and experimental results show that the proposed algorithms outperform the existing similar algorithms by one to two orders of magnitude.

1. INTRODUCTION

Recently, large amount of data modeled by graphs, such as the molecular structures of compounds in chemistry, the organizations of entities in images, the topologies of sensor networks, the objects in technical drawings in mechanical engineering field and the social networks, have been collected in various domains. One of the most essential problems for managing large amount of graphs or graph databases is how to efficiently process graph queries.

There are two kinds of queries on graph databases, which are often used in applications. One kind of queries is the *subgraph query*. Given a graph database D and a *subgraph query* Q with query graph q, the answer to Q is the set of $\{g \mid g \in D$ and q is a subgraph of $g\}$. The crucial part of the algorithms for processing subgraph queries is the subgraph isomorphism testing that is NP-complete. Thus, it is intractable to process the subgraph queries. Subgraph query processing has attracted much research attention in last several years, and many algorithms have been proposed [5, 11, 12, 17, 18, 21, 25-28]. To accelerate the processing of subgraph queries, most of the existing algorithms adopt a filtering-and-verification methodology, which first obtains a candidate answer set by pre-generated features from the given graph database and then verifies each candidate by subgraph isomorphism testing.

The other kind of queries on graph databases is the *supergraph query*. Given a graph database D and a supergraph query Q with query graph q, the answer to Q is the set of $\{g \mid g \in D$ and q is a supergraph of $g\}$. This kind of queries is important in many applications. For example, a chemical descriptor has specific properties in chemical reactions. It involves a substructure of many molecular structures and could be modeled by a graph with vertices representing atoms and edges representing the bonds between atoms. Chemists often want to find descriptors in a new molecule graph to predict possible properties of the new molecule. In this case, the chemists can issue a supergraph query with a new molecule as the query graph on the graph database of descriptors to solve their problem.

Though the supergraph query is important in practice, and the filtering-and-verification methodology [18] has shown to be efficient for subgraph query processing on large graph databases, adopting this methodology to process supergraph queries has not been extensively considered yet. To our knowledge, there is only one algorithm, named cIndex [4], to date in the literature for processing the supergraph query by adopting this methodology. cIndex first constructs an index on the *features* that are subgraphs extracted from graph databases and occurring rarely in historical query graphs. During query processing, cIndex avoids a large number of subgraph isomorphism testings by using the filtering-and-verification methodology. In addition, the size of the feature index constructed by cIndex is very small since the features in the index are filtered by the historical queries in the query logs while they are extracted from graph databases.

cIndex has a following disadvantage. The effectiveness and the efficiency of the feature index depend on the historical queries in the query logs. However, the query logs may frequently change over time so that the feature index may be outdated quite often. The mechanism for monitoring and updating feature index involves a large amount of subgraph isomorphism testings. Thus, the overall performance of cIndex is degraded greatly.

EDBT'09, March 24-26, 2009, Saint Petersburg, Russia.

To efficiently process supergraph queries, this paper investigates supergraph query processing from a new angle of view. The proposed new approach in this paper involves the compact storing of graph databases, the constructing of feature indices, and query processing technique rather than only the feature index is considered. In the proposed new approach, graph databases are stored compactly beyond the flat manner, i.e. graphs in graph databases are arranged one by one, to improve the efficiency of query processing (verification). To accelerate the construction of feature indices, a fast algorithm for extracting features from graph databases is proposed without taking the query logs into consideration. In order to further improve the efficiency of query processing (filtering), an optimal order on the feature set is added to the feature indices. To improve the performance of the crucial part of supergraph query processing greatly, a new algorithm of subgraph isomorphism testing from multiple graphs to one graph is proposed.

To examine the performance of the proposed approach, mathematical analysis and extensive experiments were carried out in the paper. The analytical and experimental results show that the proposed approach outperforms cIndex by one to two orders of magnitude.

The main contributions of this paper are as follows.

(1). An optimal compact method for storing graph databases, named *GPTree*, is proposed. Common subgraphs of the graphs in a database are stored only once in the compact organization of the database, to reduce the overall cost of subgraph isomorphism testings from stored graphs to queries during query processing. It is proved that the problem of optimally constructing *GPTree* is NP-complete, and an approximation algorithm with ratio bound 2 is proposed to construct *GPTree* from a graph database.

(2). An exact algorithm and an approximate algorithm for extracting significant features from graph databases are proposed. The extracted features are used to construct the feature indices on graph databases. To reduce the number of subgraph isomorphism testings during query processing, an optimal order on the feature set is determined by mathematics and added to the feature indices.

(3). To improve the performance of the crucial part of supergraph query processing greatly, a new algorithm of subgraph isomorphism testing from multiple graphs to one graph is proposed, named *GPTreeTest*, based on *GPTree*.

The rest of the paper is organized as follows. Section 2 presents the preliminary concepts. Section 3 presents the proposed query processing approach including the compact method for storing graph databases, the feature extracting and ordering algorithms, the subgraph isomorphism testing algorithm from multiple graphs to one graph, and the query processing method. Experimental evaluation is given in Section 4. The related work is surveyed in Section 5. Section 6 concludes the paper.

2. PRELIMINARIES

This paper focuses on the undirected, labeled and connected simple graphs, simply called graph in the rest of the paper. The algorithms proposed in the paper can be easily extended to other kinds of graphs.

DEFINITION 1. A *graph* g is defined as a 4-tuple (V, E, Σ, l), where V is the non-empty set of vertices, $E \subseteq V \times V$ is the set of edges, Σ is the set of labels and $l : V \cup E \rightarrow \Sigma$ is a labeling function assigning a label to a vertex or an edge. The size of a graph g is defined as $\mathrm{size}(g) = |E_g|$, where E_g denotes E of g and $|E_g|$ is the size of the set E_g.

DEFINITION 2. Let $g = (V, E, \Sigma, l)$ and $g' = (V', E', \Sigma', l')$ be two graphs. A *subgraph isomorphism* from g to g' is an injective function $f : V \rightarrow V'$ such that (1) $\forall u \in V$, $l(u) = l'(f(u))$, and (2) $\forall (u, v) \in E$, $(f(u), f(v)) \in E'$ and $l((u, v)) = l'((f(u), f(v)))$.

DEFINITION 3. Let $g = (V, E, \Sigma, l)$ and $g' = (V', E', \Sigma', l')$ be two graphs. An *induced subgraph isomorphism* from g to g' is an injective function $f^I : V \rightarrow V'$ such that (1) $\forall u \in V$, $l(u) = l'(f^I(u))$, (2) $\forall (u, v) \in E$, $(f^I(u), f^I(v)) \in E'$ and $l((u, v)) = l'((f^I(u), f^I(v)))$, and (3) $\forall u, v \in V$, if $(u, v) \notin E$ then $(f^I(u), f^I(v)) \notin E'$.

If there exists a subgraph isomorphism from g to g', g is called a *subgraph* of g', denoted by $g \sqsubseteq g'$, g' is called a *supergraph* of g, and g' *contains* g. If $g \sqsubseteq \tilde{g}$ and $\mathrm{size}(g) + 1 = \mathrm{size}(\tilde{g})$, $\tilde{g}$ is called a *direct supergraph* of g. If $g \sqsubseteq g'$ and $g \neq g'$, g' is called a *proper supergraph* of g. Similarly, if there exists an induced subgraph isomorphism from g to g', g is called an *induced subgraph* of g', denoted by $g \sqsubseteq^I g'$, g' is called an *induced supergraph* of g, and g' *induced-contains* g. Hereafter, we use the term '*sub-iso*' to express 'subgraph isomorphism'.

Given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a graph g, the *support set* of g in D, denoted by $sup_D(g)$, is the set of all the graphs in D that are supergraphs of g, i.e. $sup_D(g) = \{g_i \mid g \sqsubseteq g_i, g_i \in D\}$. $\sigma_D(g) = |sup_D(g)| / |D|$ is called the *support* of g in D. Similarly, the *induced-support set* of g in D, denoted by $sup^I{}_D(g)$,is the set of all the graphs in *D* that are induced supergraphs of g. $\sigma^I{}_D(g) = |sup^I{}_D(g)| / |D|$ is called the *induced-support* of g in D. For a user-specified minimum support (or induced-support) σ, $0 \leq \sigma \leq 1$, a graph g is called *frequent* (or *induced frequent*) in D if $\sigma_D(g) \geq \sigma$ (or $\sigma^I{}_D(g) \geq \sigma$).

The supergraph query processing problem can be defined as follows.

Input: a graph database $D = \{g_1, g_2, ..., g_n\}$ and a query graph q.
Output: $answer(q) = \{g_i \mid g_i \sqsubseteq q, g_i \in D\}$.

3. SUPERGRAPH QUERY PROCESSING

3.1 Overview of the Query Processing Method

The method of processing supergraph queries consists of the following three parts.

Part 1. Graph database organizing. Organize the given graph database compactly, i.e. construct the *GPTree*.

Part 2. Index creating. First, features are extracted from the graph database. Then, an order on the feature set is determined based on the containment relationship between the support sets of the extracted features. Finally, two feature indices on the given graph databases, named *FGPTree* and *CRGraph*, are created based on the algorithm for *GPTree* construction.

Part 3. Query processing. First, the candidate answer set for a given query with query graph q is generated using the *FGPTree* and the *CRGraph*. Then, all candidates (or graphs) in the candidate answer set are verified by testing the subgraph isomorphisms from all the candidates to q using the *GPTree* and the *GPTreeTest* algorithm, i.e. the algorithm of testing subgraph isomorphisms from multiple graphs to one graph, and finally the query answer is obtained.

The time cost of Part 3, i.e. query processing is $T_{query} = T_{filtering} + (|C_q| \times T_{isoCand})$, where $T_{filtering}$ is the time cost of computing the candidate set C_q by testing sub-iso from features to q, and $T_{isoCand}$ is the average time cost of testing sub-iso from each candidate to q. The preprocessing of the given graph databases, i.e.

Part 1 and Part 2, is to minimize the query processing cost T_{query}. Part 1 aims at reducing both $T_{filtering}$ and $|C_q| \times T_{isoCand}$ during query processing. Part 2 is to further reduce $T_{filtering}$.

The details of Part 1, Part 2, and Part 3 are presented in Subsection 3.2, Subsection 3.3 and Subsection 3.4 respectively.

3.2 *GPTree* of a Graph Database

The idea of the proposed compact organization of a graph database is to store all the graphs in the database into one graph with the common subgraphs of the graphs in the database being stored only once. Figure 1 shows a sample graph database containing graphs g_1, g_2, g_3 and g_4, where A, B, C represent three distinct labels of vertices and the labels of edges are ignored for simplicity. Figure 2 illustrates the compact organization of the graph database in Figure 1. In the organization, the triangle in bold solid lines is a common subgraph of g_1 and g_2 and the five-edge subgraph in solid lines is a common subgraph of g_3 and g_4.

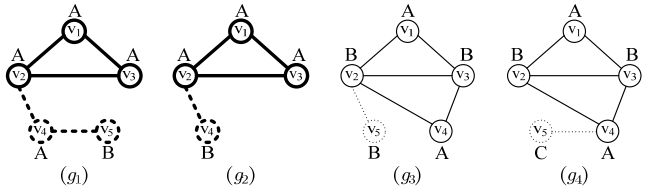

Figure 1: Running Example: a Graph Database

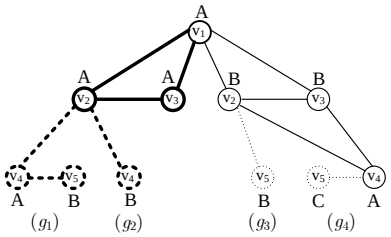

Figure 2: Intuitive Idea of a Compact Organization

Figure 1 and Figure 2 only show the intuitive idea of the compact organization of a graph database. Actually, we propose a data structure, named *GPTree*, to implement the idea. In the following, the structure of the *GPTree* and the algorithm for constructing the *GPTree* are presented.

3.2.1 The Structure of GPTree

To construct a *GPTree* of a graph database, all the graphs in the database need be encoded by an encoding method. This subsection presents a new graph encoding, named *GVCode*.

DEFINITION 4. Given a graph $g = (V, E, \Sigma, l)$, a total order $\prec_V$ on V and a total order $\prec_\Sigma$ on Σ, a *GVCode* of g, denoted as $GVCode(g)$, is a sequence of $\langle \alpha_1, \alpha_2, ..., \alpha_{|V|} \rangle$ defined as follows. For $1 \le j \le |V|$,

(1). α_j is a variable-length subsequence, named the *code* of v_j, whose length is $|\alpha_j| = |\{(v_i, v_j) \mid i < j \text{ and } (v_i, v_j) \in E\}| + 1$,

(2). the first element in α_j is a two-tuple $(j, l(v_j))$, and the other elements are distinct triplets $(i, l(v_i), l((v_i, v_j)))$, where $i < j$ and $(v_i, v_j) \in E$, and

(3). $\forall \tau = (i, l(v_i), l((v_i, v_j)))$ and $\forall \tau' = (i', l(v_{i'}), l((v_{i'}, v_j)))$ in α_j, τ is prior to τ' if one of the following conditions holds:

(a) $l(v_i) \prec_\Sigma l(v_{i'})$,

(b) $l(v_i) = l(v_{i'}) \wedge l((v_i, v_j)) \prec_\Sigma l((v_{i'}, v_j))$, and

(c) $l(v_i) = l(v_{i'}) \wedge l((v_i, v_j)) = l((v_{i'}, v_j)) \wedge (i < i')$.

EXAMPLE 1. *Figure 3 shows the GVCodes of the four graphs in Figure 1 respectively. Assuming* $A \prec_\Sigma B$ *and taking the third code* $\alpha_3 = \langle (3,A), (1,A,-), (2,A,-) \rangle$ *in the GVCode of* g_1 *as example, the first element* $(3,A)$ *in* α_3 *represents the vertex* v_3 *labeled with* A *in* g_1. *The elements* $(1,A,-)$ *and* $(2,A,-)$ *represent edges* (v_1, v_3) *and* (v_2, v_3), *and both the labels of* v_1 *and* v_2 *are* A. *The ignoring of the labels on* (v_1, v_3) *and* (v_2, v_3) *is denoted by '*$-$*'.* $(1,A,-)$ *is prior to* $(2,A,-)$ *because condition (c) of (3) in Definition 4 holds.*

Please note that a graph may have multiple *GVCode*s due to the variety of total orders on the vertex set of a graph.

It follows that a prefix of a *GVCode* of a graph corresponds to an induced subgraph of the graph, and a common prefix of *GVCode*s of two graphs corresponds to a common induced subgraph of the graphs.

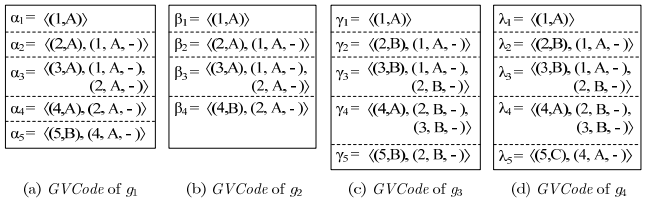

Figure 3: Four *GVCode*s

GPTree. A *GPTree* of a graph database is a trie constructed by the *GVCodes* of all the graphs in the database, by taking a code in a *GVCode* as a basic unit of the sequence. For example, Figure 4 shows a sample *GPTree* constructed by the graph database in Figure 1, where n_i for $1 \le i \le 11$ is defined in (b) of Figure 4. In the *GPTree*, the path of $\langle n_1, n_2, n_3, n_4, n_5 \rangle$ corresponds to g_1 or the *GVCode* of g_1 in Figure 3, denoted by the set of graph-IDs attached to n_5. The path of $\langle n_1, n_2, n_3, n_6 \rangle$ corresponds to g_2 or the *GVCode* of g_2, denoted by the set of graph-IDs attached to n_6. The path of $\langle n_1, n_2, n_3 \rangle$ represents a common prefix of the *GVCode*s of g_1 and g_2 or a common induced subgraph of g_1 and g_2.

Please note that a path from the root of a *GPTree* to any node of the *GPTree* may express multiple isomorphic graphs.

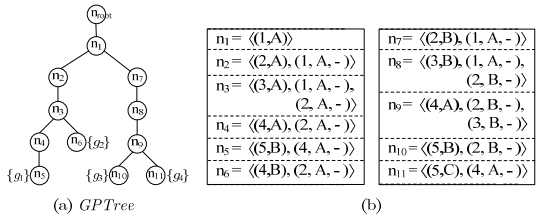

Figure 4: A *GPTree*

3.2.2 The Algorithm for Constructing GPTree

To construct a *GPTree* of a graph database, each graph in the database is encoded into a *GVCode*. Because multiple *GVCode*s may be generated from a graph as mentioned in Section 3.2.1, there may be multiple *GPTree*s that can be constructed from a database. The cost of sub-iso testing from multiple graphs in a given *GPTree* to one graph is first analyzed as follows.

Let us first consider the supergraph query with query graph q_1 in Figure 5 on the graph database in Figure 1. To process the query q_1 using naïve method, sub-iso testings are performed one by one, i.e. test sub-iso from g_1 to q_1, g_2 to q_1, …, and g_4 to q_1

one by one. It need perform 4 sub-iso testings. To process q_1 using the *GPTree* in Figure 4, the sub-iso testing from g_1 to q_1 is performed first. It will be found that the triangle in bold in Figure 2 is not a subgraph of q_1, and thus g_2 is not a subgraph of q_1 because the triangle is a subgraph of g_2. Therefore, the sub-iso testing from g_2 to q_1 is avoided. Similarly, after the sub-iso testing from g_3 to q_1, it is known that the five-edge subgraph in solid lines in Figure 2 is not a subgraph of q_1, and thus g_4 is not a subgraph of q_1 without sub-iso testing from g_4 to q_1. Thus, 2 sub-iso testings are saved compared to the naïve method.

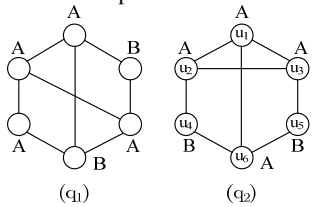

Figure 5: Running Example: Two Queries

Now let us consider the supergraph query with query graph q_2 in Figure 5 on the graph database in Figure 1. Similar to the results of q_1, it need perform 4 sub-iso testings to process q_2 using naïve method. To process q_2 using the *GPTree* in Figure 4, a subgraph isomorphism from the triangle in bold in Figure 2 to q_2 will be found. The sub-iso from g_1 and g_2 to q_2 can be found by extending the sub-iso from the triangle to q_2, respectively. In this way, the sub-iso testing from the triangle to q_2 is just performed once rather than twice by using naïve method. Similarly, the sub-iso testing from the five-edge subgraph in solid lines in Figure 2 to q_2 is just performed once. Thus, 2 sub-iso testings are saved compared to the naïve method.

In general, let the *GVCode*s of all the graphs in a graph set GS share a common prefix, cp, in a *GPTree* of GS. Then, the cost of sub-iso testings saved by the common prefix cp in the *GPTree* is

$$c(\mathrm{len}(cp)) \times (|\, GS \,| - 1), \qquad (1)$$

where $\mathrm{len}(cp)$ is the number of the codes in cp, and $c(x)$ is the average cost taken by a sub-iso testing from a subgraph with x vertices to a query graph. Please note that the number of vertices has crucial impact on the cost of sub-iso testing.

In the following discussion, $D = \{g_1, g_2, ..., g_n\}$ is a graph database, and Q is a supergraph query with query graph q.

Given a *GPTree* of D, let us consider the cost of sub-iso testings saved by a set of common prefixes in the *GPTree* during the processing of Q. Let the *GVCode*s of the distinct graphs g_{11}, g_{12} ,..., and g_{1m_1} in D share the common prefix cp_1 in the *GPTree*, the *GVCode*s of the distinct graphs g_{21}, g_{22} ,..., and g_{2m_2} in D share the common prefix cp_2 ,..., and the *GVCode*s of the distinct graphs g_{k1}, g_{k2} ,..., and g_{km_k} in D share the common prefix cp_k in the *GPTree*. Here $\sum_{j=1,2,...,k} n_j = |\, D \,|$. Based on Eq. (1), the cost saving by the common prefix cp_j is $c(\mathrm{len}(cp_j)) \times (n_j - 1)$. Therefore, the overall cost saving, denoted by C_{sav}, of sub-iso testings during the processing of Q using the *GPTree*, by cp_j for $1 \le j \le k$, is no less than $\sum_{j=1,2,...,k} c(\mathrm{len}(cp_j)) \times (n_j - 1)$.

As mentioned in Section 3.2.1, a graph g may have multiple *GVCode*s due to the variety of total orders on the vertex set of g. Since a *GPTree* of D is constructed by the *GVCode*s of all the graphs in D, an optimal *GPTree* should be constructed by selecting a best *GVCode* for each graph in D such that the overall cost saving C_{sav} is maximized.

A prefix of a *GVCode* of a graph corresponds to an induced subgraph of the graph as mentioned in Section 3.2.1. Conversely, given an induced subgraph of a graph, it can generate a prefix of a *GVCode* of the graph. Thus, to get a *GVCode* of a graph $g = (V_1, E_1, \Sigma_1, l_1)$, we can first select an induced subgraph $ig = (V_2, E_2, \Sigma_2, l_2)$ of g, then generate the *GVCode* of ig, and finally extend the *GVCode* of ig based on an induced subgraph isomorphism from ig to g, to obtain the *GVCode* of g by adding the codes of the vertices in $V_1 - V_2$.

To solve the problem of constructing *GPTree*, the following steps are performed.

Step 1. Select optimal induced subgraph for each graph in D.

Step 2. Generate *GVcode*s for all the graphs in D using the induced subgraphs obtained in Step 1.

Step 3. Construct the *GPTree* of D using the *GVcode*s generated in Step 2.

Despite the lack of information about induced-containment relationship among all induced subgraphs of the graphs in a database, which is too expensive to be obtained, we focus our attention on selecting the best induced subgraph for each graph in the database such that $\sum_{j=1,2,...,k} c(|\, V_{cig_j} \,|) \times (n_j - 1)$ is maximized, where for $1 \le j \le k$, cig_j is the common induced subgraph selected for the distinct graphs g_{j1}, g_{j2}, ..., and g_{jn_j} in D, and $\sum_{j=1,2,...,k} n_j = |\, D \,|$. This problem is called *induced subgraph selecting problem* in the paper, defined as follows.

Input: Graph database D.

Output: a sequence $\langle ig_1, ig_2, ..., ig_{|D|} \rangle$, where ig_i is the induced subgraph selected for the graph g_i in D for $1 \le i \le |\, D \,|$.

Objective: Maximize $\sum_{i=1}^{|D|} c(|\, V_{ig_i} \,|) - \sum_{ig \in \bigcup_{1 \le i \le |D|} \{ig_i\}} c(|\, V_{ig} \,|)$.

Please note that

$$\sum_{j=1,2,...,k} c(|\, V_{cig_j} \,|) \times (n_j - 1) = \sum_{i=1}^{|D|} c(|\, V_{ig_i} \,|) - \sum_{ig \in \bigcup_{1 \le i \le |D|} \{ig_i\}} c(|\, V_{ig} \,|).$$

In the following discussion, IG is the set of all the induced subgraphs of all the graphs in D.

Let S be a finite set, $f : S \to \mathbb{R}^+$ be a positive function, and C be a collection of subsets of S. We can create a bijection between S and IG, a bijection between C and D, and a bijection between $f(e)$ and the cost function $c(|\, V_{ig} \,|)$ in Eq. (1), where $ig \in IG$ and V_{ig} is the vertex set of ig. If e corresponds to ig under the bijection between S and IG, then $f(e) = c(|\, V_{ig} \,|)$. If a subset $S_i = \{e_1, e_2, ..., e_l\}$ in C corresponds to a graph g_i in D, then g_i is an induced supergraph of ig_1, ig_2 ,..., ig_l but none of other graphs in IG, where ig_1, ig_2 ,..., and ig_l are the induced subgraphs corresponding to e_1, e_2 ,..., and e_l, respectively. Under these bijections, finding a sequence $\langle ig_1, ig_2, ..., ig_{|D|} \rangle$ is to find a sequence $\langle e_1, e_2, ..., e_{|C|} \rangle$, where for $1 \le i \le |\, C \,|$, e_i is a element of the subset S_i in C, e_i corresponds to ig_i, and

$$\sum_{i=1}^{|D|} c(|\, V_{ig_i} \,|) - \sum_{ig \in \bigcup_{1 \le i \le |D|} \{ig_i\}} c(|\, V_{ig} \,|) = \sum_{i=1}^{|C|} f(e_i) - \sum_{e \in \bigcup_{1 \le i \le |C|} \{e_i\}} f(e).$$

Thus, the induced subgraph selecting problem can be defined as follows.

Input: a finite set S, a positive cost function $f : S \to \mathbb{R}^+$ and a collection C of subsets of S.

Output: a sequence $\langle e_1, e_2, ..., e_{|C|} \rangle$, where e_i is a element of the subset S_i in C for $1 \le i \le |\, C \,|$.

Objective: Maximize $\sum_{i=1}^{|C|} f(e_i) - \sum_{e \in \bigcup_{1 \le i \le |C|} \{e_i\}} f(e)$.

To prove that the induced subgraph selecting problem is NP-hard, we first define the *uniquely hitting set problem* as follows.

Input: a collection C of subsets of a finite set S and a positive integer $K \leq |S|$.

Output: If there is a sequence $\langle e_1, e_2, ..., e_{|C|} \rangle$ such that $|\bigcup_{i=1,2,...,|C|} \{e_i\}| \leq K$ then output 'ture', otherwise output 'false', where e_i is a element of the subset S_i in C for $1 \leq i \leq |C|$.

LEMMA 1. The uniquely hitting set problem is NP-complete.

PROOF (sketch). It is easy to prove that the problem belongs to NP. The hitting set problem is NP-complete [9]. The hitting set problem can reduce to the uniquely hitting set problem in polynomial time. ■

THEOREM 1. The induced subgraph selecting problem is NP-hard.

PROOF (sketch). The uniquely hitting set problem is a special case of the induced subgraph selecting problem (the value of the function f is equivalent to 1). Thus, the induced subgraph selecting problem is NP-hard. ■

Since the induced subgraph selecting problem is NP-hard, we develop an approximation algorithm with ratio bound 2 to solve the problem.

In the following algorithm, the element selected from a subset S_i in C is called the uniquely hitting element of S_i, denoted as $uhe(S_i)$. The data structure $\mathcal{A}$ is used to store the subsets that have already been selected and need not be considered subsequently. Let $\mathcal{H}(e, \mathcal{Z}) = \{S_i \mid e \in S_i,\ S_i \in \mathcal{Z}\}$, where $e \in S$ and $\mathcal{Z} \subseteq C$. The proposed algorithm is as follows.

Input: a finite set S, a positive cost function $f: S \rightarrow \mathbb{R}^+$ and a collection C of subsets of S.
Output: the sequence $\langle uhe(S_1), uhe(S_1), ..., uhe(S_{|C|}) \rangle$ where $S_i \in C$ for $1 \leq i \leq |C|$.

1: $\mathcal{A} \leftarrow \varnothing$;
2: **while** $\exists e \in S$ such that $|\mathcal{H}(e, C - \mathcal{A})| > 1$ **do**
3: select $e' \in S$ with largest $f(e')$ such that $|\mathcal{H}(e', C - \mathcal{A})| > 1$;
4: **foreach** $S_i \in \mathcal{H}(e', C - \mathcal{A})$ **do**
5: $uhe(S_i) \leftarrow e'$;
6: $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{H}(e', C - \mathcal{A})$;
7: **while** $\exists e \in S$ such that $|\mathcal{H}(e, C - \mathcal{A})| = 1$ **do**
8: select $e'' \in S$ with largest $f(e'')$ such that $|\mathcal{H}(e'', C - \mathcal{A})| = 1$;
9: **foreach** $S_i \in \mathcal{H}(e'', C - \mathcal{A})$ **do**
10: $uhe(S_i) \leftarrow e''$;
11: $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{H}(e'', C - \mathcal{A})$;

THEOREM 2. The above algorithm can approximate the optimal selecting with ratio bound 2.

PROOF. (sketch) The objective $\sum_{i=1}^{|C|} f(e_i) - \sum_{e \in \bigcup_{1 \leq i \leq |C|} \{e_i\}} f(e)$ can be regarded as the sum of the contribution value from each subset S_i in C for $1 \leq i \leq |C|$, and the contribution value from S_i is equal to either $f(uhe(S_i))$ or 0, which depends on the distribution of $uhe(S_i)$ among all S_i in C.

First, we consider all iterations involving Lines 2-6 in the above algorithm. Let the first element we selected be e_{i_1}. By analyzing and verifying two complementary cases that whether all the subsets containing e_{i_1}, i.e. $\mathcal{H}(e_{i_1}, C)$, have an element with larger cost function value than $f(e_{i_1})$, we claim that, for the optimal solution the sum of the contribution values from all subsets in $\mathcal{H}(e_{i_1}, C)$ must be no more than $f(e_{i_1}) \times |\mathcal{H}(e_{i_1}, C)|$.

During the iterating within Lines 2-6, let the next element to be selected be e_{i_2}. We claim that, for the optimal solution the sum of the contribution values from all subsets in $\mathcal{H}(e_{i_2}, C - \mathcal{A})$ is no more than $f(e_{i_2}) \times |\mathcal{H}(e_{i_2}, C - \mathcal{A})|$ on the basis of the bound for the union of subsets of previously selected elements.

Therefore, after performing all the iterations involving Lines 2-6, we obtain the partial solution produced by the greedy strategy, whose objective is equal to $\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}| - 1)$, and an upper bound for the sum of the contribution values from all the subsets in $\mathcal{A}$ for any instances (or the optimal solution), which is $\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}|)$, where p_1 is the number of elements selected, $\mathcal{A}_k = \bigcup_{l=1,2,...,k} \mathcal{H}(e_{i_l}, C)$ and $\mathcal{A}_0 = \varnothing$.

Second, considering all iterations involving Lines 7-11 in the above algorithm, we have that, on the basis of the bound for subsets in $\mathcal{A}_{p_1}$ given above, i.e. $\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}|)$, for the optimal solution the sum of the contribution values from all subsets $\mathcal{B}_{p_2}$ (or $C - \mathcal{A}_{p_1}$), is equivalent to 0, where p_2 is the number of elements selected in the iterations involving Lines 7-11 and $\mathcal{B}_k = \bigcup_{l=1,2,...,k} \mathcal{H}(e_{j_l}, C)$. Obviously, the objective of the partial solution produced in the second part is equal to 0.

In summary, after all iterations involving Lines 2-11 in the algorithm are conducted, we obtain a solution with the objective $\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}| - 1)$ and an upper bound for the sum of contribution values from all subsets in C for any instances (or the optimal solution), which is $\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}|)$.

So the approximation ratio is

$$R = \frac{opt}{solution} \leq \frac{\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}|)}{\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}| - 1)}$$

$$= 1 + \frac{\sum_{l=1,2,...,p_1} f(e_{i_l})}{\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (|\mathcal{A}_l - \mathcal{A}_{l-1}| - 1)}.$$

It is clear that for $\forall l, 1 \leq l \leq p_1$, $|\mathcal{A}_l - \mathcal{A}_{l-1}| \geq 2$ due to the greedy selection strategy in Lines 2-6 in the above algorithm. So

$$R \leq 1 + \frac{\sum_{l=1,2,...,p_1} f(e_{i_l})}{\sum_{l=1,2,...,p_1} f(e_{i_l}) \times (2-1)} \leq 2. \blacksquare$$

The algorithm for *GPTree* construction, called BuildGPTree, is shown in Algorithm 1. After Lines 1-13 are carried out, Step 1 for the construction finishes; after Line 14 ends, Step 2 and 3 finishes.

Algorithm 1 BuildGPTree (D , $\sigma^I{}_T$)

Input: a graph database D and a minimum threshold $\sigma^I{}_T$
Output: the *GPTree*

1: obtain the set FIG of frequent induced subgraphs of the graphs in D, where the minimum induced-support is $\sigma^I{}_T$;
2: $CP \leftarrow \varnothing$, $S \leftarrow \varnothing$;
3: **while** $\exists ig \in FIG$, s.t., $|sup^I{}_D(ig) - S| > 1$ **do**
4: select $ig' \in FIG$ with largest $|V_{ig'}|$ such that $|sup^I{}_D(ig') - S| > 1$;
5: $cp \leftarrow ig'$;
6: $cp.GRP \leftarrow \{\langle g, vseq \rangle \mid \langle g, vseq \rangle \in ig'.SUP^I \wedge g \notin S\}$;
7: $CP \leftarrow CP \cup \{cp\}$, $S \leftarrow S \cup sup^I{}_D(ig')$;
8: **while** $\exists ig \in FIG$, s.t., $|sup^I{}_D(ig) - S| = 1$ **do**
9: select $ig'' \in FIG$ with largest $|V_{ig''}|$ such that $|sup^I{}_D(ig'') - S| = 1$;
10: $cp \leftarrow ig''$;
11: $cp.GRP \leftarrow \{\langle g, vseq \rangle \mid \langle g, vseq \rangle \in ig''.SUP^I \wedge g \notin S\}$;
12: $CP \leftarrow CP \cup \{cp\}$, $S \leftarrow S \cup sup^I{}_D(ig'')$;
13: complete the sequence of all vertices of each graph in D;
14: according to the sequence of all vertices of each graph in $cp.GRP$, where $cp \in CP$, obtain the corresponding *GVCode*, and then construct the trie of *GPTree*;

BuildGPTree first obtains the set FIG of frequent induced subgraphs in D (Line 1). Please note that due to the exponential

amount of all induced subgraphs of the graphs in a database generally, we set a minimum induced-support threshold and obtain frequent induced subgraphs instead. For each ig in FIG, during the mining, an induced sub-iso, denoted by $\varphi(ig, g_j)$, from ig to each graph $g_j \in sup^I{}_D(ig)$ is detected; and for each such g_j,a sequence,denoted by $vseq_j$,of the vertices of g_j involving $\varphi(ig, g_j)$ is retrieved. $ig.SUP^I = \{\langle g_j, vseq_j\rangle \mid g_j \in sup^I{}_D(ig)\}$ is retrieved for each ig in FIG. Then, BuildGPTree conducts induced subgraph selecting from FIG following the above greedy algorithm (Lines 2-12). After that, a pair $\langle g, vseq\rangle \in \bigcup_{cp\in CP} cp.GRP$ represents that g takes $vseq$ as the prefix of the sequence of all the vertices of g. Then, BuildGPTree generates a sequence of the remaining vertices of each graph in D and completes the sequence of all vertices of the graph (Line 13). After the order on the vertex set of each graph has been found, *GVCode* of each graph is obtained according to the order. The last step is to construct the trie of *GPTree*.

Complexity Analysis of *GPTree* Construction. Let $T_{\text{ig-m}}$ and $S_{\text{ig-m}}$ be the time and space usage by frequent induced subgraph mining.The time usage of BuildGPTree apart from $T_{\text{ig-m}}$ is $O(\mid D \mid \times \mid FIG \mid + \sum_{g\in D}(\mid V_g \mid + \mid E_g \mid))$. The memory usage apart from $S_{\text{ig-m}}$ is $O(\sum_{ig\in FIG}(\mid sup^I{}_D(ig) \mid \times \mid V_{ig} \mid) + \mid D \mid \times \mid FIG \mid + \sum_{g\in D}(\mid V_g \mid + \mid E_g \mid))$. Also, *vseq*s could be stored in disk for limited memory.

3.3 Indices on Graph Databases

3.3.1 Feature Generation

Two methods are proposed for feature generation. One is an exact method for selecting all significant frequent subgraphs, and the other is an approximate method for faster selecting a subset of significant frequent subgraphs, which has comparable filtering power to that of all significant frequent subgraphs in practice.

Given a database D, a feature set F of D, and a query graph q, the candidate answer set is $C_q = D - \bigcup_{f\not\subseteq q, f\in F} sup_D(f)$. All graphs in D that contain a feature, which is not contained in q, cannot be in results. Therefore, we prefer to select such subgraphs of the graphs in D that are less likely to be contained in query graphs, as features. Thus, relatively large subgraphs are preferred.

Frequent subgraphs expose the intrinsic characteristics of a graph database and have been verified to be good choices as features for subgraph queries. For supergraph queries, the filtering power of each frequent subgraph is analyzed in the following. It will be seen that a subset of all frequent subgraphs selected as features are able to achieve the same filtering power compared to all of them. Given two distinct frequent subgraphs g and g' such that $g \sqsubseteq g'$, if $\mid sup(g) \mid = \mid sup(g') \mid$, it is advisable to select g' as a feature rather than both of them or only g, since the larger subgraph g' is less likely to be contained in query graphs. In this way, g' is more *helpful* than g for filtering during query processing, and g is not *helpful* w.r.t. g'. Thus, if g' is a feature, then g need not be a feature. In the case of multiple subgraphs, for a subgraph g and a set of subgraphs $SG = \{g_1, g_2, \ldots, g_k\}$ such that for $1 \le i \le k$, $g \sqsubseteq g_i$ and $g \ne g_i$, if $sup(g) = \bigcup_{i=1,2,\ldots,k} sup(g_i)$, then g is not *helpful* w.r.t. SG. The reason is that if all subgraphs in SG are features, then selecting g as a feature is not able to improve the overall filtering power any more, i.e. identifying less candidates. The *significance metric* δ w.r.t. g is defined as $\delta = \frac{\mid sup(g)\mid}{\mid\bigcup_{i=1,2,\ldots,m} sup(f_i)\mid}$ [1], where g is a subgraph, and f_1, f_2,…, and f_m are all the features that contain g. In order to generate features, i.e., to obtain *significant* frequent subgraphs, we set a minimum significance threshold δ_{min} and select all the frequent subgraphs with significance no less than δ_{min} as features.

In the space of frequent subgraph patterns in a graph database D, the process of selecting significant frequent subgraphs as features is discussed as follows. For two frequent subgraphs g and g', such that $g \sqsubseteq g'$, $g \ne g'$ and $sup_D(g) = sup_D(g')$, let the data structure F store all the features selected till now, and let all the selected features in F that contain g' be f_1, f_2,…, and f_m. Given δ_{min}, δ_{min} >1, Case 1, if $\frac{\mid sup(g')\mid}{\mid\bigcup_{i=1,2,\ldots,m} sup(f_i)\mid} \ge \delta_{min}$, then g' should be added into F before g being added, because g' is more helpful than g for filtering; and then g should not be added into F, because after g' is added into F, g' is a feature in F that contains g, and $\frac{\mid sup(g)\mid}{\mid(\bigcup_{i=1,2,\ldots,m} sup(f_i))\ \bigcup\ sup(g')\mid} = 1 < \delta_{min}$. Case 2, otherwise, i.e. $\frac{\mid sup(g')\mid}{\mid\bigcup_{i=1,2,\ldots,m} sup(f_i)\mid} < \delta_{min}$, then $\frac{\mid sup(g)\mid}{\mid\bigcup_{i=1,2,\ldots,m} sup(f_i)\mid} < \delta_{min}$, thus both g' and g are not significant and are not added into F. So in any case g should be discarded. In the existing works, the set CSG of *closed frequent subgraphs* [24] is defined as $CSG = \{g \mid g \in FSG$ and $\nexists g' \in FSG$ s.t. $g \sqsubseteq g' \wedge g \ne g' \wedge sup(g) = sup(g')\}$, where FSG is the set of all frequent subgraphs. Therefore, any significant frequent subgraph selected here must be a closed frequent subgraph. Please note that closed frequent subgraphs may be orders of magnitude less than all frequent subgraphs in practice [24].

The Exact algorithm for generating feature set from a database consists of the following two steps. Step 1, mine closed frequent subgraphs. Step 2, refine subgraphs obtained in Step 1, i.e. eliminate insignificant frequent subgraphs according to a user-specified δ_{min}, which proceeds in a level-wise manner from large size to small size. It follows that all maximal frequent subgraphs are selected as features. According to depth-first search order in the mining algorithm, the containment relationship among some closed frequent subgraphs in Step 1 can be obtained, which need not be examined again in Step 2; thus, the number of sub-iso testings is reduced in the exact algorithm for feature generation.

The approximate algorithm for generating feature set directly mines features instead of refining after mining. During mining of closed frequent subgraphs, for a closed frequent subgraph g in the space of closed frequent subgraph patterns in D,if $\frac{\mid sup_D(g)\mid}{\mid\bigcup_{\tilde{g}} sup_D(\tilde{g})\mid} \ge \delta_{min}$, where $\tilde{g}$ denotes all the direct supergraphs of g that are frequent, i.e., the denominator is the cardinality of union over all support sets of such $\tilde{g}$, then g is selected as a feature; otherwise, it is discarded. This condition can be directly embedded in closed frequent subgraph mining algorithms without extra subgraph isomorphism testings, i.e. only the union of support sets is additionally computed. Thus, we can fast generate a feature set without costly refining step, i.e. Step 2 of the exact algorithm above.

For a subgraph g in the space of closed frequent subgraph patterns in D and any subgraph set S, we have $\mid\bigcup_{g'} sup_D(g')\mid \le \mid\bigcup_{\tilde{g}} sup_D(\tilde{g})\mid$, where $\tilde{g}$ denotes all frequent direct supergraphs of g, and g' denotes the frequent proper supergraphs of g that belong to S. Thus, we conclude that each subgraph in the feature set obtained by using the approximate algorithm above must be in the feature set obtained by the exact algorithm, i.e., the feature set generated by the approximate algorithm is a subset of that generated by the exact algorithm. The approximate algorithm is very

[1] If the denominator is zero, we define that the left fraction is equal to a very large number that is always larger than any δ_{min}.

suitable for the scenarios where closed frequent subgraphs are of large amount, which shows approximate filtering power in experiments during query processing. Please note that in order to enable the graphs in D that do not contain any frequent subgraphs to be supported in the filtering step, for both exact and approximate feature generation methods, all subgraphs, i.e. both frequent and infrequent subgraphs, of the graphs in D with size less than 4 are examined during mining. In other words, the minimum support threshold is set to be $1/|D|$ when the size of a subgraph pattern is less than 4.

3.3.2 Feature Ordering and Index Structures

Although all features extracted are significant, in the filtering step of processing queries there probably exist unnecessary sub-iso testings from features to query graphs. Appropriately ordering features could reduce the number of such unnecessary sub-iso testings. Considering two features f_1 and f_2, s.t., $sup_D(f_1) \supseteq sup_D(f_2)$, for a query graph q, if $f_1 \not\subseteq q$ we can filter out all graphs in $sup_D(f_1)$ immediately, and then, f_2 need not be examined because no matter it is contained by q or not we cannot filter out any other graphs by $sup_D(f_2)$. Thus we save one subgraph isomorphism testing from a feature to the query graph. Conversely, we have to perform two sub-iso testings with no saving. This example shows that the order on the feature set can influence the overall cost of query processing.

Let S be a finite set, C be a collection of subsets of S and $g: C \to [0,1]$ be a nonnegative function. We can create a bijection between S and the graph database D, and a bijection between C and the feature set F. For $\forall k,l, 1 \le k,l \le |F|$, if the subset $A_k \in C$ then $f_k \in F$ (and vice versa), and if $A_k \supseteq A_l$ then $sup_D(f_k) \supseteq sup_D(f_l)$ (and vice versa), where A_k corresponds to f_k (A_l to f_l) under the bijection between C and F.

Let $h: 2^C \to [0,1]$ be a function, where $h(\{A_1, A_2, ..., A_m\})$ is equal to the joint probability that all features in the set of $\{f_1, f_2, ..., f_m\}$ are subgraphs of a given query graph, where A_k corresponds to f_k for $1 \le k \le m$. Given an order on C, i.e. $\langle A_{j_1}, A_{j_2}, ..., A_{j_{|C|}} \rangle$, where $A_{j_k} \in C$ for $1 \le k \le |C|$, let $g: C \to [0,1]$ be a function, where $g(A_{j_k}) = 1 - h(\{A_{j_k} | A_{j_k} \in C, A_{j_k} \supseteq A_{j_t}, k < t\})$. Thus, the problem of determining the optimal order on a feature set can be defined as follows.

Input: a finite set S, a collection C of the subsets of S, and a nonnegative function $g: C \to [0,1]$.

Output: an order $\prec$ on C, i.e. $\langle A_{j_1}, A_{j_2}, ..., A_{j_{|C|}} \rangle$, where $A_{j_k} \in C$ for $1 \le k \le |C|$.

Objective: Maximize $\sum_{k=1,...,|C|} g(A_{j_k})$.

Given a query q, the objective $\sum_{k=1,...,|C|} g(A_{j_k})$ is the expected number of subgraph isomorphism testings that can be saved compared to testing one by one. The following theorem is presented to expose the case in which the maximum objective can be obtained.

THEOREM 3. An order $\prec^*$ on C is *optimal*, if and only if, for any two subsets A_i and A_j in C, $A_i \prec^* A_j$ if $A_i \supseteq A_j$ and $A_i \neq A_j$. ■

Also it follows that arranging features in non-ascending order in terms of their support could derive an optimal order.

CRGraph. The CRGraph index structure is a directed acyclic graph (DAG). Given D and the feature set F of D, for two features $f_1, f_2 \in F$, if $sup_D(f_1) \supseteq sup_D(f_2)$ and $\nexists f' \in F$ s.t. $sup_D(f_1) \supseteq sup_D(f') \wedge sup_D(f') \supseteq sup_D(f_2)$, then we say $sup_D(f_1)$ direct contains $sup_D(f_2)$. This direct containment relationship between support sets of features defines a partial order relation. Then, the support sets of features can be conceptually organized into a lattice, which is represented by a DAG with vertices representing corresponding features and edges representing the direct containment relationship.

Complexity Analysis of *CRGraph* Construction. *CRGraph* construction is not explicitly given due to its commonality. The time complexity of *CRGraph* construction is $O(|F|^2 \times |D|)$, and the space complexity of it is $O(|F|^2)$. Using *CRGraph*, an optimal order can be yielded by topological ordering on it with the time complexity of $O(|F| + |E_{CRGraph}|)$, where $E_{CRGraph}$ is the edge set of *CRGraph*.

FGPTree. After obtaining an ordered feature set, we organize all features using a similar strategy as *GPTree* to obtain a quasi '*GPTree*' structure, called *FGPTree*. In *FGPTree*, some but not all common prefixes of *GVCode*s of distinct features are combined, and the order on the feature set is preserved. Based on the *GPTree* construction algorithm and the *CRGraph*, *FGPTree* construction is not difficult to conduct, thus it is not explicitly elaborated here.

3.3.3 Discussions on Graph Database Organizing and Index Creating

The preprocessing in our approach includes organizing graph database into the *GPTree*, and creating indices of the *CRGraph* and the *FGPTree*. The process of the specified frequent induced subgraph mining on database for *GPTree* construction is merged with the specified closed frequent subgraph mining for feature generation in the following manner. In the progress of the integrated mining, if a search branch can be pruned by the conditions of one mining algorithm, the other mining algorithm is solely conducted along the branch; otherwise, the examinations for these two mining schemes are both conducted in the original way.

In summary, the preprocessing in our approach consists of two steps. The *GPTree* of a given graph database is constructed first. Simultaneously, the initial feature set is generated. Then, feature selection from the initial feature set is carried out if exact algorithm for feature generation is adopted, and the initial feature set does not change if approximate method for feature generation is used; features are ordered and the *CRGraph* and the *FGPTree* are created.

3.4 Query Processing

3.4.1 Subgraph Isomorphism Testing from Many to One

The process of subgraph isomorphism testing from a small graph $g_s = (V_s, E_s, \Sigma_s, l_s)$ to a large one $g_L = (V_L, E_L, \Sigma_L, l_L)$ consists in the search for a subgraph isomorphism, i.e. injective function, which associates all vertices in V_s to some distinct ones in V_L. A subgraph isomorphism is represented by a set of ordered pairs of matched vertices, and the two matched vertices in each pair are from g_s and g_L, respectively.

The sub-iso testing from g_s, encoded into $GVCode(g_s)$, to g_L involves a process of tree search with backtracking. The state space is constructed following the way that the vertices of V_s are examined in the order consistent with $GVCode(g_s)$, and g_L is compared in a vertex growth way against g_s one vertex after another. Each state t in search space is associated with a partial subgraph isomorphism, $psi(t)$, which is a set of ordered pairs of matched vertices corresponding to the path from the root state to the state t in the search tree. A transition between two states corresponds to the addition of a new pair of matched vertices.

Depth-first search on the search space is adopted. For each state t, we compute a candidate pair set $CPS(t)$ consisting of pairs of potential subsequently-matched vertices. For all pairs in $CPS(t)$, the first components are the same and equal to the next vertex in g_s in the order on vertex set in $GVCode(g_s)$, and the second components correspond to each of vertices in g_L that is not involved by $psi(t)$, respectively. We denote $V_s(t) = \{v \mid (v,\mu) \in psi(t)\}$, s.t., $V_s(t) \subseteq V_s$ and $V_L(t) = \{\mu \mid (v,\mu) \in psi(t)\}$, s.t., $V_L(t) \subseteq V_L$. The neighbor set of a vertex $v \in V_s$ is represented by $N_s(v)$, and the neighbor set of a vertex $\mu \in V_L$ is denoted as $N_L(\mu)$.

Example 2. *Consider the process of sub-iso testing from g_2 in Figure 1 to q_2 in Figure 5. For a state t, the partial sub-iso $psi(t) = \{(v_1,u_1), (v_2,u_2),(v_3,u_3)\}$ denotes a sub-iso from the bold-line triangle subgraph of g_2 to a triangle subgraph of q_2. Here $V_s(t) = \{v_1,v_2,v_3\}$, and $V_L(t) = \{u_1,u_2,u_3\}$. $CPS(t) = \{(v_4,u_4), (v_4,u_5), (v_4,u_6)\}$.*

At each state during searching, $CPS(t)$ is refined by removing the vertex pairs that cannot be expanded to a subgraph isomorphism subsequently. The conditions used to refine $CPS(t)$ are called *Pruning Conditions*. It is apparent that a vertex in V_s cannot match one in V_L with a different label. For a pair of vertices (v,μ), where $v \in V_s$, $\mu \in V_L$, we have the following pruning condition, which is denoted by $PruCond_v$.

PRUNING CONDITION 1. $l_s(v) \neq l_L(\mu)$.

For a pair of vertices (v,μ) in $CPS(t)$, where $v \in V_s$, $\mu \in V_L$, the pruning condition by relationship between (v,μ) and the pairs in $psi(t)$ is given as follows. Note that let t' be the state transited by t after adding (v,μ) into $psi(t)$. It is clear that each pair in $CPS(t)$ does not satisfy Pruning Condition 1.

PRUNING CONDITION 2. $\exists x \in N_s(v) \cap V_s(t)$, $\xi \notin N_L(\mu)$ or $l_s(x,v) \neq l_L(\xi,\mu)$, where $(x,\xi) \in psi(t)$.

The Pruning Condition 2 ensures that the partial subgraph isomorphism $psi(t')$ itself is a subgraph isomorphism from the subgraph of g_s induced by the vertices in $V_s(t')$ to g_L. The Pruning Condition 2 is denoted as $PruCond_{ig}$. For example, after refined by Pruning Condition 1, the $CPS(t)$ in Example 2 is equal to $\{(v_4,u_4),(v_4,u_5)\}$; and then, after refined by Pruning Condition 2, the $CPS(t)$ is equal to $\{(v_4,u_4)\}$.

During searching, the algorithm of sub-iso testing step by step picks up vertices of g_s and constructs a growing induced subgraph of g_s that is subgraph isomorphic to g_L. In particular, during the process of sub-iso testing, let g be a non-induced subgraph of g_s, where the vertex set of g is V' and $V' \subseteq V_s$, and let g^I be the subgraph of g_s induced by V'. After verifying that $g^I \sqsubseteq g_L$, we can continuously examine the next vertex of g_s in the same way to proceed. However, even if we have verified $g \sqsubseteq g_L$, we cannot directly turn to the next vertex of g_s to examine if the growing subgraph is a subgraph of g_L, because g_s cannot be finally constructed via growing g by adding the remaining vertices. For example, consider g_2 in Figure 1 and q_2 in Figure 5. During the process of sub-iso testing, if we find a partial subgraph isomorphism $psi = \{(v_1,u_1),(v_2,u_2),(v_3,u_3)\}$, i.e. $g^I \sqsubseteq q_2$, where g^I is the subgraph of g_2 induced by $\{v_1,v_2,v_3\}$, then we could proceed to examine the next vertex v_4 of g_2. However, if we find the sub-iso $\{(v_1,u_1),(v_2,u_6),(v_3,u_3)\}$ from the subgraph with the vertex set being $V' = \{v_1, v_2, v_3\}$ and the edge set being $E' = \{(v_1,v_2), (v_1,v_3)\}$ of g_2 to q_2, we could not extend this sub-iso and proceed to add other matched vertex pairs to obtain a sub-iso from g_2 to q_2.

The above illustration enables a *GPTree* to be used to reduce the number of sub-iso testings. That is, the graph corresponding to the path from the root to any node n in a *GPTree* is an induced subgraph of the graphs that correspond to paths from the root to descendant nodes of n. When we perform sub-iso testings from multiple graphs in a *GPTree*, their common induced subgraphs in one path are examined together. So each embedding (image of a subgraph isomorphism) in q is compared with the common induced subgraph of multiple graphs at the same time. In this way, a number of sub-iso testings are saved.

Algorithm 2 *GPTreeTest* (T , q)

```
Input:    a GPTree T and a query q
Output:   the answer set ANS
1:  ANS ← ∅ ;
2:  SubIsoOnGPTree ( n_root , ∅ );
3:  return ANS ;
Procedure: SubIsoOnGPTree ( n , psi )
4:  if n.GID ≠ ∅ and n.alOut = false then
5:     ANS ← ANS ∪ n.GID ; n.alOut ← true ;
6:  if n.ableChildCnt = 0 then
7:     n.flag ← false ; return;
8:  foreach cn ∈ CHILD(n) do
9:    if cn.flag = false then continue;
10:   compute candidate pair set CPS w.r.t. cn with PruCond_v ;
11:   refine CPS w.r.t. cn with PruCond_ig ;
12:   if CPS = ∅ then continue;
13:   foreach p in CPS do
14:      compute psi' by concatenating p to psi ;
15:      SubIsoOnGPTree( cn , psi');
16:      if cn.flag = false then
17:         n.ableChildCnt - - ;
18:         if n.ableChildCnt = 0 then
19:            n.flag ← false ; return;
20:         break;
```

The algorithm of sub-iso testing from multiple graphs in a *GPTree* to one graph, called *GPTreeTest*, is presented in Algorithm 2. Note that T, q and ANS are global variables. For a node n in *GPTree*, $n.GID$ is the set of graph-IDs attached to n, and $n.flag$, $n.alOut$, $n.ableChildCnt$ are variables, which represent whether n should be examined subsequently, whether $n.GID$ has already outputted, and the number of its children whose *flag* variable is 'true', respectively. Initially, for each n, these three variables are assigned by 'true', 'false', and the number of its children, respectively. In the algorithm, the procedure SubIsoOnGPTree traverses T in a pre-order manner. For a subgraph (or a path in T), it explores states in search space in depth-first manner for seeking sub-iso. Once we find a contained graph we record it and will not consider it again during subsequent searching. In particular, psi is a partial subgraph isomorphism during searching, the last added pair in psi corresponds to the current visited node n in T. In the beginning, if n is associated with some graph-IDs which are not outputted, they should be outputted (Lines 4-5). Then, if all children of n need not be examined, the procedure will directly return (Lines 6-7). In next step, the procedure pre-order traverses the subtree rooted by each child cn of n whose *flag* is 'true' (Lines 8-9). For the current subgraph in T, which corresponds to the path from root to n, we calculate the candidate pair set CPS of next matched vertices surviving in the pruning conditions of $PruCond_v$ and $PruCond_{ig}$ w.r.t. cn (Lines 10-11). If at least one pair in CPS is survived, then for each of the survived pairs in CPS, we construct the next partial mapping psi' by concatenating the pair to psi and then

perform SubIsoOnGPTree from the child recursively (Lines 12-15). At last, ANS is the answer to the query q.

Example 3. *Figure 6 shows the state search space when we performing sub-iso testing from all the graphs in the GPTree in Figure 4 to q_2 in Figure 5 using GPTreeTest. Each state is associated with a vertex pair (last added pair in psi). For the state t associated with (n_6, u_4), $psi(t) = \{(n_1, u_1), (n_2, u_2), (n_3, u_3), (n_6, u_4)\}$. Notice that the first component of each pair in $psi(t)$ is a node in the GPTree, since a subgraph corresponding to any path from the root in a GPTree is isomorphic to any subgraph the path represents. The rectangle in a state t represents that some answers to q_2 are found in t. In particular, in the state with (n_6, u_4), g_2 is found; in the state with (n_5, u_4), g_1 is found. Thus, when the space is finished being searched, the answer set is obtained, which is $\{g_1, g_2\}$.*

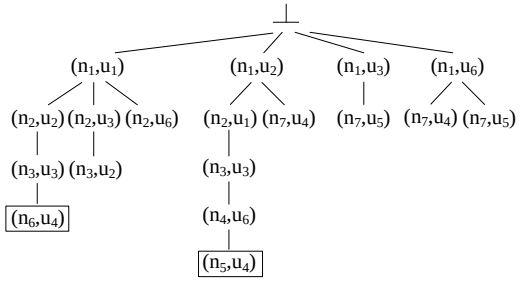

Figure 6: A Search Space

Response Time Analysis of *GPTreeTest*. One of the determinants of the cost of sub-iso testing from a small graph to a large one is the number of vertices of the small graph sg, denoted as $|V_{sg}|$. The reason is that a subgraph isomorphism must involve all vertices in V_{sg}. Therefore, here we analyze the response time of *GPTreeTest* in terms of the number of sub-iso testings with different values of $|V_{sg}|$ explicitly. Let n be the number of graphs in a given *GPTree*. Let n_i be the number of the graphs in the *GPTree* that share the same common prefix, and let $cplen_i$ be the number of codes (or vertices) in the common prefix (or common induced subgraph), where $\sum_{i=1,\ldots,k} n_i = n$. As analyzed in Section 3.2.2, we have that for the n_i graphs with the same $cplen_i$-length common prefix, the testing time saved by the common prefix is $T(cplen_i) \times (n_i - 1)$, where $T(cplen_i)$ is the average time taken by sub-iso testing from the graph corresponding to the common prefix to a query. Therefore, the searching time in *GPTreeTest* is at most $n \times T_{isoEach} - \sum_{i=1,\ldots,k} T(cplen_i) \times (n_i - 1)$, where $T_{isoEach}$ is the average time taken by sub-iso testing from each of the n graphs in the *GPTree* to a query. In addition, by utilizing consecutive storage such as arrays, the overall time taken by initializing the three variables ($flag$, $alOut$, $ableChildCnt$) for all nodes in a *GPTree* is $\Theta(n)$, which is negligibly small compared to the above searching time.

3.4.2 On-line Redundant Features Shedding

Based on the index of *CRGraph*, on-line redundant features shedding for q, which is embedded in the filtering phase, works as follows. Features are examined in an optimal order $\prec^*$. During the process, if a feature f is not contained in q, we add all descendants of the vertex corresponding to f in the *CRGraph* to the set SF, named *shed feature set.* After that, all features in SF need not be tested for sub-iso to q. In next iteration, we examine the next feature in the order $\prec^*$ which is not in SF. This process repeats until all features are either examined or shed. Thus, the eventual shed feature set SF records all features which are avoided from subgraph isomorphism testing.

Complexity Analysis. On-line redundant features shedding involves children enumeration from the *CRGraph*. The time complexity of it is $O(|F| + |E_{CRGraph}|)$, where $E_{CRGraph}$ is the edge set of the *CRGraph*. Thus, it takes $\Theta(|F|^2)$ time in the worst case, but usually close to $\Theta(|F|)$ time in practice. And the space usage is $O(|F|)$ for SF.

3.4.3 The Integrated Query Processing Method

Query processing on the *FGPTree* integrates the on-line redundant features shedding and *GPTreeTest* on the *FGPTree*, whereas query processing on the *GPTree* follows *GPTreeTest* directly with reference to the candidate set obtained after the filtering step. That is, the overall query processing is an integrated method with techniques including on-line redundant features shedding and sub-iso testing from multiple graphs in the *GPTree* to the query. It can be integrated easily, so we do not explicitly elaborate it.

In summary, query processing consists of two steps. Given a query, *FGPTree* is examined by using the *CRGraph* first, i.e. on-line redundant features shedding, and the candidate set is produced. Then, *GPTreeTest* is used against the projection of graphs in *GPTree* onto the candidate set, and the answer set is returned.

3.4.4 Discussions

We briefly discuss the support for external storage as follows. A disk-based strategy is that the trie of the *GPTree* of a graph database is not physically implemented, but only the order, in the *GVCode*, on the vertex set of each graph is recorded. When processing queries, only candidate graphs are retrieved and the trie of *GPTree* of these candidates is built on-the-fly. If the *GPTree* of all candidates cannot be accommodated in memory, then one portion after another of candidates are loaded, and *GPTreeTest* is invoked multiple times to finish the verification step. In this way, the time usage of BuildGPTree apart from $T_{\text{ig-m}}$ mentioned in 3.2.2 is $O(|D| \times |FIG| + \sum_{g \in D} |V_g|)$ operations in memory and the disk-IOs involving storing vertex sequences of all graphs in D; memory usage apart from $S_{\text{ig-m}}$ is $O(\sum_{ig \in FIG} (|sup^I{}_D(ig)| \times |V_{ig}|) + |D| \times |FIG|)$ if *vseq*s are stored in memory. Besides, an alternative strategy is storing the *GPTree* in disk edge by edge in a pre-order manner. Actually, we lay the emphasis on in-memory aspects of the approach, and the exploration of elaborate strategies for support for external storage is considered as future work.

Next, the maintenance of the organization and the indices is discussed in two cases of insertion and deletion. For insertion, when a new graph g is to be inserted into D, Step 1, for *GPTree*, $GVCode(g)$ is generated with the codes in it in random order, and the generated $GVCode(g)$ is inserted into the *GPTree* subsequently; Step 2, *GPTreeTest* on *FGPTree* is performed to update the support sets of all the features contained by g; Step 3, for *CRGraph*, each directed edge is removed if its origin endpoint corresponds to a feature that is not contained in g and its destination endpoint corresponds to a feature that is contained in g. Step 2 and Step 3 are conducted together. For Deletion, when a graph g is to be deleted from D by its graph-ID, the path which only relates to g is directly deleted from the *GPTree*.

4. EXPERIMENTAL EVALUATION

In this section, we present our experimental studies that validate the effectiveness and efficiency of our approach (*GPTree* for short) by comparing it with the state-of-the-art method, cIndex [4].

Two kinds of datasets are used: the real dataset that is used in the evaluation of cIndex and a series of synthetic datasets. All the experiments are performed on an Intel PIV3.0GHz PC with 2GB RAM, running Redhat Linux 8.0. Both cIndex and *GPTree* are implemented in C and compiled by gcc compiler (-O2).

4.1 AIDS Antiviral Screen Dataset

The experiments described in this section use the AIDS antiviral screen dataset ($AIDS$ for short). It contains more than 40,000 chemical compounds and is available publicly. The parameters in cIndex and *GPTree* are set as follows. (1) In cIndex, we randomly draw 10,000 graphs to form a dataset W, then divide W to a query log set $\underline{L}$ (8,000) and a testing query set Q (2000). Contrast subgraphs are mined with the minimum support $\underline{\sigma}_c = 0.05$ (this value relates to the minimum average candidate set size compared to $\underline{\sigma}_c = 0.1$ and 0.01). The smallest number of queries in each leaf for cIndex-TopDown, min_size, is still set 100. (2) In *GPTree*, query logs are not used, and the query set is the same as that in cIndex; the minimum significance threshold $\delta^E{}_{min}$ is 1/0.8 for the exact method and $\delta^A{}_{min} = 1/0.9$ for the approximate one; and the minimum support threshold $\sigma^I{}_T$ for *GPTree* and that for Index construction σ_F are both 0.05, and that for *FGPTree* $\sigma^I{}_{FT}$ is 0.1 In order to build a graph database, we apply frequent subgraph mining on $AIDS$ and retain all subgraphs whose support ranges from 0.5% to 10%, which is denoted by D_{init}. The test dataset consists of 10,000 graphs, denoted by $D_{10,000}$, which are randomly selected from D_{init}. Among all the three particular methods of cIndex, we select cIndex-Basic for comparison of index construction because it yields the least features, and cIndex-TopDown for comparison of query processing due to its greatest efficiency in query processing rather than cIndex-BottomUp and cIndex-Basic.

We first test the index size and index construction time of cIndex-Basic and *GPTree*, and organization construction time of *GPTree*. As mentioned in Section 3.3.3, the index and organization constructions of *GPTree* are integrated, so we compare the overall preprocessing time of it with index construction of cIndex. Table 1 reports the construction time and the number of features on varying σ_F (= $\sigma^I{}_T$) or $\underline{\sigma}_c$.

Table 1. Preprocessing performance for AIDS

min-sup	*GPTree* (E:0.8)		*GPTree* (A:0.9)		cIndex	
	time (s)	**|F|**	**time** (s)	**|F|**	**time** (s)	**|F|**
0.10	26.0	104	25.1	123	465.9	16
0.05	45.8	276	47.3	339	1242.0	15
0.01	611.2	1688	74.9	1981	7095.2	14

The time of constructing *GPTree* and indices in the proposed approach for $AIDS$ is one to two orders of magnitude smaller than that of cIndex. It is because cIndex needs to examine the containment relationship between the initial feature set F_0 and query logs, which is very costly. Although the number of features in the proposed method is more than that in cIndex, but hundreds of features would not occupy too much space. More importantly, we next show that query performance by these features is more efficient than cIndex. The reason is that features are significant and the on-line redundant features shedding can eliminate redundant features in the filtering step, besides the *GPTree*.

To evaluate the query performance of our approach, the 2,000 queries are divided into eight bins: $[0,10)$, $[10,20)$, $[20,30)$, $[30,40)$, $[40,100)$, $[100,200)$, $[200,500)$, $[500,\infty)$, based on the size of the average query answer set, i.e. the number of the graphs in the database that are contained in the query. Figure 7 reports that the average query processing time is about one order of magnitude faster than that using cIndex-TopDown. The main reason is that the compact organization of databases could save more underlying subgraph isomorphism testings.

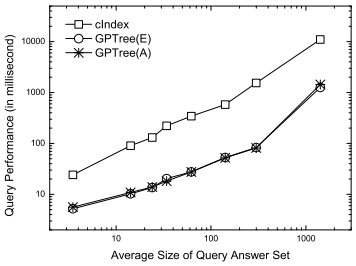

Figure 7: Query Processing Time

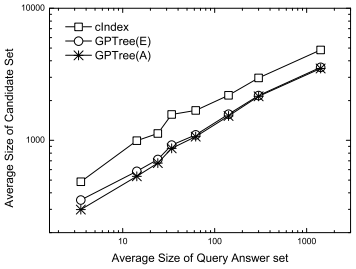

Figure 8: Candidate Answer Set Size

To verify the generated features are significant, we focus the candidate answer set size in Figure 8. X axis shows the average answer set size while Y axis shows the average candidate set size, i.e., $|C_q|$. This figure shows that the candidate set size by the features generated in *GPTree* is several times smaller than that by cIndex-TopDown. Since we use $\delta^A{}_{min}$ (1/0.9)< $\delta^E{}_{min}$ (1/0.8), the feature set generated by the approximate method shows a very close filtering power to that by the exact one.

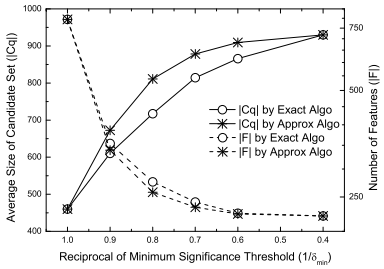

Figure 9: Sensitivity of δ_{min}

Next, we assess the effect of δ_{min} in exact and approximate feature generation methods on the feature set size $|F|$ and the candidate set size $|C_q|$ in Figure 9. In this experiment, the query set $[20,30)$ is processed on the dataset $D_{10,000}$. It shows that the candidate set gradually grows when δ_{min} increases. Simultaneously, the feature set size decreases. In practice, we have to make a trade-off between the performance and the space cost. Moreover, the filtering power of the approximately significant feature set is close to that of exact significant feature set for the same δ_{min}, and they could be approximately equal to each other by decreasing

δ^{A}_{min} slightly. This experiment validates the effectiveness of the approximate method for feature generation, whereas its index construction time is much less than the exact method.

To evaluate scalability, we generate four datasets by randomly selecting graphs from D_{init}, whose size range from 10,000 to 70,000. Query set is Q. Figure 10 and Figure 11 report the query processing time and average candidate set size on databases of various sizes. It shows the high scalability of *GPTree*.

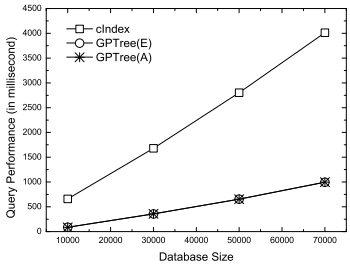

Figure 10: Query Performance on Varying Database

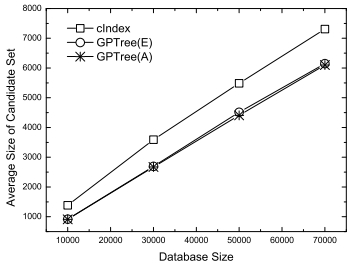

Figure 11: Query Performance on Varying Database

4.2 Synthetic Dataset

In this section, the performance studies on synthetic datasets are conducted. The broadly used graph generator [13] is used to generate datasets, which relates to size parameters: D (number of graphs), T (average size of graphs), L (number of seed small graphs), I (average size of seed small graphs), V (number of vertex labels) and S (allowing overlaps of seed small graphs in generated graphs). It generates graphs as follows. First a set of seed small graphs are generated randomly, whose sizes are determined by a Poisson distribution with mean I. Then seed graphs are randomly selected and inserted into a graph one by one until the graph reaches its expected size T.

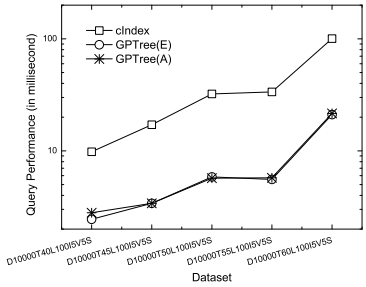

Figure 12: Query Processing Time

For conducting experiments on datasets with different characteristics from the real one, we individually generate database and queries. The database we generated is $D10kT15L100I5V5S$, and query sets are $D10kT40L100I5V5S$, $D10kT45L100I5V5S$, $D10kT50L100I5V5S$, $D10kT55L100I5V5S$ and $D10kT60L100I5V5S$. $D10kT15L100I5V5S$ denotes a set of 10,000 graphs whose average size is 15 and there are 5 vertex labels altogether. We divide each query set into a query log set (8,000) and a testing query set (2000). We set $\sigma_c = 0.10$, min_size =100; in the setting of *GPTree*, the testing query set is the same, and δ^{E}_{min} =1/0.7, δ^{A}_{min} =1/0.8, $\sigma^{I}_{T} = \sigma_{F} = \sigma^{I}_{FT}$ =0.10.

Figure 12 reports that the average query processing time is much less than that using cIndex-TopDown. Figure 13 shows that the candidate set size by the features generated in *GPTree* is still much smaller than that by cIndex-TopDown.

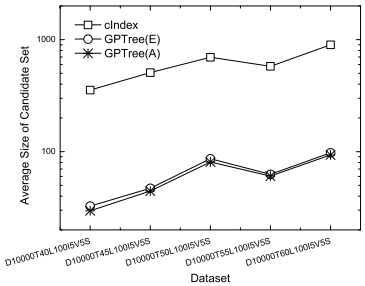

Figure 13: Candidate Answer Set Size

Other synthetic datasets with different parameters were also tested. Similar results were observed in these experiments.

5. RELATED WORK

There have been a number of studies on subgraph query processing. To overcome the difficulty answering arbitrary in structure graph queries, some filtering-and-verification based approaches are proposed to upgrade performance. In these approaches, the first kind [5, 17, 25-27] applies data mining techniques as building blocks for extracting features, and the second kind [11, 12, 18, 21, 28] uses other strategy to construct feature set. However, these methods target subgraph query and are not applicable to the supergraph query. In addition, although closure-tree [11] also arranges graphs into hierarchical indexing structures, which is used to remove false positives and construct a candidate answer set in the filtering step, the proposed method *GPTree* is different from closure-tree and applicable to all the cases of subgraph isomorphism testing from multiple graphs to one graph, including accelerating the computing of candidate set and the verification of each candidate for the studied query processing problem.

As graphs are prevalently used in various domains, a basic problem among these applications is comparing graphs including determining the subgraph relationship between two graphs. This problem may be associated with different names, such as graph matching, (sub)graph isomorphism testing, and so on. It recently obtains a growing attention [3, 6, 8]. For subgraph relationship decision, the Ullmann's algorithm [19] performs a tree search in terms of vertices, and in each substep refines the future vertex pairs on the basis of the current partial matching. A recent algorithm VF2 [7], whose refinement heuristic is faster to compute, achieves in many cases significant improvement over other algorithms. These algorithms all aim at finding sub-iso from one to one, thus they are inefficient for the problem studied in this paper.

For the detection of subgraph isomorphisms from many graphs to one, [15] builds a decision tree in preprocessing phase and results in a quadratic time with respect to the input graph size, but with exponential space requirement and preprocessing time,

which leads to its inapplicability to large-size databases. An inspiring decomposition-based method [16] results in a time sublinear w.r.t. the number of graphs in a database. However, owing to the underlying decomposition strategy, the output of the method are all subgraph isomorphisms from each graph in answer set to the query graph, which is unnecessary and time consuming for the supergraph query. In the XML context, [10] constructs a single deterministic pushdown automata to generalize and improve tree pattern matching technique (not for arbitrary in structure graphs) for the specific task of evaluating XPath queries. [1] lays emphasis on finding XML schema embeddings by which an instance-level mapping can be automatically derived and it guarantees information preservation w.r.t. an XML query language. It does not focus on finding schema embeddings from large amounts of source DTD schemas to a single target DTD in a scalable manner. To the best of our knowledge, cIndex [4] is the only method employing the filtering-and-verification methodology to process supergraph queries so far. However, there is no algorithm that exploits the efficient methodology and considers organizing graphs in databases to upgrade the supergraph query processing performance, which is the emphasis of this study.

An introduction on graph mining is given in [20]. There has been many methods proposed [2, 13, 22-24], which can efficiently obtain (closed) frequent (induced) subgraphs from a database. To decrease the number of frequent subgraphs in a parameterized way, graph patterns summarization was proposed in [14]. They play an important role in preprocessing phase in the paper.

6. CONCLUSIONS

In this paper, in order to answer the supergraph query, a novel compact organization of a graph database, *GPTree*, was proposed. Adopting the filtering-and-verification methodology, we introduced two methods for feature generation. Besides the exact significant feature set generation method, an approximate method for generating significant feature set was proposed. The approximate method could comparatively fast generate a feature set. Features are arranged in an optimal order, and by using the proposed online redundant features shedding method, the number of subgraph isomorphism testings from features to query graphs is reduced. Based on *GPTree*, a new algorithm from multiple graphs to one, *GPTreeTest*, was proposed. Benefiting from *GPTree* and the algorithm of *GPTreeTest*, much less number of subgraph isomorphism testings need be performed in both the filtering and the verification steps. Based on all the above techniques, the proposed supergraph query processing approach outperforms the existing counterpart method by one to two orders of magnitude.

7. ACKNOWLEDGMENTS

This work is supported in part by the 973 Program of China (Grant No. 2006CB303000) and NSFC (Grant No. 60533110 and Grant No. 60773063).

8. REFERENCES

[1] P. Bohannon, W. Fan, M. Flaster, and P. P. S. Narayan. Information Preserving XML Schema Embedding. in *VLDB*, 2005.

[2] C. Borgelt and M. R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. in *ICDM*, 2002.

[3] H. Bunke. Graph Matching: Theoretical Foundations, Algorithms, and Applications. in *Vision Interface*, pages 82-88, 2000.

[4] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards Graph Containment Search and Indexing. in *VLDB*, 2007.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-Index: Towards Verification-Free Query Processing on Graph Databases. in *SIGMOD*, 2007.

[6] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years Of Graph Matching In Pattern Recognition. *IJPRAI*, *18*, 3 (2004), 265-298.

[7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, *26*, 10 (2004), 1367-1372.

[8] S. Fortin, *The Graph Isomorphism Problem*, in *Univ. of Alberta*. 1996.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[10] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. in *SIGMOD*, 2003.

[11] H. He and A. K. Singh. Closure-Tree: An Index Structure for Graph Queries. in *ICDE*, 2006.

[12] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. GString: A Novel Approach for Efficient Search in Graph Databases. in *ICDE*, 2007.

[13]M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. in *ICDM*, 2001.

[14] Y. Liu, J. Li, and H. Gao. Summarizing Graph Patterns. in *ICDE*, 2008.

[15] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, *32*, 12 (1999), 1979-1998.

[16] B. T. Messmer and H. Bunke. Efficient Subgraph Isomorphism Detection: A Decomposition Approach. *IEEE Trans. Knowl. Data. Eng.*, *12*, 2 (2000), 307-323.

[17] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. in *VLDB*, 2008.

[18] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. in *PODS*, 2002.

[19] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, *23*, 1 (1976), 31-42.

[20] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, *5*, 1 (2003), 59-68.

[21] D. W. Williams, J. Huan, and W. Wang. Graph Database Indexing Using Structured Graph Decomposition. in *ICDE*, 2007.

[22] M. Worlein, *Extension and parallelization of a graph-mining-algorithm*, in *Friedrich-Alexander-Universität*. 2006.

[23] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. in *ICDM*, 2002.

[24] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. in *KDD*, 2003.

[25] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. in *SIGMOD*, 2004.

[26] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. in *ICDE*, 2007.

[27] P. Zhao, J. X. Yu, and P. S. Yu. Graph Indexing: Tree + Delta >= Graph. in *VLDB*, 2007.

[28] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. in *EDBT*, 2008.