PrefIndex: An Efficient Supergraph Containment Search Technique

Gaoping Zhu, Xuemin Lin, Wenjie Zhang, Wei Wang, and Haichuan Shang

The University of New South Wales, Sydney, NSW, Australia
{gzhu,lxue,zhangw,weiw,shangh}@cse.unsw.edu.au

Abstract. Graphs are prevailingly used in many applications to model complex data structures. In this paper, we study the problem of supergraph containment search. To avoid the NP-complete subgraph isomorphism test, most existing works follow the *filtering-verification* framework and select graph-features to build effective indexes, which filter false results (graphs) before conducting the costly verification. However, searching features multiple times in the query graphs yields huge redundant computation, which leads to the emergence of the *computation-sharing* framework. This paper follows the roadmap of computation-sharing framework to efficiently process supergraph containment queries. Firstly, database graphs are clustered into disjoint groups for sharing the computation cost within each group. While it is shown NP-hard to maximize the computation-sharing benefits of a clustering, efficient algorithm is developed to approximate the optimal solution with an approximation factor of $\frac{1}{2}$. A novel prefix-sharing indexing technique, PrefIndex, is then proposed based on which efficient query processing algorithm integrating both filtering and verification is developed. Finally, PrefIndex is enhanced with multi-level sharing and suffix-sharing to further avoid redundant computation. An extensive empirical study demonstrates the efficiency and scalability of our techniques which achieve orders of magnitudes of speed-up against the state-of-the-art techniques.

1 Introduction

Recently, graph structured data have been increasingly adopted in applications such as Bio-informatics, Chemistry, Social Networks, WWW, etc. For instance, graphs are used to model protein interaction networks and chemical compounds in Bio-informatics and Chemistry, respectively. Efficient query processing is thus strongly demanded by graph database.

Graph containment search is defined as *supergraph containment search* [2] and *subgraph containment search* [12]. Given a query graph q and a graph database $D = \{g_1, ..., g_n\}$, supergraph containment search finds all the graphs in D contained by q, while subgraph containment search finds all the graphs in D containing q. In Chemistry, given a newly found molecule (query graph) and a large number of descriptors (data graphs indicating chemical properties), we can predict its chemical function based on the descriptors it contains. In pattern recognition, given a graph structured background (query graph) and various

M. Gertz and B. Ludäscher (Eds.): SSDBM 2010, LNCS 6187, pp. 360–378, 2010.

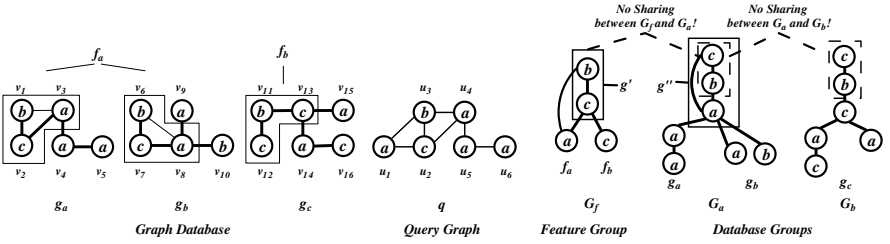


Fig. 1. Supergraph Containment Search

objects (data graphs), we may detect the foreground objects contained in the background. More applications can be found in [2, 12]. Regarding Figure 1, the result of supergraph containment search of query graph q is g_a.

Since the subgraph isomorphism test is NP-complete [4], most works adopt the *filtering and verification* framework. While a feature-based index filters most false results in the filtering phase, survived candidate graphs are checked in the verification phase. Unlike its extensively studied dual version [3, 5–7, 9, 10, 12, 13, 16, 17], supergraph containment search is comprehensively investigated in only two studies [2, 14]. cIndex, proposed in [2], adopts historical *query-log* to select features for maximizing pruning power. Regarding Figure 1, assume that f_a and f_b are two features. cIndex first tests if q contains f_a and f_b. As f_b is not contained by q, g_c is filtered; while g_a and g_b survived to be candidates as f_a is contained by q. As f_a is a subgraph of g_a and g_b, it will be searched for two more times in q for the verifications of g_a and g_b. Generally speaking, if a feature f is contained by n candidates, the subgraph isomorphism test on f against q will be repeated $(n + 1)$ times (including one test in the filtering phase).

To avoid redundant subgraph isomorphism test cost, [14] proposes GPTree, a computation-sharing framework. It encodes each graph or feature in a sequence called GVCode. The GVCodes of a group of graphs or features are organized in a tree called GPTree such that a common subgraph of these graphs or features is stored only once as a prefix of the tree. This finally yields a forest structured database or index. GPTreeTest, the proposed subgraph isomorphism algorithm, verifies a group of graphs or features altogether by sharing the computation (subgraph isomorphism test) cost of the common subgraph within the group. For instance, in Figure 1, one feature group G_f (for f_a, f_b), two database groups G_a (for g_a, g_b) and G_b (for g_c) are built. The computation cost of the common subgraphs g' and g'' are shared within G_f and G_a, respectively. However, with further observations, GPTree has the following defects: (1)Computation cost can not be shared between filtering and verification, as the common edge $b - c$ of G_f and G_a can not be shared between them; (2)Computation cost can not be shared between database groups, as the common edge $b - c$ of G_a and G_b can not be shared between them; (3)GPTree prefers to select large-sized common subgraphs for sharing, which goes against the fact that large-sized subgraphs are usually infrequent and not likely to be shared by many graphs.

Motivated by the above observations, this paper proposes a novel computation-sharing framework with the aim to maximize the computation-sharing benefits. The main contributions of this paper are summarized as follows.

- We propose to cluster database graphs into disjoint groups such that graphs in each group contain a common feature f. While it is shown NP-hard to optimize the computation-sharing benefits, an efficient greedy algorithm is used to approximate the optimal solution with an approximation factor of $\frac{1}{2}$.
- Based on QuickSI traversal technique [9], a novel prefix-sharing indexing technique PrefIndex and a query processing algorithm PrefSearch are developed to share computation cost between filtering and verification.
- A group encoding technique is proposed to efficiently construct PrefIndex for a group of graphs based on the effective ordering of their GVCodes.
- Multi-level sharing and suffix-sharing techniques are proposed to enhance PrefIndex for sharing computation cost among database groups and further sharing computation within each database group, respectively.

Organization. The rest of the paper is organized as follows. Section 2 gives preliminaries and formalizes the problem. Section 3 presents the database clustering, feature selection and index construction techniques. Section 4 proposes our sharing-aware query processing algorithm integrating filtering and verification. Section 5 presents our multi-level sharing and suffix-sharing techniques. Experimental results and related work are reported in Section 6 and 7, while Section 8 concludes our study.

2 Preliminaries

2.1 Problem Statement

For presentation simplicity, our study only focuses on *simple, vertex-labeled* graphs. A *simple* graph is an *undirected* graph with no self-loops nor multiple edges between any two different vertices. From now on, a database graph is called a *data graph*, while a query graph is called a *query*. All data graphs are assumed to be *connected*. Nevertheless, our approach can be immediately extended to *directed* or *edge-labeled* graphs.

Given two sets of labels, Σ_V and Σ_E, a graph g is defined as a triplet $(V(g), E(g), l)$ where $V(g)$ and $E(g)$ denote the vertex set and edge set of g, respectively. l is a mapping: $V(g) \rightarrow \Sigma_V$ which assigns a label $l(u)$ to each vertex $u \in V(g)$.

Definition 1 (Subgraph Isomorphic). *Given two graphs $g = (V, E, l)$ and $g' = (V', E', l')$, g is subgraph isomorphic to g', denoted by $g \subseteq g'$, if there is an injective function $f : V \rightarrow V'$ such that (1) $\forall v \in V$, $f(v) \in V'$ and $l(v) = l'(f(v))$; (2) $\forall (u, v) \in E$, $(f(u), f(v)) \in E'$ and $l(u, v) = l'(f(u), f(v))$. Under the above conditions, g (g') is a subgraph (supergraph) of g' (g).*

Definition 2 (Induced Subgraph). *Given a graph g, a graph g' is an induced subgraph of g, if and only if (1) g' is subgraph isomorphic to g under an injective function f; (2) $\forall u, v \in V(g')$, if $(f(u), f(v)) \in E(g)$, $(u, v) \in E(g')$.*

Definition 3 (Supergraph Containment Search). *Given a graph database* $D = \{g_1, g_2, ..., g_n\}$ *and a query graph* q*, find the answer set* D_q *which consists of each* $g_i \in D$ *such that* $g_i \subseteq q$*.*

2.2 Computation-Sharing Framework

cIndex [2] is the first filtering-verification framework for supergraph containment search. It applies the *exclusive logic* to filter data graphs; namely, if a feature $f \not\subseteq q$, any data graph g such that $f \subseteq g$ can be filtered. However, sequentially testing each feature (candidate graph) against the query involves huge redundant computation cost in the filtering (verification) phase.

GPTree [14], the first computation-sharing framework, directly extends the filtering-verification framework to avoid the redundant computation cost in cIndex. Inspired by DFS code [11] and QISequence [9], GPTree proposes a new graph encoding technique called GVCode. Based on a spanning tree t of a graph g, it encodes g into a sequence represented by a regular expression $Code_g = [[S_i E_{ij}^*]^{V(g)}]$. Each entry S_i is the mapped image of a vertex v in g. While $S_i.l$ keeps the label of v, $S_i.p$ stores the entry corresponding to the parent vertex of v in t. Once S_1 is fixed, t is viewed as a tree rooted at the vertex corresponding to S_1 and thus $S_1.p = 0$. Each edge in g but not in t is recorded as a back edge. If S_i has back edges, they are kept in $\{E_{ij}\}$.

Table 1 gives two GVCodes of g_a and g_b in Figure 1. The corresponding vertex of S_i is in the bracket. The bold lines in Figure 1 show the spanning trees of g_a and g_b. As g'' in Figure 1 is represented as a three-entry prefix from S_1 to S_3 in both $Code_a$ and $Code_b$, a tree structured organization of $Code_a$ and $Code_b$, called GPTree, can be built to share the three-entry prefix as a common prefix. Note that a common prefix must be an induced subgraph of all graphs in the group and the GVCode of a graph is not unique.

Based on QuickSI [9], GPTreeTest, a new subgraph isomorphism test algorithm is proposed to verify a group of graphs sharing a common prefix. Regarding Table 1, GPTreeTest first searches in q a subgraph isomorphism mapping of the common prefix from S_1 to S_3. A found mapping is then extended in a depth first fashion to search a whole mapping for the rest $Code_a$ and $Code_b$ under the common prefix, respectively.

Table 1. The GVCodes of g_a and g_b

$Type$	$(S_i.l, S_i.p)$
$S_1(v_2)$	$(c, 0)$
$S_2(v_1)$	(b, S_1)
$S_3(v_3)$	(a, S_2)
E_{31}	$[edge : S_3]$
$S_4(v_4)$	(a, S_3)
$S_5(v_5)$	(a, S_4)

$Code_a$

$Type$	$(S_i.l, S_i.p)$
$S_1(v_7)$	$(c, 0)$
$S_2(v_6)$	(b, S_1)
$S_3(v_8)$	(a, S_2)
E_{31}	$[edge : S_3]$
$S_4(v_{10})$	(b, S_1)
$S_5(v_9)$	(a, S_1)

$Code_b$

The framework of GPTree can be outlined by four phases: (1) Mine frequent subgraphs from the database and build a feature-based index $\mathcal{F} = \{f_1, ..., f_n\}$. Each f_i is attached a graph-ID list $list_{f_i} = \{g.id | f_i \subseteq g \wedge g \subseteq D\}$. (2) Mine common induced subgraphs from features and data graphs respectively. Greedily select the largest common induced subgraphs and divide features and data graphs into disjoint groups such that each group G shares a common induced subgraph g'. Graphs in each group G are encoded to share $Code_{g'}$ as a common prefix, based on which a GPTree is built for G. (3) For filtering, each feature group is tested by GPTreeTest to obtain the candidate set $C_q = D - \bigcup_f list_f (f \not\subseteq q \wedge f \in \mathcal{F})$. (4) For verification, each database group is projected on C_q and all non-empty projected database groups are verified by GPTreeTest to obtain the answer set D_q.

3 PrefIndex

Our Framework. We propose a novel computation-sharing framework called PrefIndex, which directly selects a feature as the common subgraph shared by a group of data graphs. Since a feature is encoded as a common prefix of all the data graphs in the group, its test cost can be shared between filtering and verification. The whole framework is outlined as follows.

1. Mine frequent induced subgraphs from the database and cluster all data graphs into disjoint groups $\{(f_i, G_i) | 1 \leq i \leq k\}$ such that for each graph g in a group G_i, f_i is an induced subgraph of g and a selected feature.
2. Encode each graph g in a group G_i into $Code_g$ of which $Code_{f_i}$ is a prefix. Organize all GVCodes of graphs G_i into a PrefIndex tree.
3. Apply our query processing algorithm integrating filtering and verification to process each group G_i by sharing the computation cost of $Code_{f_i}$.

3.1 Cost Model and Feature Selection

Given a data graph g and a query q, if $f \subseteq g$, the subgraph isomorphism test cost of g may be approximately represented by $cost_f + cost_{(g-f)}$. Given a group G_i of graphs sharing a common induced subgraph f_i, if we test all graphs in G_i by sharing the test cost of f_i (to process f_i only once), the cost gain (computation-sharing benefits) approximately equals (1). Assume that no pre-knowledge is given on q, (1) may be interpreted as the expected gain for any q.

Given a database D and a set $\mathcal{D} = \{(f_i, D_i) | 1 \leq i \leq m\}$ where each D_i contains all graphs in D which share f_i as a common induced subgraphs, we call f_i and D_i *master feature* and *master group* of f_i, respectively.

Definition 4 (Disjoint Database Cover). *Given a set $\mathcal{D} = \{(f_i, D_i) | 1 \leq i \leq m\}$ of master features and master groups and assume that $\bigcup_i D_i = D$, $\mathcal{G} = \{(f'_j, G_j) | 1 \leq j \leq k\}$ is a disjoint cover of D conforming $\mathcal{D}$ if and only if (1) $\forall (f'_i, G_i)$, $\exists (f_j, D_j) \in \mathcal{D}$ such that $f'_i = f_j$ and $G_i \subseteq D_j$; (2) $\forall G_j \neq G'_j, G_j \bigcap G'_j = \emptyset$; (3) $\forall g \in D$, $\exists G_j \in \mathcal{G}$ such that $g \in G_j$.*

Given a disjoint cover $\mathcal{G}$ of D, the total cost gain by sharing the test cost of the master feature within each master group is in (2).

$$gain_{G_i} = cost_{G_i} - cost'_{G_i} = (|G_i| - 1) \times cost_{f_i} \quad (1)$$

$$gain_{\mathcal{G}} = \sum_{i=1}^{k} (|G_i| - 1) cost_{f_i} \quad (2)$$

Definition 5 (Maximized Gain (MG)). *Given a database D and a set $\mathcal{D} = \{(f_i, D_i) | 1 \leq i \leq m\}$ of master features and master groups such that $\forall g \in D$, $\exists D_i \in \mathcal{D}$, $g \in D_i$, find a disjoint cover $\mathcal{G}$ of D such that $gain_{\mathcal{G}}$ is maximized.*

Theorem 1. *The problem of Maximized Gain is NP-hard.*

Proof. In a special case of MG where each $cost_{f_i}$ equals a constant c, $gain_{\mathcal{G}} = c \times (n - k)$. Consequently, solutions of MG in this case aim to minimize k, which makes MG exactly a *minimum set cover* problem (NP-hard) [4].

Assume that each data graph contains at least one feature. Algorithm 1 is adopted to approximate the optimal disjoint cover with the maximum $gain_{\mathcal{G}}$. Let $\mathcal{G}$ be a set of already selected disjoint groups ($\mathcal{G}$ is empty at the beginning). Let $g(\mathcal{G})$ be all the data graphs currently covered by $\mathcal{G}$. For each D_i with a master feature f_i, let $\frac{cost_{f_i} \times (|D_i - g(\mathcal{G})| - 1)}{|D_i - g(\mathcal{G})|}$ be the average gain from each remaining data graph in $D_i - g(\mathcal{G})$. The greedy algorithm iteratively selects a group $(f_i, D_i - g(\mathcal{G}))$ with the highest average gain, until $\mathcal{G}$ covers all the data graphs.

Algorithm 1. ***Clustering***

Input . D: a graph database $\{g_1, ..., g_n\}$;
$\mathcal{D}$: a set of features and master groups $\{(f_1, D_1)..., (f_m, D_m)\}$;
Output . $\mathcal{G}$: a disjoint cover of D;

```
1 G := ∅; S := D;;
2 while D − g(G) ≠ ∅ do
3     Select (f_i, D_i) ∈ S with the maximum (cost_{f_i} × (|D_i − g(G)| − 1)) / |D_i − g(G)|;
4     Insert (f_i, D_i − g(G)) to G;
5     S := S − {(f_i, D_i)};
6 return G;
```

Example 1. Given $D = \{g_1, ..., g_5\}$ in Figure 2, for each f_i, its master group D_i and cost $cost_{f_i}$ are in the left two tables, respectively. Consider clustering (a) of PrefIndex by Algorithm 1. For the first iteration, the average gains from D_1 to D_4 are 2, 2, $\frac{5}{2}$ and $\frac{8}{3}$, respectively. Consequently, $(f_4, \{g_1, g_4, g_5\})$ is selected. Then g_1, g_4 are removed from D_1, while g_5 is removed from D_3. The average gains of D_1 and D_3 become 0, while the average gain of D_2 remains unchanged. For the second iteration, $(f_2, \{g_2, g_3\})$ is selected. Finally, we obtain a disjoint cover $\mathcal{G} = \{(f_4, \{g_1, g_4, g_5\}), (f_2, \{g_2, g_3\})\}$ with a total gain of 12.

Time Complexity. Due to the greedy nature of Algorithm 1, the worst case time complexity is $O(n^2 \times m)$ where n and m are the numbers of data graphs and features, respectively.

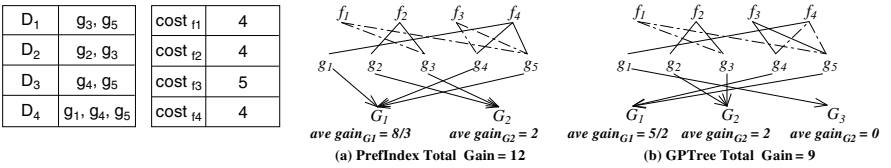


Fig. 2. Example of Database Clustering

Accuracy Guarantee. Let OPT and A denote the total gains of the optimal solution and Algorithm 1, respectively. The following theorem can be proved.

Theorem 2. $A \geq \frac{1}{2}OPT$.

Proof. Let $\mathcal{G} = \{(f_i, G_i)|1 \leq i \leq l\}$ be the optimal solution of MG and OPT_{G_i} be the gain of group G_i; the total gain OPT of $\mathcal{G}$ is (3). Let the number of graphs in G_i be n_{G_i}; the average gain a_{G_i} of G_i is (4).

$$OPT = \sum_{i=1}^{l} cost_{f_i} \times (|G_i| - 1) = \sum_{i=1}^{l} OPT_{G_i} \tag{3}$$

$$a_{G_i} = \frac{cost_{f_i} \times (n_{G_i} - 1)}{n_{G_i}} \tag{4}$$

Since the gain of any group with only one graph is 0, we only consider those groups with at least two graphs. Let $\mathcal{G}' = \{(f'_j, G'_j)|1 \leq j \leq l'\}$ be the greedy solution generated by Algorithm 1 and A be the total gain of $\mathcal{G}'$. Assume G'_j is selected at the jth iteration. By removing any graph $g \notin G'_j$ from G_i, we obtain $G_i^j = G_i \bigcap G'_j (1 \leq j \leq l')$. By removing any group $G_i^j = \emptyset$, G_i can be partitioned into a set of disjoint subgroups $\{G_i^{j_1}, ..., G_i^{j_k}\}$ such that (1)$\bigcup_{t=1}^{k} G_i^{j_t} = G_i$; (2)$\forall t, t' (1 \leq t < t' \leq k), j_t < j_{t'}$.

Let the average gain of $G_i^{j_t}$ in the greedy algorithm be a'_t and the number of graphs in $G_i^{j_t}$ be n'_t ($\sum_{t=1}^{k} n'_t = n_{G_i}$). Two observations can be made based on the partition of G_i. (1) Only the last subgroup $G_i^{j_k}$ may have a gain of 0 in the greedy solution, since G'_{j_k} may have only one graph and $G_i^{j_k}$ is not empty. If another group $G'_{j_t} (j_t < j_k)$ has only one graph and $G_i^{j_t}$ is not empty, a new subgroup $G_i^{j_k} \bigcup G_i^{j_t}$ can be obtained to yield a gain greater than 0. (2) Due to the greedy nature of Algorithm 1, a'_1, the average gain of the first subgroup $G_i^{j_1}$ in the greedy solution, must be no less than a_{G_i}, Otherwise, G_i instead of G'_{j_1} will be selected for the j_1th iteration in the greedy solution. It can be concluded that the gain of $G_i^{j_1} \bigcup G_i^{j_k}$ in the greedy solution is at least $\frac{1}{2}$ of that in the optimal solution. For each rest subgroups $G_i^{j_t}$ $(t \neq 1, k)$, due to the greedy nature of Algorithm 1, (5) can be immediately verified. By replacing $cost_{f_i}$ in (5) with (4), we have (6). Let the gain of G_i in the greedy solution be A_{G_i}. By (6), we have (7), which leads us to our conclusion that $A \geq \frac{1}{2}OPT$.

$$a'_{t'} \geq \frac{cost_{f_i} \times (n_{G_i} - \sum_{t=1}^{t'-1} n_t - 1)}{n_{G_i} - \sum_{t=1}^{t'-1} n_t} \tag{5}$$

$$a'_{t'} \geq \frac{n_{G_i} \times (n_{G_i} - \sum_{t=1}^{t'-1} n_{t'} - 1)}{(n_{G_i} - 1) \times \sum_{t=1}^{t'-1} n_t} \times a_{G_i} \geq \frac{1}{2} \times a_{G_i} \quad (6)$$

$$A_{G_i} = \sum_{t=1}^{k} a'_t \times n_t \geq \sum_{t=1}^{k} \frac{a_{G_i} \times n_t}{2} = \frac{1}{2} OPT_{G_i} \quad (7)$$

Remark. Generally speaking, the subgraph isomorphism test runs in an exponential time in the worst case and is algorithm, graph topology and graph size dependent. While our algorithm and its analysis apply to any given cost formula, $|V(f)|$ is used to approximate $cost_f$ in our implementation.

3.2 Computation-Sharing Comparison

On computation-sharing strategy, PrefIndex differs from GPTree in two ways. Firstly, GPTree selects common induced subgraphs for features and data graphs respectively, while PrefIndex directly selects common induced subgraphs of data graphs as features and shares the computation between filtering and verification. Secondly, GPTree greedily selects common induced subgraphs with the highest cost (largest size). Since larger subgraphs are usually unlikely to be contained by many graphs, PrefIndex uses a more natural heuristic to greedily select features with the highest average gain.

Consider clustering (b) of GPTree in Figure 2, though f_3 has the highest cost, it contributes less than f_4 as its master group has less data graphs, which only leads to a total gain of 8. In our experiments, PrefIndex outperforms GPTree in all cases on computation-sharing benefits.

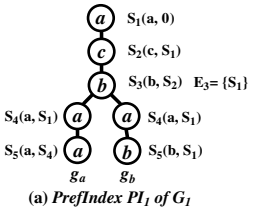
3.3 Index Structure

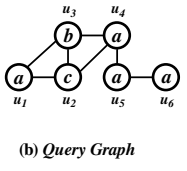
In PrefIndex, each feature f is encoded into a common prefix of the GVCode of each graph g in its master group G_f. This requires that f must be an induced subgraph of each g in G_f. We extend gSpan [11] to mine frequent, discriminative, induced subgraphs. $\mathcal{D}$ is initialized with the mined induced subgraphs and their master groups and then fed into Algorithm 1 for feature selection. Although a feature f may be contained by a data graph g not in G_f as a non-induced subgraph, such information is not recorded in PrefIndex due to mining efficiency.

The index structure of PrefIndex is outlined as follows: (1) Given a disjoint cover $\mathcal{G} = \{(f_i, G_i) | 1 \leq i \leq k\}$ generated by Algorithm 1, encode each g in a group G_i into $Code_g$ by having $Code_{f_i}$ as its prefix. In practice, QuickSI algorithm [9] is used to efficiently identify subgraph isomorphism mappings from f_i to g. (2) Each entry S_i of $Code_g$ only stores the label of its corresponding vertex in g, its parent vertex in $Code_g$ and its back edge information. Figure 3(a) shows a PrefIndex of graphs g_a and g_b in Figure 1. The effective ordering of common prefixes and suffixes is discussed in the next section.

4 PrefIndex Search

This section firstly presents our querying algorithm for a group of graphs based on PrefIndex and then proposes two techniques to further enhance PrefIndex: (1)





$Type$	$(S_i.l, S_i.p)$
$S_1(v_3)$	$(a, 0)$
$S_2(v_2)$	(c, S_1)
$S_3(v_1)$	(b, S_2)
E_{31}	$[edge : S_1]$
$S_4(v_4)$	(a, S_1)
$S_5(v_5)$	(a, S_4)

$Code_a$

$Type$	$(S_i.l, S_i.p)$
$S_1(v_8)$	$(a, 0)$
$S_2(v_7)$	(c, S_1)
$S_3(v_6)$	(b, S_2)
E_{31}	$[edge : S_1]$
$S_4(v_9)$	(a, S_1)
$S_5(v_{10})$	(b, S_1)

$Code_b$

Fig. 3. PrefIndex

Ordering GVCodes efficiently for PrefIndex; (2) Sharing pruning power among master groups. Given a disjoint cover $\mathcal{G} = \{(f_i, G_i)|1 \leq i \leq k\}$, the PrefIndex of G_i is denoted by $PI_i = \{Code_g|g \in G_i\}$. We aim to share the test cost of $Code_{f_i}$ between filtering and verification. $Code_{f_i}$ is first processed to check whether a subgraph isomorphism mapping exists from f_i to q. The test of $Code_{f_i}$ is enforced to be conducted only once for G_i, which leads to the following fundamental theorem. The trivial proof is omitted here.

Theorem 3. *If a query q contains a data graph g, for each prefix $Code'$ of $Code_g$, there must be a subgraph isomorphism mapping from $Code'$ to g.*

4.1 Algorithm

Algorithm Sketch. Based on Theorem 3, for each group G_i, our algorithm probes PI_i in a depth first fashion. For each $Code_g$ in PI_i, once a mapping $\mathcal{P}'$ is found from a prefix $Code'$ of $Code_g$ to q, the algorithm checks if $\mathcal{P}'$ can be extended to cover the next vertex in $Code_g$. If impossible, it backtracks in PI_i to search the next mapping from $Code'$ to q. Since G_i may contain many graphs, the last vertex of $Code_{f_i}$ may link to many suffix branches. The algorithm backtracks from the last vertex of $Code_{f_i}$, if all suffix branches under this vertex have been explored in a depth first fashion. The algorithm terminates when no new mapping can be found for the first vertex of $Code_{f_i}$ or all GVCodes in PI_i are detected to be subgraph isomorphic to q. The algorithm consists of two parts: PrefixQ and SuffixQ in Algorithm 2 and 3.

Example 2. Regarding the example in Figure 1, assume the database is clustered based on disjoint cover $\mathcal{G} = \{(f_a, G_1 = \{g_a, g_b\}), (f_b, G_2 = \{g_c\})\}$. In Figure 3, the two tables show the GVCodes of g_a and g_b, while Figure 3 (a) and (b) show the PrefIndex of G_1 and the query.

In our algorithm, S_1 is first mapped to u_1 in q as their labels match. S_2 is then mapped to u_2 as their labels and parents match; similarly, S_3 is mapped to u_3 as their labels, parents and back edges match ($S_3.l = b$, $S_3.p = S_2$ and the back edge (S_3, S_1)). An intermediate mapping $\mathcal{P}_1$ from $Code_{f_a}$ to q is found. This corresponds to the filtering phase.

In verification phase, $\mathcal{P}_1$ is respectively extended for the suffixes of $Code_a$ and $Code_b$. For both $Code_a$ and $Code_b$, $\mathcal{P}_1$ fails to extend to S_4 as all adjacent vertices

of u_3 are either already mapped or unable to meet $S_4.p = S_1$. Consequently, $\mathcal{P}_1$ is abandoned and it backtracks from S_3 to search a new mapping. It then finds that no new mapping can be found for S_3 while keeping the existing mappings of S_1 and S_2. After it backtracks for one more depth, no new mapping can be found for S_2 while keeping the existing mapping of S_1. Finally, it starts from S_1 again and finds another intermediate mapping $\mathcal{P}_2$ from $\{S_1, S_2, S_3\}$ to $\{u_4, u_2\ u_3\}$. For $Code_a$, u_5 and u_6 are mapped to S_4 and S_5 respectively , which verifies g_a as an answer. For $Code_b$, the extension fails at S_5. Since $\mathcal{P}_2$ is the last mapping from $Code_a$ to q, the query processing of G_1 terminates.

Algorithm 2. ***PrefixQ***$(G, Code_f, \mathcal{P}, \mathcal{F}, q, d)$

```
Input  . G: a group of data graphs represented as GVCodes (PrefIndex);
         Code_f: the common prefix;
         P: a vector, initialized with ∅;
         F: a vector, initialized with 0;
         q: a query graph;
         d: the mapping depth;
Output . G_q: the answer set of q for G;
1  if d > |Code_f| then
2      for each g ∈ G do
3          if SuffixQ(Code_g, P, F, q, d) then
4              G_q := G_q ⋃ {g};
5
6      G := G − G_q;
7      if G = ∅ then
8          return G_q;
9
10 S := S_d(∈ Code_f); E := E_d(∈ Code_f);
11 if d = 1 then
12     V := {v | v ∈ V(q) ∧ l(v) = S.l ∧ F_v = 0};
13 else
14     V := {v | v ∈ V(q) ∧ l(v) = S.l ∧ (v, P_{S.p}) ∈ E(q) ∧ F_v = 0};
15 for each v ∈ V do
16     for each back edge e ∈ E do
17         goto line 13 if e ∉ E(q);
18     P_d := v; F_v := 1;
19     G_q := G_q ⋃ PrefixQ(G, Code_f, P, F, q, d + 1);
20     G := G − G_q;
21     if G = ∅ then
22         return G_q;
23     F_v := 0;
24 return G_q;
```

PrefixQ processes the common prefix $Code_f$ of a group G and calls SuffixQ to complete the searching. It is in a recursive, depth first search fashion presented in Algorithm 2. The output G_q is the answer set for G. $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ...\}$ stores

the vertex mappings from $Code_f$ to q. $\mathcal{P}_d = v_i$ means $S_d \in Code_f$ is mapped to v_i in q. $\mathcal{F} = \{\mathcal{F}_1, ..., \mathcal{F}_{|V(q)|}\}$ stores the vertex state for each v_i in q. $\mathcal{F}_i = 1$ means v_i is already mapped to a vertex in $Code_f$. The mapping depth d indicates the current vertex $S_d \in Code_f$ to be mapped. PrefixQ firstly checks if the current mapping $\mathcal{P}$ covers all the vertices of $Code_f$. Condition $d > |Code_f|$ (line 1) implies that a mapping $\mathcal{P}$ from $Code_f$ to q is found. SuffixQ (to process each suffix respectively) is then called (line 3) to extend $\mathcal{P}$. Once a mapping from $Code_g$ to q is successfully extended, g is moved from G to G_q. PrefixQ terminates when all GVCodes in G are detected subgraph isomorphic to q (line 7) or all mappings have been exhausted (line 13 and then line 22) for S_1 of $Code_f$.

SuffixQ processes a suffix under $Code_f$; namely, $Code_g - Code_f$. It has the same input and also follows a recursive, depth first search fashion. The correctness of Algorithm 2 and 3 is immediate from Theorem 3. Although costing exponential time in the worst case, they are very efficient in practice.

Algorithm 3. ***SuffixQ***$(Code_g, \mathcal{P}, \mathcal{F}, q, d)$

```
   Input  . Same as PrefixQ;
   Output . Boolean: Code_g is a subgraph of q;
 1 if d > |V(q)| then
 2 |  return True;
 3 S := S_d ∈ Code_g; E := E_d ∈ Code_g;
 4 V := {v|v ∈ V(g) ∧ l(v) = S.l ∧ (v, P_{S.p}) ∈ E(q) ∧ F_v = 0};
 5 for each v ∈ V do
 6 |  for each back edge e ∈ E do
 7 |  |  goto line 5 if e ∉ E(q);
 8 |  P_d := v; F_v := 1;
 9 |  if SuffixQ (Code_g, P, F, q, d + 1) then
10 |  |  return True;
11 |  F_v := 0;
12 return False;
```

4.2 Effectively Ordering GVCode

Given a graph g with m vertices, there are $m!$ different possible GVCodes. As shown in [9], a good ordering of a query q can determine earlier if a subgraph isomorphism mapping from q to a graph g exists. Thus the edges (labels) in q with lower occurrence rates should have a higher priority to be allocated earlier in $Code_q$ to reduce the number of intermediate mappings to be considered. As our problem is the dual problem of that in [9], edges (labels) in a data graph g with lower occurrence rates in the database are "signatures" of g and should be allocated earlier in $Code_g$ for early pruning. When constructing PrefIndex PI_i for a group G_i with a feature f_i, QISequence ordering technique is firstly applied on $Code_{f_i}$ and then on each suffix under $Code_{f_i}$ in the same way.

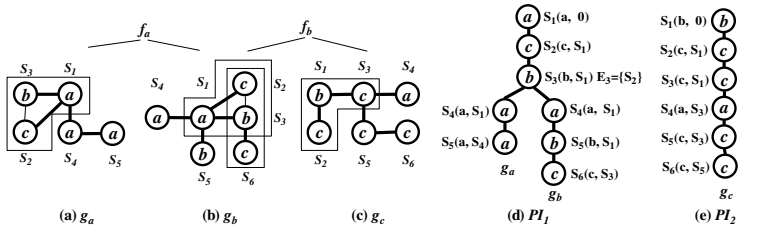


Fig. 4. Enhancing Pruning Power

4.3 Enhancing Pruning Power

In Figure 4, three graphs are clustered into two groups $G_1 = \{g_a, g_b\}$ with f_a and $G_2 = \{g_c\}$ with f_b. Since g_b contains f_b, g_b can also be pruned if f_b fails to pass the filtering phase. To share pruning power among different groups, two lists of graph IDs, M(Master)-List and R(Reference)-list are added at the last vertex of $Code_{f_i}$ in PI_i. M-list consists of IDs of the graphs in G_i, while R-List consists of IDs of the graphs not in G_i but containing f_i. Regarding G_2, $M\text{-}List_{G_2} = \{g_c\}$ and $R\text{-}List_{G_2} = \{g_b\}$. When PrefixQ detects that $f_b \not\subseteq q$, g_b can be pruned from G_1 by not invoking SuffixQ on g_b. To realize this, the first SuffixQ call in each G_i is enforced to happen only after PrefixQ calls on all groups are finished and a filtering list R is obtained. In each survived G_i, SuffixQ is only invoked on those graphs not in R.

PrefixQ can be immediately modified to accommodate the above requirements. If PrefixQ reaches the depth $|Code_f| + 1$ of G for the first time, SuffixQ is not invoked until the depth $|Code_{f'}| + 1$ of all other G' is either reached for the first time (survived groups) or detected impossible to reach (R-lists of pruned groups are added to R). The modified algorithm is presented in Algorithm 4.

Algorithm 4. ***PrefSearch (PS)***

Step 1. For each (f_i, G_i) $(1 \leq i \leq k)$, probe $Code_{f_i}$ with PrefixQ to check if a subgraph isomorphism mapping $\mathcal{P}_i$ exists from $Code_{f_i}$ to q.
If $\mathcal{P}_i$ is found for the first time, we add $(\mathcal{P}_i, PI_i)$ to the verification job list J; otherwise, we add the R-List of f_i to the filtering list R.

Step 2. For each $(\mathcal{P}_i, PI_i) \in J$, conduct verification on PI_i by invoking SuffixQ starting from the end of $Code_{f_i}$ in a depth first search against q, while ignoring those $Code_g$ whose IDs are in R.

5 Hierarchical PrefIndex Search

This section explores two further computation-sharing opportunities missed by GPTree: (1) sharing computation among common prefixes of multiple groups; (2) sharing computation among multiple suffixes in each group. We propose to organize the PrefIndexes of multiple groups in a hierarchical structure.

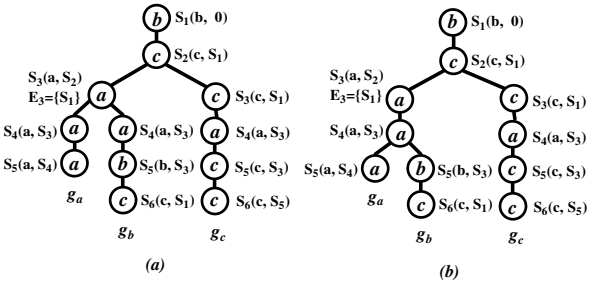


Fig. 5. Hierarchical PrefIndex

Multi-level Sharing. Regarding Figure 4, assume that (d) and (e) are the PrefIndexes of $G_1 = \{g_a, g_b\}$ and $G_2 = \{g_c\}$. Although f_a and f_b contain the same induced subgraph, an edge (b, c), the chance to share it between G_1 and G_2 is missed since it is not the prefix of PI_1, To address this, a level 2 PrefIndex can be obtained by applying PrefIndex on all selected master features. Generally, a level n PrefIndex can be obtained by applying PrefIndex on all sub-features of level $n-1$. The procedure can be iteratively performed until no new common subgraphs are identified at the current level.

Regarding Figures 4, we first mine frequent induced subgraphs from f_a and f_b as sub-features and then apply feature selection to construct a 2-level PrefIndex. Figure 5 (a) shows a 2-level PrefIndex where edge (b, c) (sub-feature) is shared.

Suffix-Sharing. In PrefixQ, SuffixQ is recursively invoked for each survived suffix. Note that multiple suffixes in a group are still likely to share common entries. As in Figure 5 (a), two entries $Code_a.S_4$ and $Code_b.S_4$ have identical information as well as the common prefix. By sharing them as a common prefix of these two suffixes, the resulted PrefIndex is organized in Figure 5 (b).

Greedy algorithm and PrefIndex technique are adopted to explore the common prefixes of suffixes within each group. All possible next vertices in all suffixes which connects to the common prefix are identified and classified into different types based on label, parent and back edge information. We greedily selecting the vertex contained by the most suffixes and encode the vertex as the next entry of GVCodes of these suffixes. The greedy selection terminates when all types are contained by only one suffix. Frequent induced subgraph mining is not used here since the subgraph graph corresponding to each suffix is not always connected.

Building HiPrefIndex. PrefIndex technique is firstly applied on data graphs to build the first level index. Then we iteratively build index on common sub-features identified on each next level. Finally, suffix-sharing is applied within each group to complete the hierarchical PrefIndex which is called HiPrefIndex. If the generated HiPrefIndex is forest structured, a dummy root, which links to the top of each PrefIndex, is inserted into HiPrefIndex.

Searching HiPrefIndex. The querying processing on HiPrefIndex starts from the (dummy) root of HiPrefIndex and probes HiPrefIndex in a depth first fashion.

PrefixQ and SuffixQ can be immediately modified to support query processing on HiPrefIndex. For space limits, the details are not presented here.

Space-Time Efficiency vs Pruning Power. A branch in HiPrefIndex may correspond to more than one data graph. Regarding Figure 5 (b), the left branch of (S_3, S_4) leads to g_a and g_b. Given the query in Figure 1, the prefix corresponding to f_b is not contained by q. Since we know g_b also contains f_b, in order to remove g_b in the filtering phase, we need to record graph IDs along each edge in HiPrefIndex to share pruning power. This increases not only the storage space but also the computation cost to check graph IDs on each edge. Thus we ignore such information and do not share pruning power in HiPrefIndex.

6 Performance Evaluation

We evaluate the performance of our techniques by comparing with GPTree. The following techniques are examined: (1) Indexing techniques PrefIndex and HiPrefIndex in Section 3 and 5. (2) Querying algorithms PrefSearch and HiPrefSearch in Section 4 and 5. (3) Querying algorithms GPTree(A) and GPTree(E) in [14]. GPTree(A) differs from GPTree(E) as it approximately mines frequent closed subgraphs to save index construction cost, which yields an (incomplete) feature set contained by the (complete) feature set of GPTree(E). We obtain the code of GPTree from its authors [14]. All algorithms are implemented in C++ and compiled by GNU GCC. Experiments are conducted on PCs with Intel Xeon 2.4GHz dual CPU and 4G memory under Debian Linux.

Datasets: AIDS and AIDS10K. Two real datsets are used. AIDS Antiviral dataset, denoted by AIDS, contains 43,905 graph structured chemical compounds. It is a popular benchmark for studying graph queries downloaded from Development Therapeutics Program. To compare with GPTree based on its experiment settings, a subset of AIDS with 10K graphs, denoted by AIDS10K, is downloaded from **http://www.xifengyan.net/software.htm**.

Database and Query Set. For fair comparison, we adopt the experiment settings in [14]. We mine frequent subgraphs from AIDS10K with frequency ranging from 0.5% to 10% and randomly select 10K graphs as the *default database*, while the *default query set* is exactly AIDS10K.

6.1 Efficiency on Real Dataset

The default database and query set is used to evaluate the efficiency of our techniques. The default query set is divided into 5 groups from Q_1 to Q_5. The answer set size of queries in Q_1 falls in $[0, 20)$, while those of the rest groups from Q_2 to Q_5 fall in $[20, 40)$, $[40, 60)$, $[60, 80)$ and $[80, \infty)$. Both techniques mine candidate features from the default database with a minimum frequency of 1%.

Table 2. Index Construction

	PrefIndex	HiPrefIndex	GPTree(A)	GPTree(E)
Index Construction (sec)	311.5	313.6	129.7	697.9
Index Size (# of Features)	268	268	1301	1278

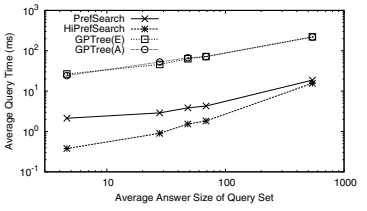


Fig. 6. Query Response

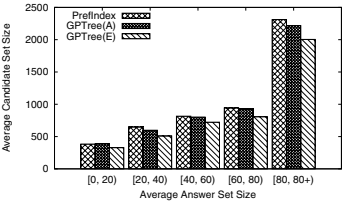


Fig. 7. Pruning Power

Query Processing. Figure 6 shows average query response time within each query group[1]. PrefSearch and HiPrefSearch outperform GPTree(A) and GPTree(E) for up to 2 orders of magnitudes in query processing, while GPTree(E) slightly outperforms GPTree(A). Although HiPrefSearch disables some of its pruning power, it outperforms other techniques due to its multi-level and suffix-sharing techniques.

Pruning Power. Figure 7 shows average pruning power measured by candidate size within each query group. The pruning power of PrefIndex is very similar to that of GPTree(E) and GPTree(A), while GPTree(E) outperforms GPTree(A) due to its complete frequent closed subgraph mining. Since HiPrefIndex disables a part of its pruning power to share more computation as discussed in Section 5, its pruning power is not evaluated here.

Index Construction. Table 2 shows index construction cost and index size measured by number of features. While most of the cost for both techniques is spent on the frequent subgraph mining, the effective ordering of GVCode of PrefIndex and HiPrefIndex only consumes less than 0.8% of the total cost. PrefIndex slightly outperforms HiPrefIndex due to the extra cost spent on mining multi-level subgraphs and common suffixes. GPTree(A) costs much less construction time as it approximately mine a small feature set. GPTree(E) costs the most index construction time since it mines not only a complete feature set but also common induced subgraphs from features and data graphs respectively. For fair comparison, we only focus on GPTree(E) for the rest experiments.

6.2 Scalability on Real Dataset

Varying Database Size. We first evaluate the scalability of our techniques by varying database size. For this reason, AIDS instead of AIDS10K is adopted to generate databases of various size. We first randomly select 10K graphs from

[1] The X-axis represents the average number of answer graphs in each group.

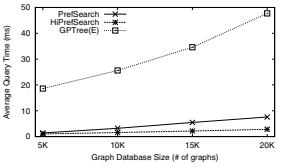


Fig. 8. Query Response

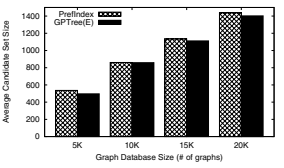


Fig. 9. Pruning Power

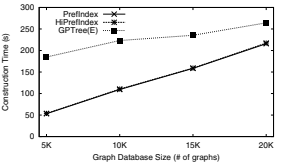


Fig. 10. Construction

AIDS as the query set from which we mine frequent subgraphs via the same way as in the overall performance and randomly select 5K, 10K, 15K and 20K frequent subgraphs to form 4 databases from D_1 to D_4.

Figure 8 shows average query response time for each database. The increment of query response time is almost linear for PrefSearch and HiPrefSearch with increasing database size. HiPrefSearch is still up to an order of magnitude faster than GPTree(E). Figure 9 shows average pruning power for each database. The pruning power of PrefIndex is similar to that of GPTree(E), which confirms the advantage of PrefSearch and HiPrefSearch over GPTree(E) mainly comes from the maximized computation-sharing benefits. Figure 10 shows index construction cost for each database. PrefIndex and HiPrefIndex always needs similar index construction time. It is because HiPrefIndex mines frequent subgraphs from a sequentially decreasing set of sub-features on each level, while common prefixes of suffixes are searched only within each group. GPTree(E) costs the most time due to the extra cost on mining common induced subgraphs.

Varying Data Graph Size. We then evaluate the scalability of our techniques by varying data graph size (in # of vertices). We randomly select 10K graphs from AIDS as the query set from which we mine frequent subgraphs via the same way as above and randomly select 5K frequent subgraphs of 10 vertices as database D_1. We construct other four database from D_2 to D_5 by selecting 5K frequent subgraphs of 12, 14, 16, and 18 vertices respectively.

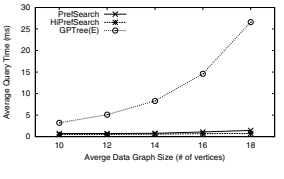


Fig. 11. Query Response

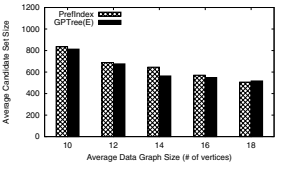


Fig. 12. Pruning Power

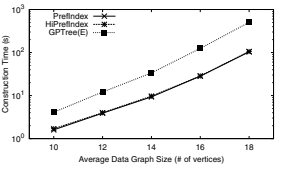


Fig. 13. Construction

The query response time, pruning power and index construction time for each database are respectively reported in Figures 11, 12 and 13. Note that the gap on query response time between PrefIndex and GPTree(E) are dramatically furthered with increasing data graph size, since the benefits of multi-level and suffix-sharing are more likely to be obtained on large graphs. While the pruning

power for both techniques remains similar, index construction cost of PrefIndex and HiPrefIndex are very close and increase less significantly than GPTree(E).

6.3 Scalability on Synthetic Dataset

We evaluate the scalability on synthetic dataset by varying database size. A graph generator from [3] is used. A default query set of 10K graphs is generated by setting the average graph size to 30 vertices, while the average density ($\frac{|V|}{|E|}$) is set to 1.3. The distinct number of labels is set to 10 and distributed uniformly. The default database and 4 databases of 5K, 10K, 15K and 20K graphs are constructed in the same way as above.

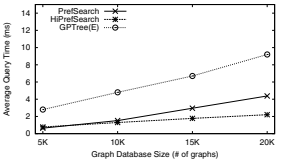


Fig. 14. Query Response

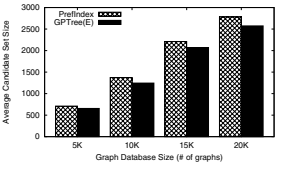


Fig. 15. Pruning Power

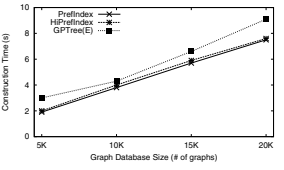


Fig. 16. Construction

The query response time, pruning power and index construction time are respectively recorded in Figure 14, 15 and 16. Although PrefSearch and HiPrefSearch outperform GPTree(E) on query response time and index construction cost, the gap between the techniques shortens a lot. Due the uniform distribution of vertex label, the number and size of frequent subgraphs greatly decrease and thus yields limited computation-sharing opportunity. However, PrefIndex still has its advantage over GPTree(E) as expected.

7 Related Work

Many studies have been done on graph containment search. While the subgraph containment search has been extensively studied, the supergraph containment search does not draw attentions from database community until most recently.

The filtering-verification framework is popular among most related work on subgraph containment search, which uses effective indexing techniques to filter most false results before the costly verification. Shasha et al. propose a path-based index, GraphGrep [5], which is known as the first feature-based index for subgraph containment search. To enhance the pruning power, frequent subgraph mining techniques such as gSpan [11] and F3TM [15] are developed. Yan et al. develop an effective indexing approach gIndex [12] based on frequent, discriminative subgraphs. Due to the expensive cost of frequent subgraph mining, Zhang et al. and Zhao et al. propose TreePI [13] and (Tree+Δ) [16] independently to index frequent subtrees. Cheng et al. propose a verification-free framework FG-Index [3] to further avoid subgraph isomorphism test. Besides the feature-based

index approaches, He et al. propose a clustering-based approach, called C-tree [6], to index graph closures (integration of graphs) in a B-tree like structure. It is the first work to support both exact and similarity subgraph containment search. Other clustering-based approaches include [8] and [1]. Moreover, Williams et al. [10] focus on the efficiency of processing small data graphs, while Jiang et al. [7] convert subgraph containment search to a string search problem. Recently, Shang et al. [9] present an efficient verification algorithm QuickSI.

On supergraph containment search, the first work cIndex proposed by Chen et al. [2] adopts historical query-log information to select features with maximized pruning power. Zhang et al. propose GPTree [14], a computation-sharing framework to share computation cost respectively in the filtering phase and verification phase. To address the defects of GPTree, our techniques propose efficient clustering and query processing algorithm to further share computation cost between filtering and verification, while multi-level and suffix-sharing techniques provide other opportunities to avoid redundant computation.

8 Conclusions

In this paper, a novel computation-sharing framework is proposed for supergraph containment search. All data graphs are clustered into disjoint groups for computation-sharing within each group. While the optimization problem MG is shown NP-hard, efficient greedy heuristic is used to approximate the optimal solution with an approximation factor of $\frac{1}{2}$. Based on the compact index structure, PrefIndex, an efficient algorithm PrefSearch integrating filtering and verification is proposed. PrefIndex is enhanced with multi-level sharing and suffix-sharing techniques to explore further sharing opportunities. An extensive empirical study demonstrates the efficiency and scalability of our proposed techniques which achieve orders of magnitudes of speed-up against the state-of-the-art techniques.

References

1. Berretti, S., Bimbo, A.D., Vicario, E.: Efficient matching and indexing of graph models in content-based retrieval. IEEE Trans. Pattern Anal. Mach. Intell. 23(10), 1089–1105 (2001)
2. Chen, C., Yan, X., Yu, P.S., Han, J., Zhang, D.-Q., Gu, X.: Towards graph containment search and indexing. In: VLDB, pp. 926–937 (2007)
3. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD Conference, pp. 857–872 (2007)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
5. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS, pp. 39–52, 200
6. He, H., Singh, A.K.: Closure-tree: An index structure for graph queries. In: ICDE, p. 38 (2006)
7. Jiang, H., Wang, H., Yu, P.S., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE, pp. 566–575 (2007)

8. Messmer, B.T., Bunke, H.: A decision tree approach to graph and subgraph isomorphism detection. Pattern Recognition 32(12), 1979–1998 (1999)
9. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB 1(1), 364–375 (2008)
10. Williams, D.W., Huan, J., Wang, W.: Graph database indexing using structured graph decomposition. In: ICDE, pp. 976–985 (2007)
11. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: ICDM, pp. 721–724 (2002)
12. Yan, X., Yu, P.S., Han, J.: Graph indexing: A frequent structure-based approach. In: SIGMOD Conference, pp. 335–346 (2004)
13. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: ICDE, pp. 966–975 (2007)
14. Zhang, S., Li, J., Gao, H., Zou, Z.: A novel approach for efficient supergraph query processing on graph databases. In: EDBT, pp. 204–215 (2009)
15. Zhao, P., Yu, J.X.: Fast frequent free tree mining in graph databases. In: ICDM Workshops, pp. 315–319 (2006)
16. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: Tree + delta $\geq$ graph. In: VLDB, pp. 938–949 (2007)
17. Zou, L., Chen, L., Yu, J.X., Lu, Y.: A novel spectral coding in a large graph database. In: EDBT, pp. 181–192 (2008)