J Comb Optim (2011) 21: 159–191
DOI 10.1007/s10878-009-9221-1

Efficient algorithms for supergraph query processing on graph databases

Shuo Zhang · Xiaofeng Gao · Weili Wu · Jianzhong Li · Hong Gao

Published online: 17 March 2009

Abstract We study the problem of processing supergraph queries on graph databases. A graph database D is a large set of graphs. A *supergraph query* q on D is to retrieve all the graphs in D such that q is a supergraph of them. The large number of graphs in databases and the NP-completeness of subgraph isomorphism testing make it challenging to efficiently processing supergraph queries. In this paper, a new approach to processing supergraph queries is proposed. Specifically, a method for compactly organizing graph databases is first presented. 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 the stored graphs to queries during query processing. Then, an exact algorithm and an approximate algorithm for generating the significant feature set with optimal order are proposed, followed by the algorithms for indices construction 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 for testing subgraph isomorphisms from multiple graphs to one graph is presented. Finally, based on all the above 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.

Keywords Graph database · Supergraph query · Query processing · Graph indexing

S. Zhang · J. Li (✉) · H. Gao
Harbin Institute of Technology, Harbin, China
e-mail: lijzh@hit.edu.cn

X. Gao · W. Wu
University of Texas at Dallas, Dallas, USA

1 Introduction

Recently, large amount of data modeled by graphs, such as molecular structures of compounds in chemistry (Willett et al. 1998; Agrafiotis et al. 2007), organizations of entities in images (Petrakis and Faloutsos 1997; Burge and Kropatsch 1999), topologies of sensor networks (Li et al. 2003), objects in technical drawings in mechanical engineering field (Cordella et al. 2000) and social networks (Cai et al. 2005), 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 that 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 subgraph queries. Subgraph query processing has attracted much research attention in last several years, and many algorithms have been proposed (Shasha et al. 2002; Yan et al. 2005; He and Singh 2006; Zhang et al. 2007; Williams et al. 2007; Jiang et al. 2007; Cheng et al. 2007; Zhao et al. 2007; Zou et al. 2008; Shang et al. 2008). 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 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 (Shasha et al. 2002) 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 the best of our knowledge, there is only one algorithm, named cIndex (Chen et al. 2007), to date in the literature for processing the supergraph query by adopting this methodology. cIndex constructs an index on the *features*, which 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 based on the feature index. 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 query logs while they are extracted from graph databases.

However, the effectiveness of the feature index constructed by cIndex depends on the historical queries in query logs. 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.

To efficiently process supergraph queries, this paper investigates supergraph query processing from a new perspective. The proposed new method 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 method, graph databases are stored compactly beyond the flat manner, i.e. the graphs in graph databases are arranged one by one, to improve the efficiency of query processing. To accelerate the construction of feature indices, a fast algorithm for extracting features from graph databases is proposed without taking query logs into consideration. In order to further improve the efficiency of query processing, 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 for subgraph isomorphism testing from multiple graphs to one graph is proposed.

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

The main contributions of this paper are as follows.

(1) A method for compactly storing graph databases is proposed. Common subgraphs of the graphs in a database are stored only once in the compact organization of the database, named *GPTree*, in order to reduce the overall cost of subgraph isomorphism testings from the stored graphs to queries during query processing. To construct GPTree from graph databases, after the problem of optimal induced subgraph selecting is proved to be NP-hard, an approximation algorithm with ratio bound 2 is proposed.
(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. The index structures of *CRGraph* and *FGPForest* are devised for accommodating features and the order on the feature set so as to accelerate query processing.
(3) To improve the performance of the crucial part of supergraph query processing greatly, a new algorithm for 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 reviews some preliminary concepts and presents the problem definition. Section 3 presents the proposed query processing method including the compact method for storing graph databases, the feature extracting and ordering algorithms, the index structures and their construction algorithms, the subgraph isomorphism testing algorithm from multiple graphs to one graph, and the integrated query processing method. Experimental evaluation is given in Sect. 4. The related work is surveyed in Sect. 5. Section 6 concludes the paper.

2 Subgraph isomorphism and supergraph query

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

Definition 2 (Subgraph isomorphism) 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 $f : V \to V'$ 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 $l(u, v)$[1] $= l'(f(u), f(v))$.

Definition 3 (Induced subgraph isomorphism) 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 $f^I : V \to V'$ 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 $\text{size}(g) + 1 = \text{size}(\tilde{g})$, $\tilde{g}$ 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. Hereinafter, we use the term *sub-iso* to express *subgraph isomorphism*.

Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ 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, $0 \le \sigma \le 1$, a graph g is called frequent (or induced frequent) in D if $\sigma_D(g) \ge \sigma$ (or $\sigma^I_D(g) \ge \sigma$).

The supergraph query processing problem can be defined as follows.

Input: a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a query graph q.

Output: $Answer(q) = \{g_i \mid g_i \sqsubseteq q, g_i \in D\}$.

[1] $l((u, v))$ is denoted by $l(u, v)$ for ease of presentation, similarly hereinafter.

3 Supergraph query processing

3.1 Overview of query processing method

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

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

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 features. Finally, two feature indices on the given graph database, named *CRGraph* and *FGPForest*, 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 indices of CRGraph and FGPForest. 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 of the graph database and the *GPTreeTest* algorithm, i.e. the algorithm for 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 $T_{query} = T_{filtering} + T_{verification}$, where $T_{filtering}$ is the time cost of computing a candidate set by testing sub-iso from features to q, and $T_{verification}$ is the time cost of verifying all candidates by testing sub-iso from candidate graphs to q. The preprocessing of the given graph database, i.e. both Part 1 and Part 2, aim at reducing the query processing cost T_{query}, i.e. both $T_{filtering}$ and $T_{verification}$.

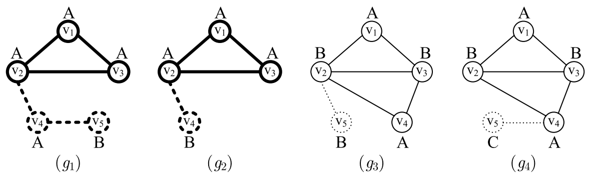
The details of Part 1, Part 2, and Part 3 are presented in Sects. 3.2, 3.3 and 3.4, respectively.

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 consisting of the 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 intuitive idea of the compact organization of the graph database in Fig. 1. In Fig. 2 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 2 only shows the intuitive idea of the compact organization of a graph database. Actually, we propose a data structure, named *GPTree*, to implement the

Fig. 1 Running example: a graph database



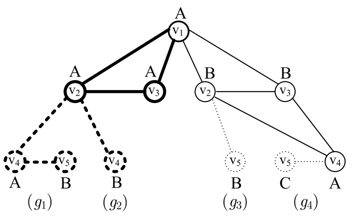


Fig. 2 Intuitive idea of a compact organization

idea. In the following, the structure of GPTree and the algorithm for constructing GPTree are presented.

3.2.1 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 (GVCode) Given a graph $g = (V, E, \Sigma, l)$ and a total order $\prec_V$ on V, a GVCode of g, denoted by $GVCode(g)$, is a sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_{|V|} \rangle$ that is defined as follows.
For $1 \le j \le |V|$, $v_j \in V$,

(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) for $\forall \tau = (i, l(v_i), l(v_i, v_j))$ and $\forall \tau' = (i', l(v_{i'}), l(v_{i'}, v_j))$ in α_j, τ is before τ' if and only if $i < i'$.

Example 1 (GVCode) Figure 3 shows the GVCodes of the four graphs in Fig. 1. Take 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 the edges (v_1, v_3) and (v_2, v_3), and show that 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 before $(2, A, -)$ since the condition (3) in Definition 4 holds.

Please note that a graph may have multiple GVCodes due to the variety of total orders on the vertex set of the 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 GVCodes of multiple graphs corresponds to a common induced subgraph of these graphs.

GPTree A *GPTree* of a graph database is a trie constructed by the GVCodes of all the graphs in the database, with taking a code in a GVCode as a basic unit of the sequence. For example, Fig. 4 shows a sample GPTree constructed by the graph database in Fig. 1, where for $1 \le i \le 11$ n_i is defined in Fig. 4b. In the GPTree, the

$\alpha_1 = \langle(1,A)\rangle$
$\alpha_2 = \langle(2,A), (1, A, -)\rangle$
$\alpha_3 = \langle(3,A), (1, A, -), (2, A, -)\rangle$
$\alpha_4 = \langle(4,A), (2, A, -)\rangle$
$\alpha_5 = \langle(5,B), (4, A, -)\rangle$

(a) *GVCode* of g_1

$\beta_1 = \langle(1,A)\rangle$
$\beta_2 = \langle(2,A), (1, A, -)\rangle$
$\beta_3 = \langle(3,A), (1, A, -), (2, A, -)\rangle$
$\beta_4 = \langle(4,B), (2, A, -)\rangle$

(b) *GVCode* of g_2

$\gamma_1 = \langle(1,A)\rangle$
$\gamma_2 = \langle(2,B), (1, A, -)\rangle$
$\gamma_3 = \langle(3,B), (1, A, -), (2, B, -)\rangle$
$\gamma_4 = \langle(4,A), (2, B, -), (3, B, -)\rangle$
$\gamma_5 = \langle(5,B), (2, B, -)\rangle$

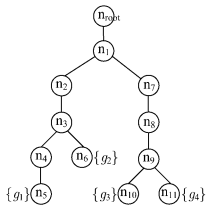
(c) *GVCode* of g_3

$\lambda_1 = \langle(1,A)\rangle$
$\lambda_2 = \langle(2,B), (1, A, -)\rangle$
$\lambda_3 = \langle(3,B), (1, A, -), (2, B, -)\rangle$
$\lambda_4 = \langle(4,A), (2, B, -), (3, B, -)\rangle$
$\lambda_5 = \langle(5,C), (4, A, -)\rangle$

(d) *GVCode* of g_4

Fig. 3 Four GVCodes

Fig. 4 A GPTree



(a) *GPTree*

$n_1 = \langle(1,A)\rangle$	$n_7 = \langle(2,B), (1, A, -)\rangle$
$n_2 = \langle(2,A), (1, A, -)\rangle$	$n_8 = \langle(3,B), (1, A, -), (2, B, -)\rangle$
$n_3 = \langle(3,A), (1, A, -), (2, A, -)\rangle$	$n_9 = \langle(4,A), (2, B, -), (3, B, -)\rangle$
$n_4 = \langle(4,A), (2, A, -)\rangle$	$n_{10} = \langle(5,B), (2, B, -)\rangle$
$n_5 = \langle(5,B), (4, A, -)\rangle$	$n_{11} = \langle(5,C), (4, A, -)\rangle$
$n_6 = \langle(4,B), (2, A, -)\rangle$	

(b)

path $\langle n_1, n_2, n_3, n_4, n_5\rangle$[2] corresponds to g_1 (or the GVCode of g_1 in Fig. 3), which is indicated by the set of graph-IDs attached to the last vertex, n_5, of the path. The path $\langle n_1, n_2, n_3, n_6\rangle$ corresponds to g_2 (or the GVCode of g_2), indicated by the set of graph-IDs attached to n_6. The path $\langle n_1, n_2, n_3\rangle$ represents a common prefix of the GVCodes 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 represent multiple isomorphic graphs.

3.2.2 Algorithm for constructing GPTree

To construct a GPTree of a graph database, each graph in the database is encoded into a GVCode. Because multiple GVCodes may be generated from a graph as mentioned in Sect. 3.2.1, there may be multiple GPTrees that can be constructed from a database. To identify a good GPTree for efficiently answering supergraph queries, the cost of sub-iso testing from multiple graphs organized 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 Fig. 5 issued on the graph database in Fig. 1. To process the query q_1 using naive method, sub-iso testings are performed one by one, i.e. test sub-iso from g_1 to q_1, g_2 to $q_1, \ldots,$ and g_4 to q_1 one by one. It need perform 4 sub-iso testings. To process q_1 using the GPTree in Fig. 4, the sub-iso testing from g_1 to q_1 is performed first. It will be found that the triangle in bold solid lines in Fig. 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

[2] I.e. $\langle n_{root}, n_1, n_2, n_3, n_4, n_5\rangle$. A path starting from the root of a GPTree is presented with the root removed hereinafter, for ease of presentation.

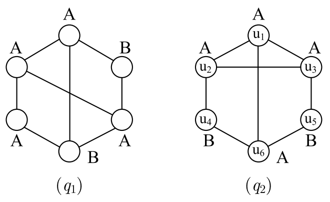


Fig. 5 Running example: two queries

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 Fig. 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 (or avoided) altogether compared to the naive method.

Then let us consider the supergraph query with query graph q_2 in Fig. 5, which is issued on the graph database in Fig. 1. Similar to the results of q_1, the naive method need perform 4 sub-iso testings to process q_2. When processing q_2 using the GPTree in Fig. 4, a subgraph isomorphism from the triangle in bold solid lines in Fig. 2 to q_2 is found. Then, the subgraph isomorphisms 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 as the naive method does. Similarly, the sub-iso testing from the five-edge subgraph in solid lines in Fig. 2 to q_2 is just performed once. Thus, 2 sub-iso testings are saved compared to the naive method.

In general, let the GVCodes 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

$$c(\text{len}(cp)) \times (|GS| - 1), \tag{1}$$

where $\text{len}(cp)$ is the number of the codes in cp, $c : \mathbb{N} \to \mathbb{R}^+$ is a monotonic increasing function and the value of $c(n)$ is the average cost taken by one sub-iso testing from a subgraph with n 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, $D = \{g_1, g_2, \ldots, g_n\}$ 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 GVCodes of the distinct graphs $g_{11}, g_{12}, \ldots, g_{1n_1}$ in D share the common prefix cp_1 in the GPTree, the GVCodes of the distinct graphs $g_{21}, g_{22}, \ldots, g_{2n_2}$ in D share the common prefix cp_2, ..., and the GVCodes of the distinct graphs $g_{k1}, g_{k2}, \ldots, g_{kn_k}$ in D share the common prefix cp_k in the GPTree. Here $\sum_{j=1}^{k} n_j = |D|$. Based on (1), for $1 \le j \le k$ the cost saving by the common prefix cp_j is $c(\text{len}(cp_j)) \times (n_j - 1)$. Therefore, the overall cost saving, denoted by C_{sav}, of sub-iso testings during the processing of Q using the GPTree, by the common prefixes cp_j for $1 \le j \le k$, is no less than $\sum_{j=1}^{k} c(\text{len}(cp_j)) \times (n_j - 1)$.

Based on the above analysis, because a GPTree of D is constructed by the GVCodes of all the graphs in D and each graph in D may have multiple GVCodes, 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, in order to efficiently answer supergraph queries.

A prefix of a GVCode of a graph corresponds to an induced subgraph of the graph as mentioned in Sect. 3.2.1. Conversely, given an induced subgraph of a graph, it can be used to generate a prefix of a GVCode of the graph. Thus, to get a GVCode of a graph $g = (V, E, \Sigma, l)$, we can first select an induced subgraph $ig = (V_{ig}, E_{ig}, \Sigma_{ig}, l_{ig})$ 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 $V - V_{ig}$.

Thus, to solve the problem of constructing GPTree for efficiently answering supergraph queries, the following steps are performed.

Step 1. Select the optimal induced subgraph for each graph in D.
Step 2. Generate the GVCodes of all the graphs in D using the induced subgraphs obtained in Step 1.
Step 3. Construct the GPTree of D by the GVCodes 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 $\sum_{j=1}^{k} c(|V_{cig_j}|) \times (n_j - 1)$ is maximized, where for $1 \leq j \leq k$, cig_j is the common induced subgraph selected for the distinct graphs $g_{j1}, g_{j2}, \ldots, g_{jn_j}$ in D, and $\sum_{j=1}^{k} n_j = |D|$. This problem is called *induced subgraph selecting problem* in the paper, defined as follows.

Input: a graph database $D = \{g_1, g_2, \ldots, g_n\}$.

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

Objective: maximize

$$\sum_{i=1}^{|D|} c(|V_{ig_i}|) - \sum_{ig \in \bigcup_{1 \leq i \leq |D|} \{ig_i\}} c(|V_{ig}|).$$

Please note that $\sum_{j=1}^{k} c(|V_{cig_j}|) \times (n_j - 1) =$

$$\sum_{i=1}^{|D|} c(|V_{ig_i}|) - \sum_{ig \in \bigcup_{1 \leq i \leq |D|} \{ig_i\}} c(|V_{ig}|).$$

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

Let S be a finite set, $f : S \to \mathbb{R}^+$ be a positive function, and $\mathcal{C}$ be a collection of subsets of S. We create a bijection between S and IG, a bijection between $\mathcal{C}$ and D, and a bijection between $f(e)$ and the cost function $c(|V_{ig}|)$ in (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 $S_i = \{e'_1, e'_2, \ldots, e'_l\}$ in $\mathcal{C}$ corresponds to a graph g_i in D, then g_i is an induced supergraph of $ig'_1, ig'_2, \ldots, ig'_l$ (all in IG) but none of other graphs in IG, where $ig'_1, ig'_2, \ldots, ig'_l$ correspond to $e'_1, e'_2, \ldots, e'_l$, respectively.

$|\mathcal{C}| = |D|$. Under these bijections, finding a sequence $\langle ig_1, ig_2, \dots, ig_{|D|}\rangle$ is to find a sequence $\langle e_1, e_2, \dots, e_{|\mathcal{C}|}\rangle$, where for $1 \le i \le |\mathcal{C}|$, e_i is an element of the subset S_i in $\mathcal{C}$, e_i corresponds to ig_i, and

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

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 $\mathcal{C}$ of subsets of S.

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

Objective: maximize

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

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

Input: a collection $\mathcal{C}$ of subsets of a finite set S and a positive integer $K \le |S|$.

Output: if there is a sequence $\langle e_1, e_2, \dots, e_{|\mathcal{C}|}\rangle$ such that $|\bigcup_{i=1}^{|\mathcal{C}|}\{e_i\}| \le K$ then output 'true', otherwise output 'false', where for $1 \le i \le |\mathcal{C}|$, e_i is an element of the subset S_i in $\mathcal{C}$.

Lemma 1 *The uniquely hitting set problem is NP-complete.*

Proof It is easy to prove that the uniquely hitting set problem is NP. Because a non-deterministic algorithm need only guess a sequence of elements $\langle e_1, e_2, \dots, e_{|\mathcal{C}|}\rangle$, where for $1 \le i \le |\mathcal{C}|$, e_i is an element of the subset S_i in $\mathcal{C}$, and check in polynomial time whether $|\bigcup_{i=1}^{|\mathcal{C}|}\{e_i\}| \le K$ or not.

The hitting set problem is NP-complete (Garey and Johnson 1979), which can be reduced to the uniquely hitting set problem in polynomial time as follows. Given any instance of the hitting set problem, i.e. a collection $\mathcal{C} = \{S_1, S_2, \dots, S_{|\mathcal{C}|}\}$ of subsets of a ground set S and a positive integer $K \le |S|$, we define an instance of the uniquely hitting set problem by the same instance unchangeably. Apparently, the construction of the new (defined) instance takes polynomial time. Then we claim that there exists a satisfied sequence of elements for the uniquely hitting set problem if and only if there exists a satisfied subset of S for the hitting set problem. The reason is explained as follows. For necessity, if there is a sequence of elements $\langle e_1, e_2, \dots, e_{|\mathcal{C}|}\rangle$ such that $|\bigcup_{i=1}^{|\mathcal{C}|}\{e_i\}| \le K$, where for $1 \le i \le |\mathcal{C}|$, e_i is an element of the subset S_i in $\mathcal{C}$, then we have that for each subset in $\mathcal{C}$, the set $\bigcup_{i=1}^{|\mathcal{C}|}\{e_i\}$ contains at least one element of the subset. For sufficiency, suppose there exists a subset $S' \subseteq S$ such that $|S'| \le K$ and for each subset in $\mathcal{C}$, S' contains at least one element of the subset. For each subset in $\mathcal{C}$, we arbitrarily select one element from the intersection of S' and the subset; and then we construct a sequence of elements $\langle e'_1, e'_2, \ldots, e'_{|\mathcal{C}|}\rangle$. Please note that the intersection of S' and any subset in $\mathcal{C}$ cannot be the empty set. It follows that $|\bigcup_{i=1}^{|\mathcal{C}|}\{e'_i\}| \leq |S'|$, and thus, $|\bigcup_{i=1}^{|\mathcal{C}|}\{e'_i\}| \leq K$. □

Theorem 1 *The induced subgraph selecting problem is NP-hard.*

Proof 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 as follows to solve the problem. In the approximation algorithm, the element selected from a subset S_i in $\mathcal{C}$ is called the uniquely hitting element of S_i, denoted by $uhe(S_i)$. The data structure $\mathcal{A}$ is used to store all 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 $\mathcal{Z} \subseteq \mathcal{C}$. The proposed algorithm is as follows.

```
Input: a finite set $S$, a positive cost function $f : S \to \mathbb{R}^+$ and a collection $\mathcal{C}$ of
       subsets of $S$.
Output: the sequence $\langle uhe(S_1), uhe(S_2), \ldots, uhe(S_{|\mathcal{C}|})\rangle$, where $S_i \in \mathcal{C}$ for
        $1 \leq i \leq |\mathcal{C}|$.
1  $\mathcal{A} \leftarrow \emptyset$ ;
2  while $\exists e \in S$ such that $|\mathcal{H}(e, \mathcal{C} - \mathcal{A})| > 1$ do
3      select $e' \in S$ with largest $f(e')$ such that $|\mathcal{H}(e', \mathcal{C} - \mathcal{A})| > 1$ ;
4      for each $S_i \in \mathcal{H}(e', \mathcal{C} - \mathcal{A})$ do
5          $uhe(S_i) \leftarrow e'$ ;
6      $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{H}(e', \mathcal{C} - \mathcal{A})$ ;
7  while $\exists e \in S$ such that $|\mathcal{H}(e, \mathcal{C} - \mathcal{A})| = 1$ do
8      select $e'' \in S$ with largest $f(e'')$ such that $|\mathcal{H}(e'', \mathcal{C} - \mathcal{A})| = 1$ ;
9      for each $S_i \in \mathcal{H}(e'', \mathcal{C} - \mathcal{A})$ do
10         $uhe(S_i) \leftarrow e''$ ;
11     $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{H}(e'', \mathcal{C} - \mathcal{A})$ ;
```

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

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

During the course of the algorithm, if the first element being selected is produced in Line 8 (not in Line 3), then for $1 \le i \le |\mathcal{C}|$, $\mathcal{H}(e_i, \mathcal{C}) = 1$. Thus, the algorithm outputs a solution with the objective 0, and the upper bound for the sum of the contribution values from all the subsets in $\mathcal{C}$ for all solutions (or for the optimal solution) is 0. So in this case, the solution outputted by the algorithm is an optimal solution.

Otherwise, first, we consider all the iterations within Lines 2–6 in the above algorithm. Let $e'_1, e'_2, \ldots, e'_{p_1}$ be the elements selected in turn in these iterations, where p_1 is the number of the elements selected here altogether.

During the course of the algorithm, let the first element being selected in Line 3 be e'_1. On the one hand, if all the subsets containing e'_1 (or in $\mathcal{H}(e'_1, \mathcal{C})$) do not contain an element with larger cost function value than $f(e'_1)$, then the contribution value from each of these subsets in $\mathcal{H}(e'_1, \mathcal{C})$ is no more than $f(e'_1)$, and consequently for all solutions (or for the optimal solution) the sum of the contribution values from all the subsets in $\mathcal{H}(e'_1, \mathcal{C})$ must be no more than $f(e'_1) \times \mathcal{H}(e'_1, \mathcal{C})$. On the other hand, if some subsets in $\mathcal{H}(e'_1, \mathcal{C})$ contain elements, denoted by $e'^1_1, e'^2_1, \ldots, e'^m_1$, with larger cost function values, i.e. $f(e'^l_1) > f(e'_1)$ for $1 \le l \le m$, then for $1 \le l \le m$ the number of all the subsets in $\mathcal{H}(e'^l_1, \mathcal{C})$ is 1 due to the greedy selection strategy in Lines 2–6. If a solution, denoted by $\mathfrak{sol}'$, contains some elements of them, denoted by $e'^{r_1}_1, e'^{r_2}_1, \ldots, e'^{r_k}_1$ (for $1 \le l \le k$, $1 \le r_l \le m$), in the positions corresponding to their related subsets, respectively, then for $1 \le l \le k$, the subset in $\mathcal{H}(e'^{r_l}_1, \mathcal{C})$ contributes $f(e'^{r_l}_1) \times (|\mathcal{H}(e'^{r_l}_1, \mathcal{C})| - 1) = 0$ to the objective. Then we can construct a new solution $\mathfrak{sol}$ from $\mathfrak{sol}'$ by replacing each of $e'^{r_1}_1, e'^{r_2}_1, \ldots, e'^{r_k}_1$ with e'_1, respectively. The new constructed solution $\mathfrak{sol}$ is no worse (the objective is no less) than the old solution $\mathfrak{sol}'$ since the contribution values from $e'^{r_1}_1, e'^{r_2}_1, \ldots, e'^{r_k}_1$ (as well as those from other subsets) do not decrease during the replacing. Therefore, we have that for the optimal solution the sum of contribution values from all the subsets in $\mathcal{H}(e'_1, \mathcal{C})$ must be no more than $f(e'_1) \times \mathcal{H}(e'_1, \mathcal{C})$.

During the iterating within Lines 2–6, let the next element to be selected be e'_x. Similar to the proof given in the above paragraph, on the one hand, if all the subsets in $\mathcal{H}(e'_x, \mathcal{C} - \mathcal{A})$ do not contain an element with larger cost function value than $f(e'_x)$, then for the optimal solution the sum of contribution values from all the subsets in $\mathcal{H}(e'_x, \mathcal{C} - \mathcal{A})$ is no more than $f(e'_x) \times \mathcal{H}(e'_x, \mathcal{C} - \mathcal{A})$. On the other hand, if there exist some subsets in $\mathcal{H}(e'_x, \mathcal{C} - \mathcal{A})$ such that they contain elements, denoted by $e'^1_x, e'^2_x, \ldots, e'^\mu_x$ with larger cost function values than $f(e'_x)$, then for $1 \le l \le \mu$ the number of all the subsets in $\mathcal{H}(e'^l_x, \mathcal{C} - \mathcal{A})$ is 1. Without loss of generality, suppose e'^l_x is the element with the largest cost function value in the subset in $\mathcal{H}(e'^l_x, \mathcal{C} - \mathcal{A})$. For a solution $\mathfrak{sol}'$ that contains $e'^{s_1}_x, e'^{s_2}_x, \ldots, e'^{s_\kappa}_x$ (for $1 \le l \le \kappa$, $1 \le s_l \le \mu$) in the positions corresponding to their related subsets in $\mathcal{C} - \mathcal{A}$, respectively, we claim that the new solution $\mathfrak{sol}$ constructed from $\mathfrak{sol}'$ by replacing each of $e'^{s_1}_x, e'^{s_2}_x, \ldots, e'^{s_\kappa}_x$ in these positions with e'_x, respectively, is no worse than the old solution $\mathfrak{sol}'$ before the replacing, which is proved as follows. For $e'^{s_l}_x$ ($\forall l, 1 \le l \le \kappa$), each subset in $\mathcal{H}(e'^{s_l}_x, \mathcal{A})$ must contain an element with larger cost function value than $f(e'^{s_l}_x)$ that has already been selected by the greedy selection strategy, because if not then $e'^{s_l}_x$ will be selected before e'_x by the greedy selection strategy and a contradiction is obtained. Consequently, Subcase 1, for a solution that contains $e'^{s_l}_x$ in the positions corresponding to some subsets in $\mathcal{H}(e'^{s_l}_x, \mathcal{A})$, the sum of the contribution values from all the subsets in

$\mathcal{H}(e_x'^{s_l}, \mathcal{A})$ must be no more than the sum of the cost function values of the elements selected by the greedy selection strategy for all the subsets in $\mathcal{H}(e_x'^{s_l}, \mathcal{A})$; and the contribution value from the subset in $\mathcal{H}(e_x'^{s_l}, \mathcal{C} - \mathcal{A})$ is 0. Subcase 2, for a solution that does not contain $e_x'^{s_l}$ in the position corresponding to any subset in $\mathcal{H}(e_x'^{s_l}, \mathcal{A})$, the contribution value from the subset in $\mathcal{H}(e_x'^{s_l}, \mathcal{C} - \mathcal{A})$ is 0 apparently. Therefore, the new solution $\mathfrak{sol}$ constructed from $\mathfrak{sol}'$ by replacing each of $e_x'^{s_1}, e_x'^{s_2}, \ldots, e_x'^{s_\kappa}$ in the positions corresponding to the subsets in $\mathcal{C} - \mathcal{A}$ with e_x', respectively, is no worse than $\mathfrak{sol}'$. In summary, it follows that for all solutions (or for the optimal solution) the sum of the contribution values from all the subsets in $\mathcal{H}(e_x', \mathcal{C} - \mathcal{A})$ is no more than $f(e_x') \times \mathcal{H}(e_x', \mathcal{C} - \mathcal{A})$ on the basis of the bound for the sum of the contribution values from all the subsets that contain the previously selected elements (or in $\mathcal{A}$).

Therefore, after performing all the iterations within Lines 2–6, we obtain the partial solution produced by the greedy strategy, whose objective is equal to $\sum_{j=1}^{p_1} f(e_j') \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}| - 1)$, and an upper bound for the sum of the contribution values from all the subsets in $\mathcal{A}'_{p_1}$ for all solutions (or for the optimal solution), which is $\sum_{j=1}^{p_1} f(e_j') \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}|)$, where for $k \geq 1$, $\mathcal{A}'_k = \bigcup_{j=1}^{k} \mathcal{H}(e_j', \mathcal{C})$ and $\mathcal{A}'_0 = \emptyset$.

Second, considering all the iterations within Lines 7–11 in the above algorithm, let $e_1'', e_2'', \ldots, e_{p_2}''$ be the elements selected in turn in these iterations (there are p_2 elements altogether). During the iterating, let the next element to be selected be e_x''. We have that the number of all the subsets in $\mathcal{H}(e_x'', \mathcal{C} - \mathcal{A}'_{p_1})$ is 1, since if not then e_x'' must be selected in the iterations within Lines 2–6. Also we have that e_x'' is the element with the largest cost function value in the subset in $\mathcal{H}(e_x'', \mathcal{C} - \mathcal{A}'_{p_1})$. Then, similar to the proof for the iterations within Lines 2–6, each subset in $\mathcal{H}(e_x'', \mathcal{A}'_{p_1})$ must contain an element with larger cost function value than $f(e_x'')$ that has already been selected by the greedy selection strategy within Lines 2–6. Consequently, Subcase 1, for a solution that contains e_x'' in the positions corresponding to some subsets in $\mathcal{H}(e_x'', \mathcal{A}'_{p_1})$, the sum of the contribution values from all the subsets in $\mathcal{H}(e_x'', \mathcal{A}'_{p_1})$ must be no more than the sum of the cost function values of the elements selected by the greedy selection strategy within Lines 2–6 for all the subsets in $\mathcal{H}(e_x'', \mathcal{A}'_{p_1})$; and the contribution value from the subset in $\mathcal{H}(e_x'', \mathcal{C} - \mathcal{A}'_{p_1})$ is 0. Subcase 2, for a solution that does not contain e_x'' in the position corresponding to any subset in $\mathcal{H}(e_x'', \mathcal{A}'_{p_1})$, the contribution value from the subset in $\mathcal{H}(e_x'', \mathcal{C} - \mathcal{A}'_{p_1})$ is 0 apparently. In summary, it follows that on the basis of the bound for the sum of the contribution values from all the subsets in $\mathcal{A}'_{p_1}$ given in the above paragraph, i.e. $\sum_{j=1}^{p_1} f(e_j') \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}|)$, for the optimal solution the sum of the contribution values from all the subsets in $\mathcal{A}''_{p_2}$ (or $\mathcal{C} - \mathcal{A}'_{p_1}$) is equivalent to 0, where $\mathcal{A}''_k = \bigcup_{j=1}^{k} \mathcal{H}(e_j'', \mathcal{C} - \mathcal{A}'_{p_1})$. Obviously, the objective of the partial solution produced in the iterations within Lines 7–11 is equal to 0.

In summary, after all the iterations within Lines 2–11 in the above algorithm have been conducted, a solution with the objective $\sum_{j=1}^{p_1} f(e_j') \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}| - 1)$ and an upper bound for the sum of the contribution values from all the subsets in $\mathcal{C}$ for all solutions (or for the optimal solution), which is $\sum_{j=1}^{p_1} f(e_j') \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}|)$, are obtained. So the approximation ratio is

$$R = \frac{opt}{solution} \le \frac{\sum_{j=1}^{p_1} f(e'_j) \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}|)}{\sum_{j=1}^{p_1} f(e'_j) \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}| - 1)}$$

$$= 1 + \frac{\sum_{j=1}^{p_1} f(e'_j)}{\sum_{j=1}^{p_1} f(e'_j) \times (|\mathcal{A}'_j - \mathcal{A}'_{j-1}| - 1)}.$$

It is clear that for $\forall j (1 \le j \le p_1)$, $|\mathcal{A}'_j - \mathcal{A}'_{j-1}| \ge 2$ due to the greedy selection strategy within Lines 2–6 in the above algorithm. So

$$R \le 1 + \frac{\sum_{j=1}^{p_1} f(e'_j)}{\sum_{j=1}^{p_1} f(e'_j) \times (2-1)} \le 2. \qquad \square$$

GPTree construction The algorithm for GPTree construction, called *BuildGPTree*, is shown in Algorithm 1. After Lines 1, 3–12 are carried out, Step 1 for the construction finishes; after Line 13 ends, Step 2 and Step 3 finish.

Algorithm 1: BuildGPTree(D, σ_T^I)

```
   Input: a graph database $D$ and a minimum threshold $\sigma_T^I$
   Output: the GPTree
1  obtain the set $FIG$ of frequent induced subgraphs of the graphs in $D$, where the
   minimum induced-support is $\sigma_T^I$ ;
2  return ConsTrie($D$, $\sigma_T^I$, $FIG$) ;
   Function:ConsTrie($D$, $\sigma_T^I$, $FIG$)
3  $CP \leftarrow \emptyset$, $S \leftarrow \emptyset$ ;
4  while $\exists ig \in FIG$ s.t. $|(sup_D^I(ig) \cap D) - S| > 1$ do
5      select $ig' \in FIG$ with largest $|V_{ig'}|$ s.t. $|(sup_D^I(ig') \cap D) - S| > 1$ ;
6      $cp \leftarrow ig'$, $cp.GRP \leftarrow \{\langle g, vseq\rangle \mid \langle g, vseq\rangle \in ig'.SUP^I, g \in D - S\}$ ;
7      $CP \leftarrow CP \cup \{cp\}$, $S \leftarrow S \cup (sup_D^I(ig') \cap D)$ ;
8  while $\exists ig \in FIG$ s.t. $|(sup_D^I(ig) \cap D) - S| = 1$ do
9      select $ig'' \in FIG$ with largest $|V_{ig''}|$ s.t. $|(sup_D^I(ig'') \cap D) - S| = 1$ ;
10     $cp \leftarrow ig''$, $cp.GRP \leftarrow \{\langle g, vseq\rangle \mid \langle g, vseq\rangle \in ig''.SUP^I, g \in D - S\}$ ;
11     $CP \leftarrow CP \cup \{cp\}$, $S \leftarrow S \cup (sup_D^I(ig'') \cap D)$ ;
12 complete the sequence of all the vertices of each graph in $D$ ;
13 according to the sequence of all the vertices of each graph in $cp.GRP$, where
   $cp \in CP$, obtain the corresponding GVCode, then construct the trie of GPTree
   and return the trie ;
```

BuildGPTree first obtains the set *FIG* of frequent induced subgraphs of the graphs 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

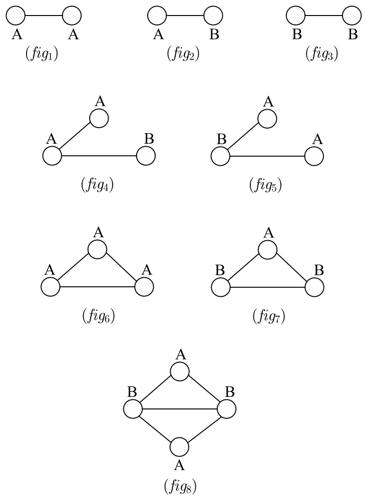


Fig. 6 Running example: frequent induced subgraphs

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 3–11). 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 the vertices of the graph (Line 12). After the order on the vertex set of each graph has been found, a GVCode of each graph is obtained according to the order. The last step is to construct the trie of GPTree.

Example 2 Figure 6 shows all the frequent induced subgraphs of the graphs in the database in Fig. 1 with $\sigma^I_T = 0.5$. The proposed algorithm for induced subgraph selecting selects fig_8 for g_3 and g_4, and fig_6 for g_1 and g_2. After the induced subgraph selecting has been conducted (Lines 3–11 in Algorithm 1), four pairs in $\bigcup_{cp\in CP} cp.GRP$ are produced, i.e. $\langle g_1, \langle v_1, v_2, v_3\rangle\rangle$, $\langle g_2, \langle v_1, v_2, v_3\rangle\rangle$, $\langle g_3, \langle v_1, v_2, v_3, v_4\rangle\rangle$, $\langle g_4, \langle v_1, v_2, v_3, v_4\rangle\rangle$. Then, after the sequence of all the vertices of each graph has been completed, a GVCode of each graph is obtained according to the sequence and the GPTree is constructed as shown in Fig. 4.

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

3.3 Indices on graph databases

3.3.1 *Feature generation*

To reduce the cost of query processing, indices constituted by the features of a given graph database are constructed. In this section, 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*, i.e. the number of candidates after filtering, to that of all significant frequent subgraphs in practice.

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. For two distinct frequent subgraphs g and g' such that $g \sqsubseteq g'$, if $|sup(g)|$[3]$= |sup(g')|$, we claim that it is advisable to select g' as a feature rather than both of them or only g. The reason is as follows. Given a graph database D, a feature set F of D, and a supergraph query with query graph q, the candidate answer set is $C_q = D - \bigcup_{f \not\sqsubseteq q, f \in F} sup_D(f)$. Therefore, if $g' \not\sqsubseteq q$ then all the graphs in $sup_D(g')$ can be removed from C_q, and no matter whether $g \sqsubseteq q$ or not C_q cannot be refined further; otherwise, if $g' \sqsubseteq q$, then $g \sqsubseteq q$ and both g' and g cannot be used to refine C_q. Thus, if g' is a feature, then g need not be a feature with the guarantee of achieving the same filtering power. In this way, g' is called more *helpful* than g for filtering during query processing, and g is not *helpful* w.r.t. g'. 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 $sup(g) = \bigcup_{i=1}^{k} sup(g_i)$, then g is not *helpful* w.r.t. SG. The reason is that if all the subgraphs in SG are features, then selecting g as another feature is not able to improve the overall filtering power any more, i.e. identifying less candidates. The helpfulness of a subgraph is quantified as follows, called significance.

Feature set On the basis of the analysis of the filtering power of frequent subgraphs, we define the significance metric δ w.r.t. g as $\delta(g) = \frac{|sup(g)|}{|\bigcup_{i=1}^{m} sup(f_i)|}$,[4] where g is a subgraph, and $f_1, f_2, \ldots, f_m$ are all the features that contain g. Then, the *feature set* is defined to be comprised of all significant frequent subgraphs, i.e. all the frequent subgraphs with significance no less than a user-specified minimum significance threshold δ_{min}. Here, the minimum support threshold is set to be $1/|D|$ when the size of a subgraph pattern is less than 4, in order to enable all the graphs in D to be supported probably by some features. It follows that all maximal frequent subgraphs are selected as features.

[3]I.e. $|sup_D(g)|$. The graph database D is omitted here for ease of presentations, similarly hereinafter.

[4]If the denominator is 0, the fraction is defined to be equal to a very large number that is always larger than any δ_{min}.

Among all frequent subgraphs of the graphs in a graph database D, the selecting of significant frequent subgraphs as features is discussed as follows. For two unselected frequent subgraphs g and g', such that $g \sqsubseteq g'$, $g \neq g'$ and $sup_D(g) = sup_D(g')$ (or $|sup_D(g)| = |sup_D(g')|$), let the data structure $\underline{F}$ store all the features selected till now, and all the selected features in $\underline{F}$ that contain g' be $f_1, f_2, \ldots, f_m$. Given δ_{min}, $\delta_{min} > 1$, Case 1, if $\frac{|sup(g')|}{|\bigcup_{i=1}^{m} sup(f_i)|} \geq \delta_{min}$, then g' should be added into $\underline{F}$ before g being added, because g' is more helpful than g for filtering; and then g should not be added into $\underline{F}$, because after g' is added into $\underline{F}$, g' is a feature in $\underline{F}$ that contains g, and $\frac{|sup(g)|}{|(\bigcup_{i=1}^{m} sup(f_i)) \cup sup(g')|} < \delta_{min}$. Case 2, otherwise, i.e. $\frac{|sup(g')|}{|\bigcup_{i=1}^{m} sup(f_i)|} < \delta_{min}$, then $\frac{|sup(g)|}{|\bigcup_{i=1}^{m} sup(f_i)|} < \delta_{min}$, thus both g' and g are not significant and both should not be added into $\underline{F}$. So in any case g should be discarded. In the existing works, the set CSG of closed frequent subgraphs (Yan and Han 2003) is defined as $CSG = \{g \mid g \in FSG$ and $\nexists g' \in FSG$ s.t. $g \sqsubseteq g', g \neq g', sup(g) = sup(g')\}$, 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 fewer than all frequent subgraphs in practice (Yan and Han 2003).

Exact algorithm for generating feature set The exact algorithm for generating a 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. According to depth-first search order in the mining algorithm, the containment relationship among some closed frequent subgraphs can be obtained in Step 1, 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.

Approximate algorithm for generating feature set The approximate algorithm for generating a feature set directly mines features instead of refining after mining. During the mining of closed frequent subgraphs, for a pattern g in the space of closed frequent subgraph patterns in D, if $\frac{|sup(g)|}{|\bigcup_{\tilde{g}} sup(\tilde{g})|} \geq \delta_{min}$, where $\tilde{g}$ denotes all the direct supergraphs of g that are frequent, i.e., the denominator is the cardinality of union over all the support sets of such $\tilde{g}$, then g is selected as a feature; otherwise, it is not selected. 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 the costly refining step, i.e. Step 2 of the exact algorithm above.

Theorem 3 *The feature set generated by the approximate algorithm is a subset of that generated by the exact algorithm.*

Proof For a subgraph g in the space of closed frequent subgraph patterns in D and any set S of subgraphs, we have $|\bigcup_{g'} sup_D(g')| \leq |\bigcup_{\tilde{g}} sup_D(\tilde{g})|$, where $\tilde{g}$ denotes all the frequent direct supergraphs of g, and g' denotes all the frequent proper supergraphs of g that belong to S. Thus, for each subgraph f in the feature set obtained by using the approximate algorithm, i.e. $\frac{|sup_D(f)|}{|\bigcup_{\tilde{f}} sup_D(\tilde{f})|} \geq \delta_{min}$, where $\tilde{f}$ denotes all the frequent direct supergraphs of f, we conclude that $\frac{|sup_D(f)|}{|\bigcup_{i=1}^{m} sup_D(f_i)|} \geq \frac{|sup_D(f)|}{|\bigcup_{\tilde{f}} sup_D(\tilde{f})|} \geq \delta_{min}$, where $f_1, f_2, \ldots,$ and f_m are all the features obtained by the exact algorithm that contain f. Then, f must be selected as a feature by the exact algorithm. □

The approximate algorithm is very suitable for the scenarios where closed frequent subgraphs are of large amount, so as to speed up the process of feature generation. It shows approximate filtering power in experiments during query processing.

3.3.2 Feature ordering

Although all the features extracted are significant, in the filtering step of query processing 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. For example, consider two features f_1 and f_2, s.t. $sup_D(f_1) \supseteq sup_D(f_2)$. For a query graph q, if $f_1 \nsubseteq q$, all the graphs in $sup_D(f_1)$ can be immediately filtered out, and then, f_2 need not be examined because no matter whether it is contained by q or not we cannot filter out any other graphs by $sup_D(f_2)$. Thus one subgraph isomorphism testing from a feature to the query graph is saved. Conversely, 2 sub-iso testings have to be performed with no saving. The above example shows that the order on the feature set can influence the overall cost of query processing.

In the following discussion, D is a graph database and F is a feature set of D. For $\forall k$, $1 \leq k \leq |F|$, $sup_D(f_k)$ is denoted by A_k indiscriminatingly. Please note that D is omitted in the remainder of the subsection for simplicity.

Let $h : 2^F \to [0, 1]$ be a function, where $h(F')$ is equal to the joint probability that all the features in F' are subgraphs of a given query graph, where $F' \subseteq F$. Let $g : \Pi \times F \to [0, 1]$ be a function, where Π is the set of all the orders defined on F; and $g(\prec, f'_t) = 1 - h(\{f'_k \mid f'_k \in F, sup(f'_k) \supseteq sup(f'_t), k < t\})$, where $\prec$ is an order on F, i.e. $\langle f'_1, f'_2, \ldots, f'_{|F|}\rangle$. Thus, the problem of determining the optimal order on a feature set can be defined as follows.

Input: a graph database D, a feature set F of D, and a function $g : \Pi \times F \to [0, 1]$, where Π is the set of all the orders defined on F.

Output: an order $\prec$ on F, i.e. $\langle f'_1, f'_2, \ldots, f'_{|F|}\rangle$, where for $1 \leq k \leq |F|$, $f'_k \in F$.

Objective: Maximize $\sum_{k=1}^{|F|} g(\cdot, f'_k)$.

Given a query graph q, the objective $\sum_{k=1}^{|F|} g(\cdot, f'_k)$ is the expected number of sub-iso testings that can be saved compared to testing one by one (i.e., $|F|$) when sub-iso testings from all the features in F to q are performed during the filtering step of query processing. The following theorem is presented to expose the case in which the maximum objective can be obtained.

Theorem 4 *An order $\prec^*$ on F is optimal, if and only if, for any two features f_i and f_j in F, f_i is before f_j according to $\prec^*$ if $sup(f_i) \supset sup(f_j)$.*

Proof (sketch) Necessity: the necessity is proved by contradiction. Let $\prec^*$ be an optimal order on F, i.e. $\langle f'_1, f'_2, \ldots, f'_{|F|}\rangle$, such that $\sum_{k=1}^{|F|} g(\cdot, f'_k)$ is maximized, where $f'_k \in F$ for $1 \le k \le |F|$. Suppose that there exist two features f'_a and f'_b in F, such that $sup(f'_a) \subset sup(f'_b)$ and f'_a is before f'_b i.e. $a < b$. Let all the features that are before f'_a and are supergraphs of f'_b be $f'_{c_1}, \ldots, f'_{c_m}$, where $c_1 < \cdots < c_m < a$. If there exist such features, we denote f'_{c_1} by f'_x; otherwise, we denote f'_a by f'_x. We denote f'_b by f'_y. Then, we construct another order $\prec''$ on F based on $\prec^*$ by appropriately moving features $f'_x, f'_{x+1}, \ldots, f'_{y-1}, f'_y$ in the way given in the next paragraph and retaining the other features' positions; and we claim that $\prec''$ is a better order than $\prec^*$ in terms of the objective. Thus, a contradiction is obtained.

On the basis of $\prec^*$, i.e. $\langle f'_1, f'_2, \ldots, f'_{|F|}\rangle$, the order $\prec''$ is constructed as follows. Step 1. The positions of features $f'_1, f'_2, \ldots, f'_{x-1}, f'_{y+1}, f'_{y+2}, \ldots, f'_{|F|}$ are not changed. Step 2. Features $f'_x, f'_{x+1}, \ldots, f'_{y-1}, f'_y$ are rearranged by bubble sort in non-ascending order of supports. The order $\prec''$ is derived from $\prec^*$ after the above two steps being performed. Please note that during the bubble sort, for two neighboring features f'_r and f'_{r+1}, if $|sup(f'_r)| < |sup(f'_{r+1})|$ then we swap f'_r and f'_{r+1}; otherwise, we do not swap.

Sufficiency: let $\tilde{\prec}$ be an order on F such that for $\forall f_i, f_j \in F$, f_i is before f_j if $sup(f_i) \supset sup(f_j)$. For $\forall f_i \in F$, all the features with support sets being proper supersets of $sup(f_i)$ must be before f_i according to $\tilde{\prec}$. Thus, it follows that for $\forall f_i \in F$, $\{f \mid f \in F, sup(f) \supset sup(f_i), f$ is before f_i according to $\tilde{\prec}\} = \{f \mid f \in F, sup(f) \supset sup(f_i)\}$. In general, without specifying query graphs, it is assumed that the features with the same support set influence the function of $h(\cdot)$ identically. Thus, all orders on the set of the features with the same support set affect the objective identically. Then, for $\forall f_i \in F$, $\{f \mid f \in F, sup(f) \supseteq sup(f_i), f$ is before f_i according to $\tilde{\prec}\} = \{f \mid f \in F, sup(f) \supseteq sup(f_i)\}$, i.e., $g(\tilde{\prec}, f_i) = \max_{\prec}\{g(\prec, f_i)\}$. Therefore, $\sum_{k=1}^{|F|} g(\tilde{\prec}, f_k) = \max_{\prec}\{\sum_{k=1}^{|F|} g(\prec, f_k)\}$, i.e., $\tilde{\prec}$ is an optimal order on F. □

Feature ordering The algorithm for ordering features is arranging features in non-ascending order in terms of their supports, which derives an optimal order.

3.3.3 Index structures

The index structures, consisting of all features with an optimal order, include CRGraph and FGPForest.

CRGraph The CRGraph index structure is a directed acyclic graph (DAG). Given a graph database D, the feature set F of D and an optimal order $\prec$ on F, for two features $f_1, f_2 \in F$, if $sup_D(f_1) \supset sup_D(f_2)$ then we say $sup_D(f_1)$ direct contains $sup_D(f_2)$. This direct containment relationship between the support sets of feature classes 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. The DAG is the CRGraph index structure.

CRGraph construction The algorithm for CRGraph construction, called *BuildCRGraph*, is shown in Algorithm 2, which is divided into two parts: ordering features (Line 1), and creating vertices and directed edges to obtain the CRGraph structure (Lines 2–11). For each feature f there is a data structure $CLOS_f$ used to record all the features f' s.t. $sup_D(f) \supset sup_D(f')$. During the loop in Lines 2–11, the data structure *TransCLOS* is used to record all the features f' s.t. $sup_D(f) \supset sup_D(f')$ and $(\exists f'' \in F)(sup_D(f) \supset sup_D(f'')$ and $sup_D(f'') \supset sup_D(f'))$.

Algorithm 2: BuildCRGraph(F)

```
Input: a feature set $F$, where for $\forall f \in F$, $sup(f)$ is attached
Output: the CRGraph
1  sort $F$ in non-ascending order in terms of support (please note that the obtained
   optimal order on $F$ is denoted by $\prec$) ;
2  for each $f \in F$ in descending order of $\prec$ do
3      create a vertex $v_f$ of CRGraph which corresponds to $f$ ;
4      $CLOS_f \leftarrow \emptyset$ ;
5      $TransCLOS \leftarrow \emptyset$ ;
6      for each $f'$ s.t. $f' \in F$ and $f' \notin TransCLOS$ in ascending order of $\prec$ do
7          if $sup(f) \supset sup(f')$ then
8              create a directed edge of CRGraph from $v_f$ to $v_{f'}$ ;
9              $CLOS_f \leftarrow CLOS_f \cup \{f'\}$ ;
10             $TransCLOS \leftarrow TransCLOS \cup CLOS_f$ ;
11     $CLOS_f \leftarrow CLOS_f \cup TransCLOS$ ;
12 return the constructed CRGraph ;
```

Complexity analysis of CRGraph construction The time complexity of feature ordering is $O(|F| \times \log(|F|) \times |D|)$ with no extra space usage. The time complexity of producing vertices and directed edges is $O(|F|^2 \times |D|)$ and the space usage is $O(|F|^2)$ for $CLOS(f)$ and $O(|F|)$ for *TransCLOS*. In sum, the time complexity of CRGraph construction is $O(|F|^2 \times |D|)$ and the space complexity is $O(|F|^2)$.

FGPForest The FGPForest index structure is a forest, comprised of GPTrees of disjoint sets of features. The GPTrees in a FGPForest are ordered. Obviously, in a FGPForest some but not all common prefixes of GVCodes of all features are combined. Please note that an optimal order on the feature set is preserved in its FGPForest.

FGPForest Construction The algorithm for FGPForest construction, called *BuildFGPForest*, is shown in Algorithm 3. It invokes the function ConsTrie defined in GPTree construction (Algorithm 1). With *CRGraph* as input, all the features in F are organized to a forest comprised of multiple GPTrees of disjoint sets of F.

Algorithm 3: BuildFGPForest(F, *CRGraph*, σ^I_{FT})

```
Input: a feature set $F$, the $CRGraph$ of $F$, and a minimum threshold $\sigma^I_{FT}$
Output: the $FGPForest$
1 initialize $FGPForest$ by the empty forest, $CRGraph' \leftarrow CRGraph$ ;
2 obtain the set $FFIG$ of frequent induced subgraphs of the features (or graphs) in
  $F$, where the minimum induced-support is $\sigma^I_{FT}$ ;
3 while there exist at least one vertex in $CRGraph'$ do
4     $FreeF \leftarrow$ the set of all the features of the vertices in $V_{no\text{-}in\text{-}edge}$, where
      $V_{no\text{-}in\text{-}edge}$ is the set of all the vertices in $CRGraph'$ which have no incoming
      edges ;
5     $FGPTree \leftarrow$ ConsTrie($FreeF$, $\sigma^I_{FT}$, $FFIG$) ;
6     append $FGPTree$ to $FGPForest$ with making $FGPTree$ the last tree in the
      ordered forest $FGPForest$ ;
7     remove all the vertices in $V_{no\text{-}in\text{-}edge}$ and all the outcoming edges of each
      vertex in $V_{no\text{-}in\text{-}edge}$ from $CRGraph'$ ;
8 return $FGPForest$ ;
```

Complexity analysis of FGPForest construction In all the iterations of the loop in Line 3, the time complexity of Lines 4 and 7 is $O(|F| + |E_{CRGraph}|)$ in total and the space complexity is $O(|F|)$; and the time complexity of Line 5 is $O(|F| \times |FFIG| + \sum_{f\in F}(|V_f| + |E_f|))$ in total and the space complexity is $O(\sum_{fig\in FFIG}(|sup^I_F(fig)| \times |V_{fig}|) + |F| \times |FFIG| + \sum_{f\in F}(|V_f| + |E_f|))$ if *vseq*s are stored in memory. The time complexity of Line 5 is $O(|F| + |E_{CRGraph}|)$ and the space complexity is $O(|F| + |E_{CRGraph}|)$. In sum, the time complexity of FGPForest construction is $O(|E_{CRGraph}| + |F| \times |FFIG| + \sum_{f\in F}(|V_f| + |E_f|))$ apart from the time taken by frequent induced subgraph mining on F, and the space complexity is $O(\sum_{fig\in FFIG}(|sup^I_F(fig)| \times |V_{fig}|) + |E_{CRGraph}| + |F| \times |FFIG| + \sum_{f\in F}(|V_f| + |E_f|))$ apart from the space consumed by frequent induced subgraph mining on F if *vseq*s are stored in memory.

3.3.4 Discussions on graph database organizing and index creating

The preprocessing in the proposed method includes organizing a given graph database into a GPTree, and creating the indices of CRGraph and FGPForest. The process of the specified frequent induced subgraph mining on databases 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 the proposed method consists of two steps. A 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 the exact algorithm for feature generation is adopted, and the initial feature set is just the outputted feature set if the approximate method for feature generation is adopted; features are ordered and the indices of CRGraph and FGPForest are created.

3.4 Query processing

3.4.1 Subgraph isomorphism testing from multiple to one

Given a GPTree of a set of small graphs, and one large graph, for each small graph, in order to determine whether it is a subgraph of the large graph, an algorithm is proposed in this section.

The algorithm for testing subgraph isomorphism from multiple small graphs organized in a GPTree to one large graph, called *GPTreeTest*, is shown in Algorithm 4, which outputs the set of IDs of the graphs in the GPTree that are contained by the large graph.

In Algorithm 4, the data structure *psi*, used to record a partial sub-iso, stores a sequence of some vertices of the large graph q. For each node n in T, $n.GID$ is the set of graph-IDs attached to n, and *n.flag*, *n.alOut*, *n.ableChildCnt* are three data structures (or variables), which are initialized by '*true*', '*false*', and the number of n's children, respectively, in the algorithm. The initialized values of these three variables represent that n should be examined subsequently, that $n.GID$ has not been outputted yet, and the number of the children of n whose *flag* variable is '*true*', respectively. $CHILD(n)$ is the set of n's children in T. The core of the algorithm is the recursion procedure SubIsoOnGPTree, which takes a node n of a GPTree and a partial sub-iso *psi* as input. In the beginning of the procedure, if n is associated with some graph-IDs that have not been outputted, they should be outputted (Lines 4–6). Then, if all the children of n need not be examined, the procedure will directly return (Lines 7–9). In the next step, the procedure pre-order traverses the subtree rooted by each child *cn* of n such that *cn.flag* is '*true*' (Lines 10–12). The data structure MV records all the matched vertices in q, such that, each of these vertices and all the vertices in *psi* constitute a subgraph isomorphism from the graph corresponding to the path from n_{root} to *cn* to q. For each matched vertex v in MV, the corresponding partial sub-iso psi' is constructed by concatenating v to *psi* (Line 14), and then SubIsoOnGPTree is performed from the child *cn* recursively (Line 15). Once a contained graph is found, it is recorded and will not be considered again during subsequent searching (Lines 4–9 and 16–21). At last, *IDSET* is the set of IDs of the graphs in T that are contained by q.

Let the path from the node n_{anc} to the node n_{desc} in a GPTree be denoted by $n_{anc} \rightsquigarrow n_{desc}$. In the GPTree T, the graph corresponding to $n_{root} \rightsquigarrow n$, denoted by $g_{n_{root} \rightsquigarrow n}$, is an induced subgraph of the graphs that correspond to the paths from n_{root} to any descendants of n. Thus, in the course of the algorithm, *psi* records a partial subgraph isomorphism from $g_{n_{root} \rightsquigarrow n}$ to q (in particular, the vertex of $g_{n_{root} \rightsquigarrow n}$ that corresponds to n_i in T is mapped to the ith vertex in *psi*), and $g_{n_{root} \rightsquigarrow n}$ represents an induced subgraph of multiple graphs accommodated in T. Therefore, when sub-iso testing from multiple graphs in a GPTree to a large graph is performed, their common induced subgraphs corresponding to one path of the GPTree are examined together, and are only tested once altogether. In this way, a number of sub-iso testings are saved.

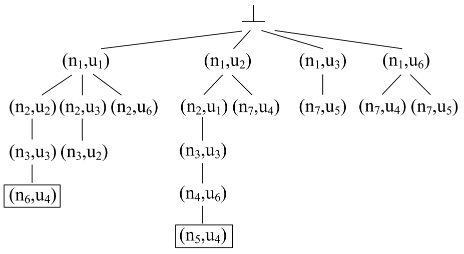
Algorithm 4: GPTreeTest(T, q)

```
Input: a GPTree $T$ with root $n_{root}$, and a large graph $q = (V_q, E_q, \Sigma_q, l_q)$
Output: $IDSET$
1  $IDSET \leftarrow \emptyset$ ;
2  SubIsoOnGPTree($n_{root}$, $\emptyset$) ;
3  return $IDSET$ ;
   Procedure: SubIsoOnGPTree($n$, $psi$)
4  if $n.GID \neq \emptyset$ and $n.alOut = false$ then
5      $IDSET \leftarrow IDSET \cup n.GID$ ;
6      $n.alOut \leftarrow true$ ;
7  if $n.ableChildCnt = 0$ then
8      $n.flag \leftarrow false$ ;
9      return ;
10 for each $cn \in CHILD(n)$ do
11     if $cn.flag = false$ then
12         continue ;
13     let $cn$ be denoted by $\langle (j, l_j), (i_1, l_{i_1}, l_{i_1 j}), \ldots, (i_m, l_{i_m}, l_{i_m j}) \rangle$,
       $MV = \{v \mid v \in V_q - \{u|u$ is in $psi\}$, $l_q(v) = l_j$, and for
       $\forall k(1 \leq k \leq m)$, $(psi[i_k], v) \in E_q$ and $l_q(psi[i_k], v) = l_{i_k j}$ $\}$,
       where $psi[i_k]$ is the $i_k$-th vertex in $psi$
       for each $v \in MV$ do
14         compute $psi'$ by concatenating $v$ to $psi$ ;
15         SubIsoOnGPTree($cn$, $psi'$) ;
16         if $cn.flag = false$ then
17             $n.ableChildCnt$– ;
18             if $n.ableChildCnt = 0$ then
19                 $n.flag \leftarrow false$ ;
20                 return;
21             break ;
```

Example 3 Figure 7 shows the state search space, in which each state[5] represents a recursion of the procedure SubIsoOnGPTree, when sub-iso testing from all the graphs in the GPTree in Fig. 4 to q_2 in Fig. 5 is performed using GPTreeTest. For the state t associated with (n_6, u_4), the recursion path in the GPTree is $\langle n_1, n_2, n_3, n_6 \rangle$ and $psi = \langle u_1, u_2, u_3, u_4 \rangle$. This psi in t (and this recursion path) represents a subgraph isomorphism (or injective function) from the graph corresponding to the GVCode $\langle n_1, n_2, n_3, n_6 \rangle$ to q_2, where n_i is defined in Fig. 4b. The rectangle in a state t represents that some graphs contained by 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, the algorithm returns $\{g_1, g_2\}$ as result finally.

[5]Each state is associated with a node-vertex pair, in which the first component is the current node n and the second component is the last vertex in psi.

Fig. 7 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 graph is the number of vertices of the small graph sg, denoted by $|V_{sg}|$, since a subgraph isomorphism must involve all the vertices in V_{sg}. Hence, here the response time of GPTreeTest is analyzed in terms of the number of sub-iso testings with different values of $|V_{sg}|$ explicitly. Let m be the number of the graphs in a given GPTree. For each j, let m_j be the number of the graphs in the GPTree that share the same common prefix, and let $cplen_j$ be the number of the codes (or vertices) in this common prefix (or common induced subgraph), where $\sum_{j=1}^{k} m_j = m$. As analyzed in Sect. 3.2.2, we have that for the m_j graphs with the same $cplen_j$-length common prefix, the testing time saved by this common prefix is $T(cplen_j) \times (m_j - 1)$, where $T(cplen_j)$ 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 $m \times T_{isoEach} - \sum_{j=1}^{k} T(cplen_j) \times (m_j - 1)$, where $T_{isoEach}$ is the average time taken by individually sub-iso testing from each of the m graphs in the GPTree to a query. In addition, by utilizing consecutive storage such as arrays, the overall time taken by initializing the variables *.flag*, *.alOut*, *.ableChildCnt* for all the nodes in the GPTree is $O(\sum_{g \text{ in the GPTree}} |V_g|)$, which is small compared to the above searching time.

3.4.2 On-line redundant features shedding

As analyzed in Sect. 3.3.2, although all features are significant, in the filtering step of query processing there probably exist unnecessary sub-iso testings from features to query graphs. For a query graph, such features that sub-iso testings from them to the query graph is unnecessary are called on-line redundant features.

On-line redundant features shedding The algorithm for on-line redundant features shedding, called *RedunFShedding*, is shown in Algorithm 5. Features are examined in an optimal order preserved in the ordered *FGPForest* (Line 4). During the process, if a feature f is not contained in q, the features corresponding to all the descendants of v_f in *CRGraph* are added to the redundant feature set *SF*, where v_f is the vertex in *CRGraph* corresponding to f. After that, all the features in *SF* need not be tested for sub-iso from them to q (Lines 6–12). In the next iteration of Line 5, the next feature that is not in *SF* is examined. This process repeats until all features are either examined or shed. Thus, the eventual shed feature set *SF* records all the features that are avoided from subgraph isomorphism testing.

Algorithm 5: RedunFShedding(*CRGraph*, *FGPForest*, *q*)

```
   Input: the indices of CRGraph and FGPForest, and a query graph $q$
   Output: the shed feature set $SF$
1  $SF \leftarrow \emptyset$ ;
2  for each vertex $v$ in CRGraph do
3      $v.expanded \leftarrow false$;
4  for the next tree FGPTree of FGPForest do
5      for each feature $f$ organized in FGPTree do
           let $v_f$ be the vertex in CRGraph corresponding to $f$
6          if $f \notin SF$ then
7              if $f \nsubseteq q$ then
8                  add the features corresponding to all the children of $v_f$ in
                   CRGraph to $SF$ ;
9                  $v_f.expanded \leftarrow true$ ;
           else
10             if $v_f.expanded = false$ then
11                 add the features corresponding to all the children of $v_f$ in
                   CRGraph to $SF$ ;
12                 $v_f.expanded \leftarrow true$ ;
13 return $SF$ ;
```

Complexity analysis of redundant features shedding RedunFShedding involves two kinds of operations, the one is sub-iso testing from features to q, and the other is children enumeration from *CRGraph*. Let T_{isoF} be the average time cost of sub-iso testings from each feature to q. Because of GPTreeTest being used to test sub-iso (Line 7), which will be mentioned in Sect. 3.4.3, the time cost of all sub-iso testings is no more than $(|F| - |SF|) \times T_{isoF}$. The time complexity of children enumeration is $O(|F| + |E_{CRGraph}|)$, where $E_{CRGraph}$ is the edge set of the *CRGraph*. In addition, the space usage of children enumeration is $O(|F|)$ for *SF* and $\Theta(|V_{CRGraph}|) = O(|F|)$ for the *.expanded* component of each vertex in *CRGraph*, where $V_{CRGraph}$ is the vertex set of the *CRGraph*. In sum, the time complexity of RedunFShedding apart from sub-iso testings on *FGPForest* is $O(|F| + |E_{CRGraph}|)$, and the space complexity is $O(|F|)$.

3.4.3 Integrated query processing method

Given a graph database D, the compact organization GPTree of D and the indices of *CRGraph* and *FGPForest*, which implies the feature set F, are constructed in the preprocessing phase. The integrated method for processing a supergraph query with query graph q consists of two steps. In the first filtering step of query processing, the candidate answer set is identified, i.e. $C_q = D - \bigcup_{f \nsubseteq q, f \in F} sup_D(f)$. The filtering step integrates on-line shedding redundant features and performing GPTreeTest on each GPTree in *FGPForest* in turn for testing sub-iso from features to q (or for Line 7 in Algorithm 5). Please note that once a feature is known to be in the shed feature set *SF*, it will be not examined when further performing GPTreeTest on GP-Trees in *FGPForest* for processing this query. In the second verification step of query processing, GPTreeTest is performed on the subtree of GPTree of D which only accommodates all the graphs in C_q, and the answer to this query, $Answer(q)$, is obtained at last.

3.4.4 Discussions

The support for external storage is briefly discussed 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 GP-Tree 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 $T_{ig\text{-}m}$ mentioned in Sect. 3.2.2 is $O(|D| \times |FIG| + \sum_{g \in D} |V_g|)$ operations in memory and the disk-IOs involving storing the vertex sequences of all the graphs in D; the memory usage apart from $S_{ig\text{-}m}$ is $O\big(\sum_{ig \in FIG}(|sup_D^I(ig)| \times |V_{ig}|) + |D| \times |FIG|\big)$ if *vseq*s are stored in memory.

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 the GPTree of D, $GVCode(g)$ is generated with its codes in an arbitrary order, and the generated $GVCode(g)$ is inserted into the GPTree subsequently; Step 2, GPTreeTest on *FGPForest* 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 of D.

4 Experimental evaluation

In this section, the experimental studies that validate the effectiveness and efficiency of the proposed method, named GPTree&CRGraph (or GPT&CRG), is presented, by comparing it with the state-of-the-art method, cIndex (Chen et al. 2007). 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 PIV3.0 GHz PC with 2 GB RAM, running Redhat Linux 8.0. Both cIndex and GPTree&CRGraph are implemented in C and compiled by gcc compiler (-O2).

4.1 AIDS antiviral screen dataset

The experiments described in this subsection 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&CRGraph are set as follows. (1) In

Table 1 Preprocessing performance for AIDS

	GPTree&CRGraph(E:0.8)		GPTree&CRGraph(A:0.9)		cIndex	
minsup	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

cIndex, we randomly draw 10,000 graphs to form a dataset W, then divide W to a query log set $\underline{L}$ (8000) 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, $\underline{min\_size}$, is still set 100. (2) In GPTree&CRGraph, query logs are not used, and the query set is the same as that in cIndex; the minimum significance threshold δ^E_{min} is 1/0.8 for the exact method and δ^A_{min} is 1/0.9 for the approximate method; and the minimum support threshold σ^I_T for the GPTree of the given database and that for index construction σ_F are both 0.05, and that for *FGPForest*, σ^I_{FT}, is 0.1. In order to build a graph database, we apply frequent subgraph mining on AIDS and retain all the 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, cIndex-Basic is selected for the comparison of index construction because it yields the fewest features, and cIndex-TopDown for the comparison of query processing due to its greatest efficiency in query processing rather than cIndex-BottomUp and cIndex-Basic.

The index size and the construction time of cIndex-Basic and those of GPTree&CRGraph are first tested. As mentioned in Sect. 3.3.4, the GPTree construction and the proposed indices construction are integrated. So we compare the overall preprocessing time of GPTree&CRGraph with the time for index construction of cIndex (or cIndex-Basic). Table 1 reports the construction time and the number of features on varying $\sigma_F (= \sigma^I_T)$ or $\underline{\sigma_c}$.

As shown in Table 1, the time of constructing GPTree and indices in the proposed method 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 (Chen et al. 2007) and query logs, which is very costly. Although the number of the 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 advantage of the trie structure of GPTree.

To evaluate the query performance of the proposed method, the 2000 queries are divided into eight bins: [0, 10), [10, 20), [20, 30), [30, 40), [40, 100), [100, 200), [200, 500), [500, ∞), based on the size of the average query answer set, i.e. the number of the graphs in the given graph database that are contained by the query. Figure 8

Fig. 8 Query processing time

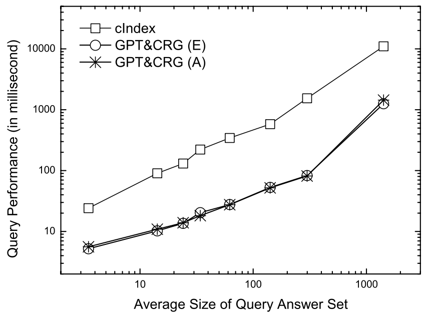
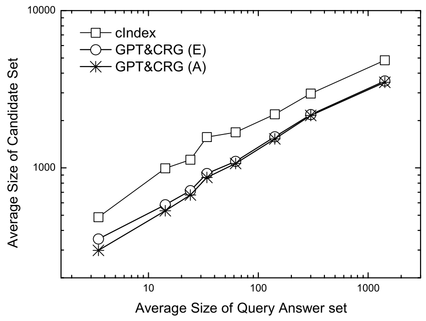


Fig. 9 Candidate answer set size



reports that the average query processing time using the proposed method 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.

To verify that the generated features are significant, the candidate answer set size is focused in Fig. 9. 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&CRGraph is several times smaller than that by cIndex-TopDown. Since $\delta^A_{min}(1/0.9) < \delta^E_{min}(1/0.8)$, the feature set generated by the approximate method shows a very close filtering power to that by the exact one.

Next, we assess the effect of δ_{min} in the exact and approximate feature generation methods on the feature set size $|F|$ and the candidate set size $|C_q|$ in Fig. 10. In this experiment, the query set [20, 30) is processed on the dataset $D_{10,000}$. It shows that the average size of 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 feature set generated by the approximate method 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, shown in Table 1.

To evaluate scalability, four datasets is generated by randomly selecting graphs from D_{init}, whose sizes range from 10,000 to 70,000. Q is chosen as the query set.

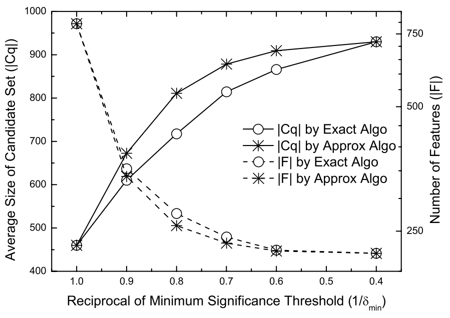


Fig. 10 Sensitivity of δ_{min}

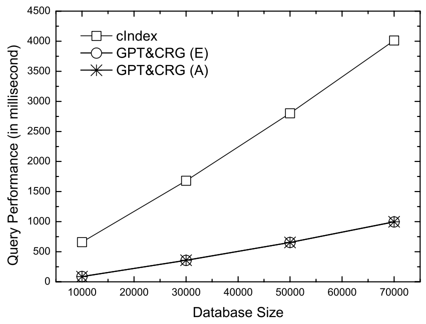


Fig. 11 Query processing time by varying database sizes

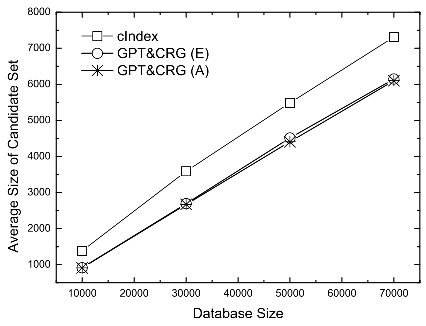


Fig. 12 Candidate answer set size by varying database sizes

Figures 11 and 12 report the query processing time and average candidate set size on the databases of various sizes. It shows the high scalability of GPTree&CRGraph.

4.2 Synthetic dataset

In this subsection, the performance studies on synthetic datasets are conducted. The broadly used graph generator (Kuramochi and Karypis 2001) is used to generate datasets, which relates to six parameters here: 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

Fig. 13 Query processing time on synthetic databases

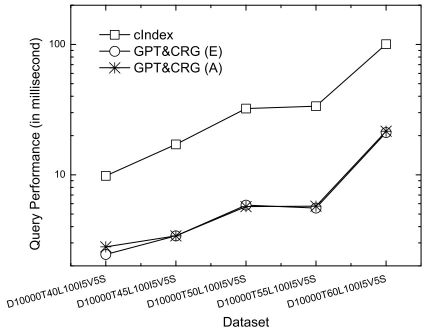
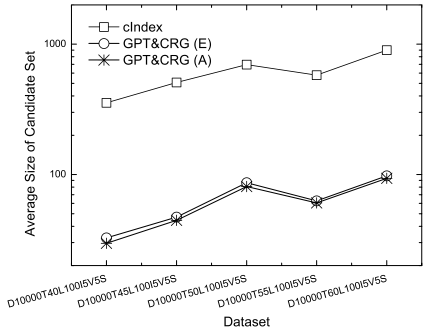


Fig. 14 Candidate answer set size on synthetic databases



seed graphs are randomly selected and inserted into a graph one by one until the graph reaches its expected size T.

For conducting experiments on the datasets with different characteristics from the real one, we individually generate database and queries. The database generated is $D10kT15L100I5V5S$, and query sets are $D10kT40L100I5V5S$, $D10kT45L100$-$I5V5S$, $D10kT50L100I5V5S$, $D10kT55L100I5V5S$ and $D10kT60L100I5V5S$. $D10kT15L100I5V5S$ denotes a set of 10,000 graphs such that the average size of these graphs is 15 and there are 5 vertex labels altogether. Each query set is divided into a query log set (8000) and a testing query set (2000). The parameters are set as follow. $\underline{\sigma_c} = 0.10$, $\underline{min\_size} = 100$; in the setting of GPTree&CRGraph, the testing query set is the same, and $\delta^E_{min} = 1/0.7$, $\delta^A_{min} = 1/0.8$, $\sigma^I_T = \sigma_F = \sigma^I_{FT} = 0.10$.

Figure 13 reports that the average query processing time is much less than that using cIndex-TopDown. Figure 14 shows that the candidate set size obtained by using the features generated in GPTree&CRGraph is still much smaller than that by cIndex-TopDown. 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 of answering arbitrary in structure graph queries, some filtering-and-verification based approaches are proposed to upgrade performance. In these approaches, the first kind (Yan et al. 2005; Zhang et al. 2007; Cheng et al. 2007; Zhao et al. 2007; Shang et al. 2008) applies data mining techniques as building blocks for extracting features, and the second kind (Shasha et al. 2002; He and Singh 2006; Williams et al. 2007; Jiang et al. 2007; Zou et al. 2008) uses other strategy to construct a feature set. However, most of these methods target subgraph query, and they are either inapplicable to or inefficient for the supergraph query. In addition, although Closure-tree (He and Singh 2006) 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 a 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 such as 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 (Fortin 1996; Bunke 2000; Conte et al. 2004). For subgraph relationship decision, the Ullmann's algorithm (Ullmann 1976) 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 (Cordella et al. 2004), 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, Messmer and Bunke (1999) builds a decision tree in the preprocessing phase and results in a quadratic time w.r.t. 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 (Messmer and Bunke 2000) results in a time sublinear w.r.t. the number of the graphs in a database. However, owing to the underlying decomposition strategy, the output of the method are all subgraph isomorphisms from each graph in the answer set to the query graph, which is unnecessary and time-consuming for the supergraph query. In the XML context, Gupta and Suciu (2003) constructs a single deterministic push down automata to generalize and improve tree pattern matching technique (not for arbitrary in structure graphs) for the specific task of evaluating XPath queries. Bohannon et al. (2005) 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 (Chen et al. 2007) 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 Washio and Motoda (2003). There has been many methods proposed (Kuramochi and Karypis 2001; Yan and Han 2002;

Borgelt and Berthold 2002; Yan and Han 2003; Wang et al. 2004; Wörlein 2006; Zeng et al. 2007), which can efficiently obtain (closed) frequent (induced) subgraphs from a graph database. To decrease the number of frequent subgraphs in a parameterized way, graph patterns summarization was proposed in Liu et al. (2008). They play an important role in the preprocessing phase in the paper.

6 Conclusions

In this paper, in order to efficiently answer supergraph queries, a novel compact organization of a graph database, GPTree, was proposed. Adopting the filtering-and-verification methodology, two methods for feature generation were presented. 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 on-line 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 method outperforms the existing counterpart method by one to two orders of magnitude.

Acknowledgements This work was supported in part by the National Grand Fundamental Research 973 Program of China (Grand No. 006CB303000), the Key Program of National Natural Science Foundation of China (Grant No. 60533110), and the NSF of China (Grant No. 60773063).

References

Agrafiotis DK, Bandyopadhyay D, Wegner JK, van Vlijmen H (2007) Recent advances in chemoinformatics. J Chem Inf Model 47(4):1279–1293

Bohannon P, Fan W, Flaster M, Narayan PPS (2005) Information preserving XML schema embedding. In: Proceedings of the international conference on very large data bases, pp 85–96

Borgelt C, Berthold MR (2002) Mining molecular fragments: finding relevant substructures of molecules. In: Proceedings of the IEEE international conference on data mining, pp 51–58

Bunke H (2000) Graph matching: Theoretical foundations, algorithms, and applications. In: Vision interface, pp 82–88

Burge M, Kropatsch WG (1999) A minimal line property preserving representation of line images. Computing 62(4):355–368

Cai D, Shao Z, He X, Yan X, Han J (2005) Community mining from multi-relational networks. In: Proceedings of European conference on principles and practice of knowledge discovery in databases, pp 445–452

Chen C, Yan X, Yu PS, Han J, Zhang D-Q, Gu X (2007) Towards graph containment search and indexing. In: Proceedings of the international conference on very large data bases, pp 926–937

Cheng J, Ke Y, Ng W, Lu A (2007) Fg-index: towards verification-free query processing on graph databases. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 857–872

Conte D, Foggia P, Sansone C, Vento M (2004) Thirty years of graph matching in pattern recognition. Int J Pattern Recognit Artif Intell 18(3):265–298

Cordella LP, Foggia P, Sansone C, Vento M (2000) Fast graph matching for detecting cad image components. In: Proceedings of the international conference on pattern recognition, pp 6034–6037

Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell 26(10):1367–1372

Fortin S (1996) The graph isomorphism problem. Technical report, University of Alberta

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York. ISBN 0-7167-1044-7

Gupta AK, Suciu D (2003) Stream processing of xpath queries with predicates. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 419–430

He H, Singh AK (2006) Closure-tree: an index structure for graph queries. In: Proceedings of the international conference on data engineering, p 38

Jiang H, Wang H, Yu PS, Zhou S (2007) Gstring: a novel approach for efficient search in graph databases. In: Proceedings of the international conference on data engineering, pp 566–575

Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: Proceedings of the IEEE international conference on data mining, pp 313–320

Li X-Y, Wan P-J, Wang Y, Yi C-W (2003) Fault tolerant deployment and topology control in wireless networks. In: Proceedings of the ACM international symposium on mobile ad hoc networking and computing, pp 117–128

Liu Y, Li J, Gao H (2008) Summarizing graph patterns. In: Proceedings of the international conference on data engineering, pp 903–912

Messmer BT, Bunke H (1999) A decision tree approach to graph and subgraph isomorphism detection. Pattern Recognit 32(12):1979–1998

Messmer BT, Bunke H (2000) Efficient subgraph isomorphism detection: a decomposition approach. IEEE Trans Knowl Data Eng 12(2):307–323

Petrakis EGM, Faloutsos C (1997) Similarity searching in medical image databases. IEEE Trans Knowl Data Eng 9(3):435–447

Shang H, Zhang Y, Lin X, Yu JX (2008) Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proc VLDB Endow 1(1):364–375

Shasha D, Wang JT-L, Giugno R (2002) Algorithmics and applications of tree and graph searching. In: Proceedings of the ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, pp 39–52

Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM 23(1):31–42

Wang C, Wang W, Pei J, Zhu Y, Shi B (2004) Scalable mining of large disk-based graph databases. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, pp 316–325

Washio T, Motoda H (2003) State of the art of graph-based data mining. SIGKDD Explor 5(1):59–68

Willett P, Barnard JM, Downs GM (1998) Chemical similarity searching. J Chem Inf Comput Sci 38(6):983–996

Williams DW, Huan J, Wang W (2007) Graph database indexing using structured graph decomposition. In: Proceedings of the international conference on data engineering, pp 976–985

Wörlein M (2006) Extension and parallelization of a graph-mining-algorithm. Master's thesis, Friedrich-Alexander-Universität, Erlangen-Nürnberg

Yan X, Han J (2002) gspan: Graph-based substructure pattern mining. In: Proceedings of the IEEE international conference on data mining, pp 721–724

Yan X, Han J (2003) Closegraph: mining closed frequent graph patterns. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, pp 286–295

Yan X, Yu PS, Han J (2005) Graph indexing based on discriminative frequent structure analysis. ACM Trans Database Syst 30(4):960–993

Zeng Z, Wang J, Zhou L, Karypis G (2007) Out-of-core coherent closed quasi-clique mining from large dense graph databases. ACM Trans Database Syst 32(2):13

Zhang S, Hu M, Yang J (2007) Treepi: a novel graph indexing method. In: Proceedings of the international conference on data engineering, pp 966–975

Zhao P, Yu JX, Yu PS (2007) Graph indexing: Tree $+$ delta $\geq$ graph. In: Proceedings of the international conference on very large data bases, pp 938–949

Zou L, Chen L, Yu JX, Lu Y (2008) A novel spectral coding in a large graph database. In: Proceedings of the international conference on extending database technology, pp 181–192