Graph Indexing: Tree + Delta >= Graph

Peixiang Zhao
The Chinese University of Hong Kong
pxzhao@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Philip S. Yu
IBM T. J. Watson Research Center
psyu@us.ibm.com

ABSTRACT

Recent scientific and technological advances have witnessed an abundance of structural patterns modeled as graphs. As a result, it is of special interest to process graph containment queries effectively on large graph databases. Given a graph database $\mathcal{G}$, and a query graph q, the *graph containment query* is to retrieve all graphs in $\mathcal{G}$ which contain q as subgraph(s). Due to the vast number of graphs in $\mathcal{G}$ and the nature of complexity for subgraph isomorphism testing, it is desirable to make use of high-quality graph indexing mechanisms to reduce the overall query processing cost. In this paper, we propose a new cost-effective graph indexing method based on frequent tree-features of the graph database. We analyze the effectiveness and efficiency of tree as indexing feature from three critical aspects: feature size, feature selection cost, and pruning power. In order to achieve better pruning ability than existing graph-based indexing methods, we select, in addition to frequent tree-features (**Tree**), a small number of discriminative graphs ($\mathbf{\Delta}$) on demand, without a costly graph mining process beforehand. Our study verifies that **(Tree+$\mathbf{\Delta}$)** is a better choice than graph for indexing purpose, denoted **(Tree+$\mathbf{\Delta} \geq$Graph)**, to address the graph containment query problem. It has two implications: (1) the index construction by (Tree+Δ) is efficient, and (2) the graph containment query processing by (Tree+Δ) is efficient. Our experimental studies demonstrate that (Tree+Δ) has a compact index structure, achieves an order of magnitude better performance in index construction, and most importantly, outperforms up-to-date graph-based indexing methods: `gIndex` and `C-Tree`, in graph containment query processing.

1. INTRODUCTION

Recent scientific and technological advances have resulted in an abundance of data modeled as graphs. As a general data structure representing relations among entities, graph has been used extensively in modeling complicated structures and schemaless data, such as proteins [3], images [4], visions [7], program flows [13], XML documents [15], the Internet and the Web [6], etc. The dominance of graphs in real-world applications asks for effective graph data management so that users can organize, access, and analyze graph data in a way one might have not yet imagined. Among myriad graph-related problems of interest, a common and critical one shared in many applications in science and engineering is the *graph containment query* problem: given a massive dataset with each transaction modeled as graphs, and a query, represented as graph too, find all graphs in the database which contain the query as a subgraph(s) [1, 4, 9, 20, 21, 23].

To answer graph containment queries effectively is challenging for a number of reasons. First, to determine whether a graph in the database contains the query graph is a *subgraph isomorphism* problem, which has been proven to be NP-complete [8]; Second, the graph database can be large, which makes a sequential scan over the database impracticable. As to graph databases, existing database infrastructures might not answer graph containment queries in an efficient manner. For example, the indices built on the labels of vertices or edges are usually not selective enough to distinguish complicated, interconnected structures. Therefore, high performance graph indexing mechanisms need to be devised to prune graphs that obviously violate the query requirement. In this way, the number of costly subgraph isomorphism testings is reduced, which is the primary motivation of our study.

The strategy of graph indexing is to shift high online query processing cost to the off-line index construction phase. So index construction is always computationally expensive because it requests deliberately selecting high quality indexing features with great pruning power from the graph database. The process of selection and evaluation of indexing features is critical because features with higher pruning power are superior to be selected as index entries. At the same time, the number of indexing features should be as small as possible to keep the whole index structure compact, better to be held in main memory for efficient access and retrieval. In sum, a high quality graph indexing mechanism should be time-efficient in index construction, and indexing features should be compact and powerful for pruning purpose.

In this paper we present a new *tree-based* indexing approach to address the graph containment query problem. Our work is motivated by an evidence that a large number of frequent graphs in the graph database are trees in nature. In fact, for many real-world datasets, over 95% of frequent graphs are trees. It leads us to reconsider an alternative

VLDB '07, September 23-28, 2007, Vienna, Austria.

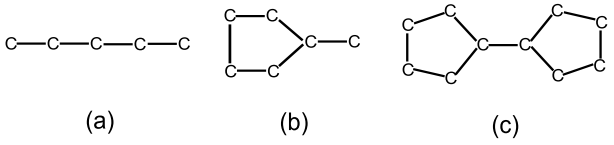


Figure 1: A Graph Database with Three Graphs

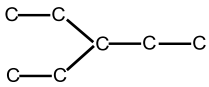


Figure 2: A Query Graph

solution: "*Can we use tree instead of graph as the basic indexing feature?*" Tree, which is also denoted as *free tree*, is a special connected, acyclic and undirected graph. As a generalization of linear sequential patterns, tree preserves plenty of structural information of graph. Meanwhile, tree is also a specialization of general graph, which avoids undesirable theoretical properties and algorithmic complexity incurred by graph. As the middle ground between these two extremes, tree becomes an ideal candidate of indexing features over the graph database. The main contributions of this paper are summarized below.

- We analyze the effectiveness and efficiency of trees as indexing features by comparing them with paths and graphs from three critical aspects, namely, feature size, feature selection cost, and pruning power. We show that tree-features can be effectively and efficiently used as indexing features for graph databases. Our main results show: (1) in many applications the majority of graph-features (usually more than 95%) are tree-features indeed; (2) frequent tree-features and graph-features share similar distributions and frequent tree-features have similar pruning power like graph-features; and (3) tree mining can be done much more efficiently than graph mining (it is not cost-effective to mine frequent graph features in which more than 95% are trees).
- We propose a new graph indexing mechanism, called (Tree+Δ), that first selects frequent tree-features as the basis of a graph index, and then on-demand selects a small number of discriminative graph-features that can prune graphs more effectively than the selected tree-features, without conducting costly graph mining beforehand. A key issue here is how to achieve the similar pruning power of graph-features without graph mining. We propose a new approach by which we can approximate the pruning power of a graph-feature by its subtree-features with upper/lower bounds.
- We conducted extensive experimental studies using a real dataset and a series of synthetic datasets. We compared our (Tree+Δ) with two up-to-date graph-based indexing methods: `gIndex` [23] and `C-Tree` [9]. Our study confirms that (tree+Δ) outperforms `gIndex` and `C-Tree` in terms of index construction cost and query processing cost.

The rest of the paper is organized as follows. In Section 2, we give the problem statement for the graph containment query processing, and discuss an algorithmic framework with a cost model. In Section 3, we analyze the indexability of frequent features (path, tree and graph) from three perspectives: feature size, feature selection cost, and pruning power. Section 4 discusses our new approach to add discriminative graph-features on demand. Section 5 presents the implementation details of our indexing algorithm (Tree+Δ) with an emphasis on index construction and query processing. Section 6 shows the related work concerning the graph containment query problem over large graph databases. Our experimental study is reported in Section 7. Section 8 concludes this paper.

2. PRELIMINARIES

In this section, we introduce preliminary concepts and outline an algorithmic framework to address the graph containment query problem. A cost evaluation model is also presented on which our analysis of graph indexing solutions are based.

2.1 Problem Statement

A graph $G = (V, E, \Sigma, \lambda)$ is defined as a undirected labeled graph where V is a set of vertices, E is a set of edges (unordered pairs of vertices), Σ is a set of labels, and λ is a labeling function, $\lambda : V \cup E \rightarrow \Sigma$, that assigns labels to vertices and edges. Let g and g' be two graphs. g is a *subgraph* of g', or g' is a *supergraph* of g, denoted $g \subseteq g'$, if there exists a subgraph isomorphism from g to g'. We also call g' *contains* g or g is *contained* by g'. The concept of *subgraph isomorphism* from g to g' is defined as a injective function from V_g to $V_{g'}$ that preserves vertex labels, edge labels and adjacency. The concept of *graph isomorphism* can be defined analogously by using a bijective function instead of an injective function. The size of g is denoted $size(g) = |V_g|$. A tree, also known as *free tree*, is a special undirected labeled graph that is connected and acyclic. For tree, the concept of *subtree*, *supertree*, *subtree isomorphism*, *tree isomorphism* can be defined accordingly. A path is the simplest tree whose vertex degrees are no more than 2.

Given a graph database $\mathcal{G} = \{g_1, g_2, \cdots, g_n\}$ and an arbitrary graph g, let $sup(g) = \{g_i | g \subseteq g_i, g_i \in \mathcal{G}\}$. $|sup(g)|$ is the *support*, or *frequency* of g in $\mathcal{G}$. g is *frequent* if its support is no less than $\sigma \cdot |\mathcal{G}|$, where σ is a minimum support threshold provided by users.

Graph Containment Query Problem: Given a graph database, $\mathcal{G} = \{g_1, g_2, \cdots, g_n\}$, and a query graph q, a graph containment query is to find the set, $sup(q)$, from $\mathcal{G}$.

The graph containment query problem is NP-complete. It is infeasible to find $sup(q)$ by sequentially checking subgraph isomorphism between q and every $g_i \in \mathcal{G}$, for $1 \leq i \leq n$. And it is especially challenging when graphs in $\mathcal{G}$ are large, and $|\mathcal{G}|$ is also large in size and diverse. Graph indexing provides an alternative to tackle the graph containment query problem effectively.

Example 2.1: A sample query graph, shown in Figure 2, is posed to a sample graph database with three graphs, shown in Figure 1. The graph in Figure 1 (c) is the answer. □

2.2 An Algorithmic Framework

Given a graph database $\mathcal{G}$, and a query, q, the graph containment query can be processed in two steps. First, a preprocessing step called *index construction* generates indexing features from $\mathcal{G}$. The feature set, denoted $\mathcal{F}$, constructs the index, and for each feature $f \in \mathcal{F}$, $sup(f)$ is maintained. Second, a *query processing* step is performed in a *filtering-*

Table 1: Frequent Features as Indices

	$\lvert\mathcal{F}\rvert$	$\mathcal{C}_{FS}$	$\lvert C_q\rvert$	Comments
$\mathcal{F}_P$	small	low	large	simple structure, easy to be discovered, limited indexing ability
$\mathcal{F}_T$	large	intermediate	small	proper structure, easy to be discovered, good indexing ability
$\mathcal{F}_G$	large	high	small	complex structure, hard to be discovered, good indexing ability

verification fashion. The filtering phase uses indexing features contained in q to compute the *candidate answer set*, defined as

$$C_q = \bigcap_{f \subseteq q \wedge f \in \mathcal{F}} sup(f) \quad (1)$$

Every graph in C_q contains all q's indexing features. Therefore, the query answer set, $sup(q)$, is a subset of C_q. The verification phase checks subgraph isomorphism for every graph in C_q. False positives are pruned from C_q and the true answer set $sup(q)$ is returned.

Eq. (1) suggests indexing structural patterns, that have great pruning power, to reduce false positives included in the candidate answer set, C_q. Furthermore, a small-sized index including high-frequency features is preferred due to its compact storage. However, it is often unknown beforehand which patterns are valuable for indexing.

2.3 Query Cost Model

The cost of processing a graph containment query q upon $\mathcal{G}$, denoted $\mathcal{C}$, can be modeled below

$$\mathcal{C} = \mathcal{C}_f + |C_q| \times \mathcal{C}_v \quad (2)$$

Here, $\mathcal{C}_f$ is the total cost for filtering based on Eq. (1) in the filtering phase. Every graph in C_q needs to be fetched from the disk in the verification phase to verify subgraph isomorphism, where $\mathcal{C}_v$ is such an average cost.

The key issue to improve query performance is to minimize $|C_q|$ for a graph containment query, q. Intuitively, $|C_q|$ will be minimized if we index all possible features of $\mathcal{G}$. However, it is infeasible because the feature set, $\mathcal{F}$, can be very large, which makes the space complexity prohibitive, and the filtering cost $\mathcal{C}_f$ becomes large accordingly. In other words, enlarging $\mathcal{F}$ will increase the cost of $\mathcal{C}_f$, but probably reduce $|C_q|$; On the other hand, reducing $\mathcal{F}$ will decease $\mathcal{C}_f$ but probably increase $|C_q|$. There is a subtle trade-off between time and space in the graph containment query problem.

As seen above, the graph containment query problem is challenging. An effective graph indexing mechanism with high quality indexing features is required to minimize the query cost $\mathcal{C}$ as much as possible. On the other hand, the feature selection process itself introduces another non-negligible cost for index construction, *i.e.*, the cost to discover $\mathcal{F}$ from $\mathcal{G}$, denoted $\mathcal{C}_{FS}$. A graph indexing mechanism is also cost-effective if it results in a small feature set size (or index size), $|\mathcal{F}|$, to reduce $\mathcal{C}_f$ in the query processing. In next section, we discuss the three major factors: $|\mathcal{F}|$, $\mathcal{C}_{FS}$, and $|C_q|$, that affect $\mathcal{C}$.

3. GRAPH *VS.* TREE *VS.* PATH

Frequent features (paths, trees, graphs) expose intrinsic characteristics of a graph database. They are representatives to discriminate between different groups of graphs in a graph database. We denote the frequent path-feature set as $\mathcal{F}_P$, the frequent tree-feature set as $\mathcal{F}_T$ and the frequent graph-feature set as $\mathcal{F}_G$. In the following, when we discuss different frequent feature sets, we assume that all the frequent feature sets are mined with the minimum support threshold σ. Note: $\mathcal{F}_P \subseteq \mathcal{F}_T \subseteq \mathcal{F}_G$. Because path is a special tree, and tree is a spatial graph, we have the following.

$$\mathcal{F}_{T'} = \mathcal{F}_T - \mathcal{F}_P \quad (3)$$

$$\mathcal{F}_{G'} = \mathcal{F}_G - \mathcal{F}_T \quad (4)$$

$$\mathcal{F}_G = \mathcal{F}_P \cup \mathcal{F}_{T'} \cup \mathcal{F}_{G'} \quad (5)$$

Here, $\mathcal{F}_{T'}$ and $\mathcal{F}_{G'}$ denote the tree-feature set without any path-features (nonlinear tree-features, Eq. (3)) and the graph-feature set without any tree-features (non-tree graph-features, Eq. (4)). The relationships among them are given in Eq. (5). Note: $\mathcal{F}_P \cap \mathcal{F}_{T'} = \emptyset$, $\mathcal{F}_P \cap \mathcal{F}_{G'} = \emptyset$, and $\mathcal{F}_{T'} \cap \mathcal{F}_{G'} = \emptyset$.

We explore the indexability of frequent features for the purpose of addressing the graph containment query problem, and focus on the three major factors, namely, the size of a frequent feature set ($|\mathcal{F}|$), the feature selection cost ($\mathcal{C}_{FS}$), and the candidate answer set size ($|C_q|$), that affect the query processing cost $\mathcal{C}$ (Eq. (1)). Some observations are summarized in Table 1 for path-feature set ($\mathcal{F}_P$), tree-feature set ($\mathcal{F}_T$) and graph-feature set ($\mathcal{F}_G$). We discuss each of them in detail below.

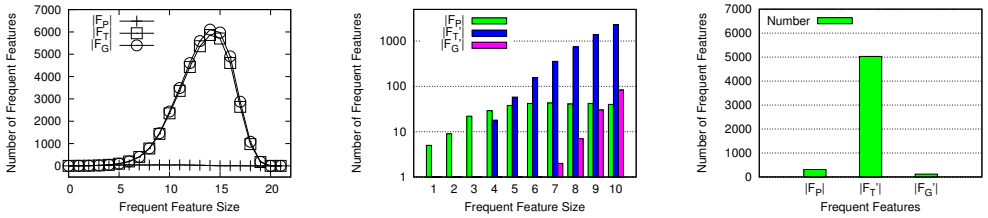
3.1 Feature Set Size: $|\mathcal{F}|$

The size of a frequent feature set ($|\mathcal{F}|$) is of special interest because a space-efficient index is critical to query performance, as mentioned in Section 2.3. Below, we discuss different frequent feature sets: $\mathcal{F}_G$, $\mathcal{F}_T$, and $\mathcal{F}_P$ *w.r.t.* feature set size.

To our surprise, among all frequent graph-features in $\mathcal{F}_G$ (Eq. (5)), the majority (usually more than 95%) are trees. The non-tree frequent graph-feature set, $\mathcal{F}_{G'}$, can be very small in size, and the significant portion of $\mathcal{F}_G$ is $\mathcal{F}_{T'}$, *i.e.*, non-linear tree-features. $\mathcal{F}_P$ shares a very small portion in $\mathcal{F}_G$, because a path-feature has a simple linear structure, which has little variety in structural complexity. We explain the reasons below. First, for a non-tree frequent graph-feature, $g \in \mathcal{F}_{G'}$, based on the Apriori principle, all g's subtrees, $t_1, t_2, \cdots, t_n$ are frequent, in other words, $t_i \in \mathcal{F}_T$, for $1 \leq i \leq n$. Second, given two *arbitrary* frequent non-tree graph-features, g and g', in $\mathcal{F}_{G'}$, because of the structural diversity and (vertex/edge) label variety, there is little chance that subtrees of g coincide with those of g'. It is true especially when graphs of $\mathcal{G}$ are large and labels of graph are diverse.

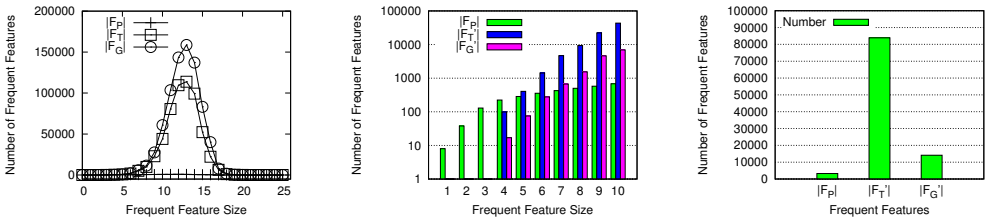
Consider a complete graph K_{10} with 10 vertices, and each vertex has a distinct label. If K_{10} is frequent, all its 10 1-vertex subtrees, 45 2-vertex subtrees, $\cdots$, and $100,000,000$ 10-vertex subtrees are frequent. There are $162,191,420$ frequent subtrees in total. If the linear trees (paths) are excluded, there are still $156,654,565$ frequent trees involved.

Next, we investigate the frequent feature distributions. In other words, we consider the distributions of the total



(a) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_T|$ *vs.* $|\mathcal{F}_G|$ (b) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_{T'}|$ *vs.* $|\mathcal{F}_{G'}|$ (c) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_{T'}|$ *vs.* $|\mathcal{F}_{G'}|$

Figure 3: The Real Dataset $N = 1000$, $\sigma = 0.1$



(a) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_T|$ *vs.* $|\mathcal{F}_G|$ (b) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_{T'}|$ *vs.* $|\mathcal{F}_{G'}|$ (c) $|\mathcal{F}_P|$ *vs.* $|\mathcal{F}_{T'}|$ *vs.* $|\mathcal{F}_{G'}|$

Figure 4: The Synthetic Dataset $N = 1000$, $\sigma = 0.1$

number of frequent features that have the same size, n. Figure 3 (a) illustrates the numbers of frequent features (paths, trees and graphs) *w.r.t.* the feature size (modeled as vertex number), in a sample AIDS antivirus screen dataset of $1,000$ graphs and $\sigma = 0.1$. When n increases, both $|\mathcal{F}_T|$ and $|\mathcal{F}_G|$ share similar distributions. They first grow exponentially, and then drop exponentially. Similar phenomena appear in the sample synthetic dataset of $1,000$ graphs and $\sigma = 0.1$, as illustrated in Figure 4 (a). It is worth noting that in terms of distribution, the frequent tree-features and the frequent graph-features behave in a similar way. Note: $|\mathcal{F}_P|$ is almost unchanged while n increases, because of simple linear structure.

We also compare among the frequent path-feature set, $\mathcal{F}_P$, the frequent nonlinear tree-feature set, $\mathcal{F}_{T'} = \mathcal{F}_T - \mathcal{F}_P$, and the frequent non-tree graph-feature set, $\mathcal{F}_{G'} = \mathcal{F}_G - \mathcal{F}_T$. In the sample AIDS antivirus screen dataset, when the feature size varies from $n = 1$ to 10, the number of frequent nonlinear tree-features grows exponentially and is much larger than frequent non-tree graph-features, as shown in Figure 3 (b). In total, for all frequent features with size up to 10, $\mathcal{F}_{T'}$ dominates, as shown in Figure 3 (c). In the sample synthetic dataset, non-tree graph-features more frequently appear in $\mathcal{G}$, so many frequent non-tree graph-features are discovered, in comparison with the sample AIDS antivirus dataset. However, the number of frequent non-path tree-features still grows much faster than non-tree graphs, as shown in Figure 4 (b). $\mathcal{F}_{T'}$ dominates $\mathcal{F}_G$, as shown in Figure 4 (c).

Based on the analysis mentioned above, some conclusions can be made *w.r.t.* the feature size $|\mathcal{F}|$: First, in terms of feature distributions, tree-features and graph-features share a very similar distribution; Second, the size of frequent tree-feature set dominates the total feature set, $\mathcal{F}$; Third, the number of non-tree graph-features is very small in $\mathcal{F}_G$.

3.2 Feature Selection Cost: $\mathcal{C}_{FS}$

In the previous section, we indicated that, in the frequent graph-feature set $\mathcal{F}_G$, the number of non-tree graph-features can be very small, *i.e.*, the majority of graph-features in $\mathcal{F}_G$ are trees (including paths) in nature. It motivates us to reconsider if there is a need to conduct costly graph mining to discover frequent graph-features in which most of them (usually more than 95%) are trees indeed. Below, we briefly review the cost for mining frequent paths/trees/graphs from graph databases.

Given a graph database, $\mathcal{G}$, and a minimum support threshold, σ, we sketch an algorithmic framework, called *FFS* (**F**requent **F**eature **S**election), shown in Algorithm 1, to discover the frequent feature set $\mathcal{F}$ ($\mathcal{F}_P/\mathcal{F}_T/\mathcal{F}_G$) from $\mathcal{G}$. The algorithm initiates from the smallest frequent features in $\mathcal{G}$ ($size(f) = 1$ at line 2) and expands current frequent features by growing one vertex each time. This pattern-growth process, denoted *FFM* (**F**requent **F**eature **M**ining), is recursively called until all frequent features are discovered from $\mathcal{G}$, shown in Algorithm 2. The *candidate frequent feature set* of a frequent feature f, $\mathcal{C}(f) = \{f' | size(f') = size(f)+1\}$, is determined as a projected database, upon which frequency checking is performed to discover larger-size frequent features (line $5-6$).

If the frequent graph-feature set $\mathcal{F}_G$ needs to be selected from $\mathcal{G}$ by the *FFS* algorithm, two prohibitive operations are unavoidable. In Algorithm 2, line 1, graph isomorphism has to be checked to determine whether a frequent graph-feature, f, has already been selected in $\mathcal{F}$ or not. If f has been discovered before, there is no need to mine f and all its supergraphs again. The graph isomorphism problem is not known to be either P or NP-complete, so it is unlikely to check $f \in \mathcal{F}$ in polynomial time. In Algorithm 2, line 5, subgraph isomorphism testing needs to be performed to determine whether a candidate graph $f' \in \mathcal{C}(f)$ is frequent or not. The subgraph isomorphism testing has been proven to be NP-complete. Therefore, the feature selection cost, $\mathcal{C}_{FS}$, for discovering $\mathcal{F}_G$ from $\mathcal{G}$ is expensive.

If the frequent tree-feature set $\mathcal{F}_T$ needs to be selected, although a costly *tree-in-graph* testing is still unavoidable in Algorithm 2, line 5, tree isomorphism in Algorithm 2, line 1, can be performed efficiently in $O(n)$, where n is tree size [2]. Compared with $\mathcal{F}_G$, $\mathcal{F}_T$ can be discovered much more efficiently from $\mathcal{G}$.

If the frequent path feature set $\mathcal{F}_P$ needs to be selected, the costly (sub)graph isomorphism checking boils down to

Algorithm 1 *FFS* ($\mathcal{G}$, σ)

Input: A graph database $\mathcal{G}$, the minimum support threshold σ
Output: The frequent feature set $\mathcal{F}$

1: $\mathcal{F} \leftarrow \emptyset$;
2: **for all** frequent features, f, with $size(f) = 1$ **do**
3: $\quad FFM(f, \mathcal{F}, \mathcal{G}, \sigma)$;
4: **return** $\mathcal{F}$;

Algorithm 2 *FFM* (f, $\mathcal{F}$, $\mathcal{G}$, σ)

Input: A frequent feature f, the frequent feature set $\mathcal{F}$, $\mathcal{G}$ and σ
Output: The frequent feature set $\mathcal{F}$

1: **if** ($f \in \mathcal{F}$) **then return**;
2: $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$;
3: Scan $\mathcal{G}$ to determine the candidate frequent feature set of f, $\mathcal{C}(f) = \{f' | size(f') = size(f) + 1\}$;
4: **for all** $f' \in \mathcal{C}(f)$ **do**
5: $\quad$ **if** $|sup(f')| \geq \sigma \times |\mathcal{G}|$ **then**
6: $\quad\quad FFM(f', \mathcal{F}, \mathcal{G}, \sigma)$;

the (sub)string matching operation, which can be done efficiently in polynomial time. Compared with $\mathcal{F}_G$ and $\mathcal{F}_T$, the discovery of $\mathcal{F}_P$ from $\mathcal{G}$ is the most efficient.

Based on the analysis mentioned above, we draw the conclusion *w.r.t.* the feature selection cost, $\mathcal{C}_{FS}$: It is not cost-effective to make use of an expensive graph mining process to select frequent graph-features, most of which are tree-features indeed.

3.3 Candidate Answer Set Size: $|C_q|$

The key to boost graph containment query performance is to minimize candidate answer set size, $|C_q|$, given a query q. It requests selecting frequent features that have great pruning power (Eq. (1)). We define the pruning power $\mathsf{power}(f)$ of a frequent feature, $f \in \mathcal{F}$, in Eq. (6),

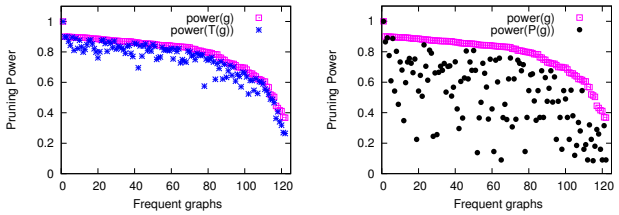
$$\mathsf{power}(f) = \frac{|\mathcal{G}| - |sup(f)|}{|\mathcal{G}|} \tag{6}$$

The pruning power, $\mathsf{power}(f)$, gives a real number between 0 and 1, such as $0 \leq P(f) < 1$. When $\mathsf{power}(f) = 0$, f is contained in every graph of $\mathcal{G}$, so f has no pruning ability if it is included in the index. When $\mathsf{power}(f)$ approaches to 1, the feature f has great pruning power to reduce $|C_q|$, if it appears in q. Note $\mathsf{power}(f)$ cannot be 1, because f is a feature that appears at least once in $\mathcal{G}$. Since q may contain multiple frequent features, we consider the pruning power bestowed by a set of such features together. Let $\mathcal{S} = \{f_1, f_2, \cdots, f_n\} \subseteq \mathcal{F}$, for $1 \leq i \leq n$. The pruning power of $\mathcal{S}$ can be similarly defined as

$$\mathsf{power}(\mathcal{S}) = \frac{|\mathcal{G}| - |\bigcap_{i=1}^{n} sup(f_i)|}{|\mathcal{G}|} \tag{7}$$

Lemma 3.1: *Given a frequent feature $f \in \mathcal{F}$. Let its frequent sub-feature set be $\mathcal{S}(f) = \{f_1, f_2, \cdots, f_n\} \subseteq \mathcal{F}$, for $f_i \subseteq f$ and $1 \leq i \leq n$. Then,* $\mathsf{power}(f) \geq \mathsf{power}(\mathcal{S}(f))$. □

Proof Sketch: for all $g_i \in sup(f)$, we have $f \subseteq g_i$. For every $f_i \in \mathcal{S}(f)$, because $f_i \subseteq f$, $f_i \subseteq g_i$ is true. Hence, $g_i \in sup(f_i)$. If $g_i \in sup(f_i)$, for $1 \leq i \leq n$, then $g_i \in \bigcap_{i=1}^{n} sup(f_i)$. Therefore, $sup(f) \subseteq \bigcap_{i=1}^{n} sup(f_i)$. It implies $|sup(f)| \leq |\bigcap_{i=1}^{n} sup(f_i)|$, so $\mathsf{power}(f) \geq \mathsf{power}(\mathcal{S}(f))$. □



(a) $\mathsf{power}(g)$ *vs.* $\mathsf{power}(\mathcal{T}(g))$ (b) $\mathsf{power}(g)$ *vs.* $\mathsf{power}(\mathcal{P}(g))$

Figure 5: Pruning Power

Based on Lemma 3.1, we can directly get the following two important results:

Theorem 3.1: Given a frequent graph-feature $g \in \mathcal{F}$, and let its frequent sub-tree set be $\mathcal{T}(g) = \{t_1, t_2, \cdots, t_n\} \subseteq \mathcal{F}$. Then, $\mathsf{power}(g) \geq \mathsf{power}(\mathcal{T}(g))$. □

Theorem 3.2: Given a frequent tree-feature $t \in \mathcal{F}$, and let its frequent sub-path set be $\mathcal{P}(t) = \{p_1, p_2, \cdots, p_n\} \subseteq \mathcal{F}$. Then, $\mathsf{power}(t) \geq \mathsf{power}(\mathcal{P}(t))$. □

Theorem 3.1 demonstrates that the pruning power of a frequent graph-feature is no less than that of all its frequent subtree-features. Similarly, the pruning power of a frequent tree-feature is no less than that of all its frequent sub-path features, as presented in Theorem 3.2. Therefore, among all frequent features in $\mathcal{F}$, graph-feature has the greatest pruning power; path-feature has the least pruning power; while tree-feature stands in the middle sharing the pruning power less than graph-feature, but more than path-feature.

It is interesting to note that the pruning power of all frequent subtree-features, $\mathcal{T}(g)$, of a frequent graph-feature g can be similar to the pruning power of g. It is because $\mathcal{T}(g)$ may well preserve structural information provided by g. However, in general, there is a big gap between the pruning power of a graph-feature g and that of all its frequent sub-path features, $\mathcal{P}(g)$. It is because, when g is replaced by $\mathcal{P}(g)$, the structural information of g is almost lost and it becomes difficult to identify g in $\mathcal{P}(g)$. Therefore, frequent path-features ($\mathcal{P}(g)$) can not be effectively used as a candidate to substitute g, in terms of pruning power.

Figure 5 illustrates the pruning power distributions of frequent graph-features with regard to their subtrees and subpaths in the sample real dataset mentioned in Section 3. Among 122 frequent non-tree graph-features found in $\mathcal{G}$, for each frequent graph-feature, g, its pruning power $\mathsf{power}(g)$ is firstly compared with $\mathsf{power}(\mathcal{T}(g))$, (Figure 5 (a)), and then compared with $\mathsf{power}(\mathcal{P}(g))$ (Figure 5 (b)). We observe that $\mathsf{power}(\mathcal{T}(g))$ is very close to $\mathsf{power}(g)$ for almost all frequent graph-features. However, $\mathcal{P}(g)$, the set of frequent subpaths of g, has quite limited pruning power.

Remark 3.1: *The frequent tree-feature set, $\mathcal{F}_T$, dominates $\mathcal{F}_G$ in quantity, and $\mathcal{F}_T$ can be discovered much more efficiently than $\mathcal{F}_G$ from $\mathcal{G}$. In addition, $\mathcal{F}_T$ can contribute similar pruning power like that provided by $\mathcal{F}_G$. It is feasible and effective to select $\mathcal{F}_T$, instead of $\mathcal{F}_G$, as indexing features for the graph containment query problem.* □

Consider Example 2.1. We explain the disadvantages of the path-based indexing approach that uses frequent path-features to prune. As shown in Example 2.1, when the query graph, q (Figure 2) is issued against the graph database Figure 1. Only the graph in Figure 1 (c) is the answer. But,

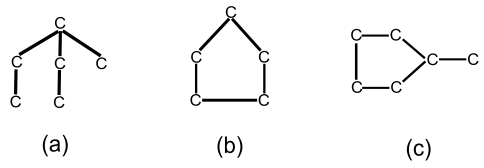


Figure 6: Frequent Graphs of $\mathcal{G}$

all the path-features appearing in the query graph q are c, c–c, c–c–c, c–c–c–c and c–c–c–c–c. They cannot be used to prune the two graphs in Figure 1 (a) and (b), even if all these path-features are frequent in the graph database. Reconsider Example 2.1 for the graph-based indexing approach that uses frequent graph-features as index entries. This approach needs to mine frequent graph-features beforehand, which incurs high computation cost. In this example, some frequent graph-features discovered are shown in Figure 6, with $\sigma = 2/3$. In order to answer the query graph (Figure 2), only the graph-feature (Figure 6 (a)) can be used, which is a tree-feature in nature, while other frequent graph-features (Figure 6 (b) and Figure 6 (c)) are mined wastefully.

4. GRAPH FEATURE ON DEMAND

Based on the discussions in Section 3, a tree-based indexing mechanism can be efficiently deployed. It is compact and can be easily maintained in main memory, as shown in our performance studies. We have also shown that a tree-based index can have similar pruning power like that provided by the costly graph-based index on average in general. However, based on Theorem 3.1, it is still necessary to use effective graph-features to reduce the candidate answer set size, $|C_q|$, while tree-features cannot. In this section, we discuss how to select additional non-tree graph-features from q on demand that have greater pruning power than their subtree-features, based on the tree-feature set discovered.

Consider a query graph q, which contains a non-tree subgraph $g \in \mathcal{F}_{G'}$. If $\mathsf{power}(g) \approx \mathsf{power}(\mathcal{T}(g))$ *w.r.t.* pruning power, there is no need to index the graph-feature g, because its subtrees jointly have the similar pruning power. However, if $\mathsf{power}(g) \gg \mathsf{power}(\mathcal{T}(g))$, it will be necessary to select g as an indexing feature because g is more *discriminative* than $\mathcal{T}(g)$ for pruning purpose. Note the concept we use here in this paper is different from the discriminative graph concept used in `gIndex`, which is based on two frequent graph-features instead.

In this paper, we select discriminative graph-features from queries on-demand, without mining the whole set of frequent graph-features from $\mathcal{G}$ beforehand. These selected discriminative graph-features are therefore used as additional indexing features, denoted Δ, which can also be reused further to answer subsequent queries.

In order to measure the similarity of pruning power between a graph-feature g and its subtrees, $\mathcal{T}(g)$, we define a *discriminative ratio*, denoted $\varepsilon(g)$, for a non-tree graph, $g \in \mathcal{F}_{G'}$ *w.r.t.* $\mathcal{T}(g)$ as

$$\varepsilon(g) = \begin{cases} \dfrac{\mathsf{power}(g) - \mathsf{power}(\mathcal{T}(g))}{\mathsf{power}(g)} & \text{if } \mathsf{power}(g) \neq 0 \\ 0 & \text{if } \mathsf{power}(g) = 0 \end{cases} \quad (8)$$

Here, $0 \leq \varepsilon(g) \leq 1$. When $\varepsilon(g) = 0$, g has the same pruning power as the set of all its frequent subtrees, $\mathcal{T}(g)$. The larger $\varepsilon(g)$ is, the greater pruning power g has than $\mathcal{T}(g)$. When $\varepsilon(g) = 1$, the frequent subtree set $\mathcal{T}(g)$ has no pruning power, while g is the most discriminative graph-feature and definitely needed to be reclaimed and indexed from the graph database, $\mathcal{G}$. Based on Eq. (8), we define a *discriminative graph* in Definition 4.1:

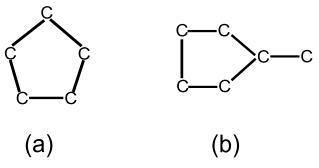


Figure 7: Discriminative Graphs

Definition 4.1: A non-tree graph $g \in \mathcal{F}_{G'}$ is discriminative if $\varepsilon(g) \geq \varepsilon_0$, where ε_0 is a user-specified minimum discriminative threshold ($0 < \varepsilon_0 < 1$). □

If a frequent non-tree graph g is not discriminative, we consider that there is no need to select g as an indexing feature, because it can not contribute more for pruning than its frequent subtrees that have already been used as indexing features. Otherwise, there is a good reason to reclaim g from $\mathcal{G}$ into the index, because g has greater pruning power than all its frequent subtrees ($\mathcal{T}(g)$).

Suppose we set $\sigma = 2/3$ and $\varepsilon_0 = 0.5$ for the sample database in Figure 1. Figure 7 illustrates two discriminative frequent graph-features. The pruning power of Figure 7 (a) is $(1 - 2/3) = 1/3$ and the pruning power of Figure 7 (b) is $(1 - 1/3) = 2/3$. Note: all frequent subtrees in Figure 7 (a) are subtrees of c–c–c–c–c, whose pruning power is 0. So the discriminative ratio, ε, of Figure 7 (a) is 1. The discriminative ratio ε of Figure 7 (b) can be computed similarly as 1/2.

4.1 Discriminative Graph Selection

Given a query q, let's denote its discriminative subgraph set as $\mathcal{D}(q) = \{g_1, g_2, \cdots, g_n\}$, where every non-tree graph $g_i \subseteq q$ $(1 \leq i \leq n)$ is frequent and discriminative *w.r.t.* its subtree set, $\mathcal{T}(g_i)$. For $\mathcal{D}(q)$, it is not necessary to reclaim every g_i from $\mathcal{G}$ as indexing features, because to reclaim g_i from $\mathcal{G}$ means to compute $sup(g_i)$ from scratch, which incurs costly subgraph isomorphism testings over the whole database. Given two graphs $g, g' \in \mathcal{D}(q)$, where $g \subseteq g'$, intuitively, if the gap between $\mathsf{power}(g')$ and $\mathsf{power}(g)$ is large enough, g' will be reclaimed from $\mathcal{G}$; Otherwise, g is discriminative enough for pruning purpose, and there is no need to reclaim g' in the presence of g. Based on the above analysis, we propose a new strategy to select discriminative graphs from $\mathcal{D}(q)$.

Recall in [23], a frequent graph-feature, g', is discriminative, if its support, $|sup(g')|$, is significantly greater than $|sup(g)|$, where g' is a supergraph of g. It is worth noting that a costly graph mining process is needed to compute $sup(g')$ and $sup(g)$. Below, we discuss our approach to select discriminative graph-features without graph mining beforehand. In order to do so, we approximate the discriminative computation between g' and g, in the presence of our knowledge on frequent tree-features discovered.

$$\begin{array}{ccc} sup(g)(?) & \xrightarrow{\ ?\ } & sup(g')(?) \\ \uparrow & & \uparrow \\ sup(\mathcal{T}_g) & \longrightarrow & sup(\mathcal{T}_{g'}) \end{array}$$

The diagram above illustrates how to estimate the dis-

criminative graph-features based on their frequent subtrees. Suppose g and g' are two graph-features from $\mathcal{D}(q)$ such that $g \subset g'$, we define the *occurrence probability* of g in the graph database, $\mathcal{G}$ as

$$Pr(g) = \frac{|sup(g)|}{|\mathcal{G}|} = \sigma_g \tag{9}$$

Similarly, the *conditional occurrence probability* of g', *w.r.t.* g, can be measured as

$$Pr(g'|g) = \frac{Pr(g \wedge g')}{Pr(g)} = \frac{Pr(g')}{Pr(g)} = \frac{|sup(g')|}{|sup(g)|} \tag{10}$$

$Pr(g \wedge g') = Pr(g')$ because g is a subgraph of g'. For each occurrence of g' in $\mathcal{G}$, g must occur simultaneously with g', but the reverse is not necessarily true. Here, $Pr(g'|g)$ models the probability to select g' from $\mathcal{G}$ in the presence of g. According to Eq. (10), if $Pr(g'|g)$ is small, g' has a high probability to be discriminative, *w.r.t.* g. However, it is still impractical to calculate $Pr(g'|g)$ based on Eq. (10), because the exact values of $sup(g)$ and $sup(g')$ are unknown yet. As illustrated above, instead, we estimate $Pr(g'|g)$ by making use of $\mathcal{T}(g)$ and $\mathcal{T}(g')$. Note: all $\mathcal{T}(g)$ ($\subseteq \mathcal{F}_T$), $\mathcal{T}(g')$ ($\subseteq \mathcal{F}_T$), and the entire frequent tree-feature set ($\mathcal{F}_T$) are known already. Below, we give the details, and discuss the tight upper and lower bound of $Pr(g'|g)$, based on $\mathcal{T}(g)$ and $\mathcal{T}(g')$.

Since g and g' are both frequent, the following inequalities hold,

$$|sup(g)| \geq \sigma|\mathcal{G}| \quad \text{and} \quad |sup(g')| \geq \sigma|\mathcal{G}| \tag{11}$$

Since g and g' are both discriminative ($g, g' \in \mathcal{D}(q)$), the following inequalities holds,

$$\epsilon(g) \geq \epsilon_0 \quad \text{and} \quad \epsilon(g') \geq \epsilon_0 \tag{12}$$

Based on Eq. (6), Eq. (7) and Eq. (8), we translate the above inequality Eq. (12) to $|sup(g)|$ ($|sup(g')|$) by an expression of $|sup(\mathcal{T}(g))|$ ($|sup(\mathcal{T}(g'))|$) and ϵ_0,

$$|sup(g)| \leq |\mathcal{G}| - \frac{|\mathcal{G}| - |sup(\mathcal{T}(g)|)}{1 - \epsilon_0} \tag{13}$$

$$|sup(g')| \leq |\mathcal{G}| - \frac{|\mathcal{G}| - |sup(\mathcal{T}(g')|)}{1 - \epsilon_0} \tag{14}$$

Based on Eq. (11) and Eq. (14), we derive the upper bound of $Pr(g'|g)$ which is solely relied on $\mathcal{T}(g')$, constant factors σ and ϵ_0, as shown in Eq. (15). Here, $\sigma_x = |sup(x)|/|\mathcal{G}|$, where x is a frequent feature (or a set of frequent features) of $\mathcal{G}$.

$$Pr(g'|g) = \frac{|sup(g')|}{|sup(g)|} \leq \frac{|\mathcal{G}| - \frac{|\mathcal{G}| - |sup(\mathcal{T}(g')|)}{1-\epsilon_0}}{\sigma|\mathcal{G}|} = \frac{\sigma_{\mathcal{T}(g')} - \epsilon_0}{(1-\epsilon_0)\sigma} \tag{15}$$

Similarly, based on Eq. (12) and Eq. (13), we derive the lower bound of $Pr(g'|g)$ which is solely relied on $\mathcal{T}(g)$, constant factors σ and ϵ_0, as shown in Eq (16).

$$Pr(g'|g) = \frac{|sup(g')|}{|sup(g)|} \geq \frac{\sigma|\mathcal{G}|}{|\mathcal{G}| - \frac{|\mathcal{G}| - |sup(\mathcal{T}(g)|)}{1-\epsilon_0}} = \frac{\sigma(1-\epsilon_0)}{\sigma_{\mathcal{T}(g)} - \epsilon_0} \tag{16}$$

Since $Pr(g'|g)$ is a probability definition, i.e., $0 \leq Pr(\cdot) \leq 1$, we have the following restrictions for $\mathcal{T}(g)$ and $\mathcal{T}(g')$

$$\sigma_{\mathcal{T}(g)} \geq max\{\epsilon_0, \sigma + (1-\sigma)\epsilon_0\} \tag{17}$$

Algorithm 3 *SelectGraph* ($\mathcal{G}$, q)

Input: A graph database $\mathcal{G}$, a non-tree query graph q
Output: The selected discriminative graph set $\mathcal{D} \subseteq \mathcal{D}(q)$

1: $\mathcal{D} \leftarrow \emptyset$;
2: $\mathcal{C} \leftarrow \{c_1, c_2, \cdots, c_n\}, c_i \subseteq q$, c_i is a simple cycle;
3: **for all** $c_i \in \mathcal{C}$ **do**
4: $\quad g \leftarrow g' \leftarrow c_i$;
5: $\quad$ **while** $size(g') \leq maxL$ **do**
6: $\quad\quad$ **if** $g \notin \Delta$ **then** $\mathcal{D} \leftarrow \mathcal{D} \cup \{g\}$;
7: $\quad\quad g' \leftarrow g' \diamond v$;
8: $\quad\quad$ **if** $\mathcal{T}(g), \mathcal{T}(g')$ satisfy Eq. (17), Eq. (18), Eq. (19) **and** $(\sigma_{\mathcal{T}(g')} < \sigma^* \times \sigma_{\mathcal{T}(g)})$ **then**
9: $\quad\quad\quad g \leftarrow g'$;
10: scan $\mathcal{G}$ to compute $sup(g)$ for every $g \in \mathcal{D}$ and add an index entry for g in Δ, if needed;
11: **return** $\mathcal{D}$;

$$max\{\epsilon_0, \sigma\} \leq \sigma_{\mathcal{T}(g')} \leq \sigma + (1-\sigma)\epsilon_0 \tag{18}$$

and

$$(\sigma_{\mathcal{T}(g)} - \epsilon_0)(\sigma_{\mathcal{T}(g')} - \epsilon_0) \geq [\sigma(1-\epsilon_0)]^2 \tag{19}$$

Our discovery is expressed in Eq. (15): the conditional occurrence probability of $Pr(g'|g)$, is solely upper-bounded by $\mathcal{T}(g')$. Therefore, to select g' from $\mathcal{D}(q)$ in the presence of g is equivalent to meet the qualification of $\sigma_{\mathcal{T}(g')}$ which drops under a specific threshold related to g. In real applications, we can just test whether the inequality $\sigma_{\mathcal{T}(g')} < \sigma^* \times \sigma_{\mathcal{T}(g)}$ satisfies or not. So the costly computation of $Pr(g'|g)$ is successfully translated to an approximate estimation toward $\mathcal{T}(g')$. Note all frequent tree-features are discovered and indexed from $\mathcal{G}$, so $\mathcal{T}(g)$ and $\mathcal{T}(g')$ can be computed efficiently. The diagram below summarizes the whole estimation process.

$$\begin{array}{ccc} sup(g)(?) & \xrightarrow{\quad ? \quad} & sup(g')(?) \\ {\scriptstyle \epsilon(g) \geq \epsilon_0}\uparrow & & \uparrow{\scriptstyle \epsilon(g') \geq \epsilon_0} \\ sup(\mathcal{T}_g) & \xrightarrow[|sup(\mathcal{T}(g'))| \geq \sigma|\mathcal{G}|]{|sup(\mathcal{T}(g))| \geq \sigma|\mathcal{G}|} & sup(\mathcal{T}_{g'}) \end{array}$$

4.2 Graph Selection Algorithm

The graph selection algorithm is outlined in Algorithm 3. Let the input query be a non-tree graph q, the algorithm selects discriminative graphs from $\mathcal{D}(q)$ based on the selection criteria discussed in Section 4.1. The algorithm initiates from simple cycles of q (line 2), which can be selected from q efficiently. After a simple cycle c is selected, we extend c by growing one vertex (and all its corresponding edges) each time to get a larger-size graph g'(denoted $\diamond$ in line 7). If the conditions hold (line 8), g' will be selected from $\mathcal{D}(q)$. Finally, in line 10, the algorithm compute $sup(g)$ for every $g \in \mathcal{D}$ and add g as an indexing feature into Δ.

There are several implications in Algorithm 3. First, given a simple cycle, $c \subseteq q$, all c's subtrees are paths in nature. According to Theorem 3.1 and Theorem 3.2, the simple cycle c is usually discriminative *w.r.t.* its subpath feature set. Therefore, it is reasonable to consider all simple cycles of q as the starting point of our discriminative graph selection algorithm. Second, a maximum feature size, $maxL$, is set

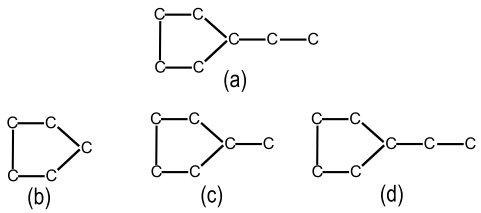


Figure 8: A non-tree query graph q and the discriminative graph-feature selection upon q

such that only discriminative graph features within a specific size are selected. And the pattern-growth process for each simple cycle c continues at most $(maxL - size(c))$ times (line 7). Third, if $\mathcal{T}(g')$ is not frequent in the condition test (line 8), both g' and all g''s subsequent supergraphs must be infrequent, *i.e.*, the while loop can be early terminated. Fourth, when a discriminative graph g has already been selected and indexed into Δ by previous queries, there is no need to select g for multiple times (line 6). Actually, all the indexed discriminative graph features can be shared by subsequent queries.

Figure 8 (a) presents a non-tree query q submitted to the sample graph database $\mathcal{G}$, shown in Figure 1. The minimum support threshold σ is set 2/3. The discriminative ratio ϵ_0 is set 1/2 and $\sigma^* = \sigma = 2/3$. Following Algorithm 3, the simple cycle c in Figure 8 (b) is selected as the starting point for graph-feature selection. Based on Figure 7, c is discriminative *w.r.t.* all its frequent subtrees. Figure 8 (c) is generated by extending one vertex of q upon c, which is denoted as g. For c and g, $\sigma_{\mathcal{T}(c)} = 1$, $\sigma_{\mathcal{T}(g)} = 2/3$, which satisfies the constraints expressed in Eq. (17), Eq. (18) and Eq. (19). However, $2/3 \times \sigma_{\mathcal{T}(c)} = 2/3 = \sigma_{\mathcal{T}(g)}$, *i.e.*, it does not satisfy the constraint: $\sigma_{\mathcal{T}(g)} < \sigma^* \times \sigma_{\mathcal{T}(c)}$. So g is not selected from $\mathcal{D}(q)$ as a discriminative graph feature. Similarly, Figure 8 (d) is generated by extending one vertex of q upon g, which is denoted as g'. Since $\mathcal{T}(g') = 1/3 < \sigma$, *i.e.*, g' is infrequent in $\mathcal{G}$, which disobeys the constraint expressed in Eq. (18), g' is not selected from $\mathcal{D}(q)$, either.

5. IMPLEMENTATIONS

In this section, we give implementation details of our cost-effective graph indexing algorithm, (Tree+Δ). We present data structures and design principles of (Tree+Δ) from the perspectives of index construction and query processing.

In order to construct (Tree+Δ), we mine frequent tree-features from the graph database, $\mathcal{G}$. There exist many reported studies on frequent structural pattern mining over large graph databases such as `gSpan` [22], `Gaston` [17], `Hybrid -TreeMiner` [5], etc. We proposed a fast frequent free tree mining algorithm [24] which takes full advantage of characteristics of tree structure. Once the frequent tree feature set $\mathcal{F}_T$ is selected from $\mathcal{G}$, every tree $t \in \mathcal{F}_T$ is sequentialized and maintained in a hash table. Every $t \in \mathcal{F}_T$ is associated with its support set $sup(t)$, which contains the ids of graphs in $\mathcal{G}$ containing t as subgraph(s). With the aid of the hash table, every frequent tree feature and its support set can be located and retrieved quickly. For index maintenance, the similar strategies discussed in [23] can be adapted.

The query processing of (Tree+Δ) is outlined in Algorithm 4 with three parameters: a query graph q, a graph database G, and the index built on frequent tree-features of $\mathcal{G}$. Below, we use $\mathcal{F}_T$ to denote the index composed

Algorithm 4 *Query Processing* (q, $\mathcal{F}_T$, $\mathcal{G}$)

Input: A query graph q, the frequent tree-feature set $\mathcal{F}_T$, and the graph database $\mathcal{G}$
Output: Candidate answer set C_q

1: $\mathcal{D} \leftarrow \emptyset$;
2: $\mathcal{T}(q) \leftarrow \{t \mid t \subseteq q, t \in \mathcal{F}_T, size(t) \leq maxL\}$;
3: $C_q \leftarrow \bigcap_{t \in \mathcal{T}(q)} sup(t)$;
4: **if** $(C_q \neq \emptyset)$ **and** (q is cyclic) **then**
5: $\quad \mathcal{D} \leftarrow SelectGraph(\mathcal{G}, q)$;
6: $\quad$ **for all** $(g \in \mathcal{D})$ **do**
7: $\quad\quad C_q \leftarrow C_q \bigcap sup(g)$;
8: **return** C_q;

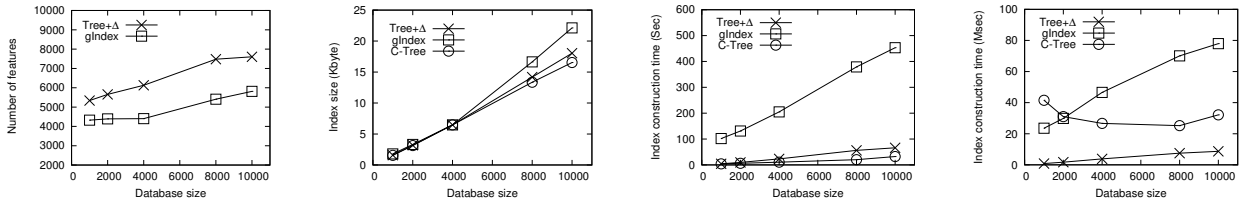
of frequent tree-features. While the selected discriminative graph-features are maintained in the additional index structure, Δ, which is handled by *SelectGraph* (Algorithm 3).

In Algorithm 4, (Tree+Δ) enumerates all frequent subtrees of q up to the maximum feature size $maxL$ which are located in the index $\mathcal{F}_T$ (line 2). Based on the obtained frequent subtree feature set of q, $\mathcal{T}(q)$, the algorithm computes the candidate answer set, C_q, by intersecting the support set of t, for all $t \in \mathcal{T}(q)$ (line 3). If q is a non-tree cyclic graph, it calls *SelectGraph* to obtain a set of discriminative graph-features, $\mathcal{D}$ (line 5). Those discriminative graph-features may be cached in Δ already. If not, *SelectGraph* will reclaim them from the graph database and maintain them in Δ. Then the algorithm further reduces the candidate answer set C_q by intersecting the support set of g, for all $g \in \mathcal{D}$ (line 6-7).

The pseudo-code in Algorithm 4 is far from optimized. At line 2, If a tree-feature $f_i \subseteq q$ does not appear in $\mathcal{F}_T$, it implies that f_i is infrequent, so there is no need to consider $f_j \subseteq q$ if $f_i \subseteq f_j$, due to the Apriori principle. At line 3 and line 7, the candidate answer set C_q is obtained by intersecting support sets of frequent trees and discriminative graphs of q. However, it is unnecessary to intersect every frequent feature derived from q. Given a series of frequent features $f_1, f_2, \cdots, f_n \subseteq q$, if $f_1 \subset f_2 \subset \cdots \subset f_n$, then $C_q \bigcap sup(f_1) \bigcap sup(f_2) \bigcap \cdots \bigcap sup(f_n) = C_q \bigcap sup(f_n)$. So only the maximum frequent features are considered when computing the candidate answer set. More formally, Let $F_m(q)$ be the set of maximum frequent features of q, i.e., $F_m(q) = \{f | f \subseteq q, \nexists f' \subseteq q, s.t., f \subset f'\}$. In order to compute C_q, we only need to perform intersection operations on the support sets of maximum frequent features in $F_m(q)$, which substantially facilitates the whole graph containment query processing.

For graph isomorphism testing in line 5, every discriminative graph g is encoded to a *canonical code*, $cc(g)$. Two graphs g and g' are isomorphic to each other, if and only if $cc(g) = cc(g')$. There exist several canonical codings for a general graph, such as *CAM* [11], *DFS-code* [22] etc. We use *CAM* as canonical code of discriminative graphs in our implementation.

Another issue to be concerned is the selection and tuning of different parameters: the minimum support threshold, σ; the minimum discriminative ratio, ϵ_0, and the discriminative graph selection threshold, σ^*. For σ, it has a close correlation with $|\mathcal{F}|$, $\mathcal{C}_{FS}$ and $|C_q|$. When σ is set small, the number of frequent tree-features discovered from $\mathcal{G}$ grows exponentially, which inevitably enhances the feature selection cost,



(a) Feature Size (b) Index Size (c) Construction Time (d) Average Time

Figure 9: Index Construction on The Real Dataset

$\mathcal{C}_{FS}$. In the mean time, the feature space, $\mathcal{F}$, is enlarged accordingly while $|C_q|$ might be reduced because of more indexing features considered. So, σ should be determined on a deliberate balance between time and space. As to ϵ_0 and σ^*, they both correlate with the filtering cost, $\mathcal{C}_f$ and $|C_q|$. Setting a loose bound for the discriminative graph selection (small ϵ_0 and large σ^*) results in more graph-features to be reclaimed from $\mathcal{G}$, which increases $\mathcal{C}_f$, whereas $|C_q|$ probably decreases for more discriminative graph-features are indexed in Δ. Since the number of discriminative graph-features held in Δ is fairly small *w.r.t.* the tree-based index size, $|\mathcal{F}_T|$, the space overhead of Δ can be negligible. A practical way to determine ϵ_0 and σ^* is to sample a small portion of $\mathcal{G}$ and select discriminative graphs *w.r.t* their subtrees which well reflect the characteristics of the whole graph database, and ϵ_0 and σ^* are tuned accordingly.

6. RELATED WORK

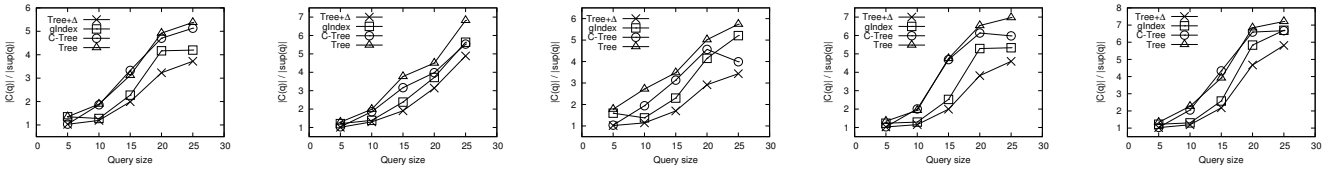
There is a wealth of literature concerning the graph containment query problem. Among them emerges a special case dealing with queries on semi-structured databases, especially for XML databases [14, 19]. The data object of XML databases is abstracted as rooted labeled graph, a special kind of general graph. Regular path expressions are used to represent substructures in the database. To avoid unnecessary traversals on the database during the evaluation of a path expression, indexing methods are introduced [15, 16]. In content-based image retrieval, Petrakis and Faloutsos [18] represented each graph as a vector of features and indexed graphs in high dimensional space using R-trees. Instead of casting a graph to a vector form, Berretti *et al.* [4] proposed a metric indexing scheme which organizes graphs hierarchically according to their mutual distances. The SUBDUE system developed by Holder *et al.* [10] used minimum description length to discover substructures that compress the database and represent structural concepts in the data. However, these systems are designed to address the exact matching problem between graphs in the database and a query, which is a special case of the graph containment query problem.

One solution to graph containment query is to index paths in graph databases, and this solution is often referred to as the *path-based* indexing approach. `GraphGrep` [20] is a famous representative of this approach. `GraphGrep` enumerates all existing paths up to a certain length l_p in a graph database $\mathcal{G}$ and selects them as indexing features. The feature selection can be done efficiently, but the index size can be large, if l_p is not small. A graph containment query is answered in two phases: the first *filtering phase* selects a set of candidate graphs from $\mathcal{G}$ in which the number of each indexed path-feature is at least that of the query. The second *verification phase* verifies each graph in the candidate answer set derived from the first phase, as opposed to $\mathcal{G}$, by subgraph isomorphism testing. False positives are discarded and the true answer set is returned.

Path-based indexing approach have two main disadvantages. First, the index structure is usually huge when $|\mathcal{G}|$ is large or graphs in $\mathcal{G}$ are large and diverse. For example, by randomly selecting $10,000$ graphs from the AIDS antiviral screening database and artificially setting $l_p = 10$, the number of path features is more than $1000,000$, most of which are redundant based on human observation. It will be inefficient to index all of them. Second, path-features have limited pruning power. In other words, the candidate answer set generated in the filtering phase can be considerably large, even when the number of path-features is large. This is mainly because the structural information exhibited in graphs is lost when breaking graphs into paths.

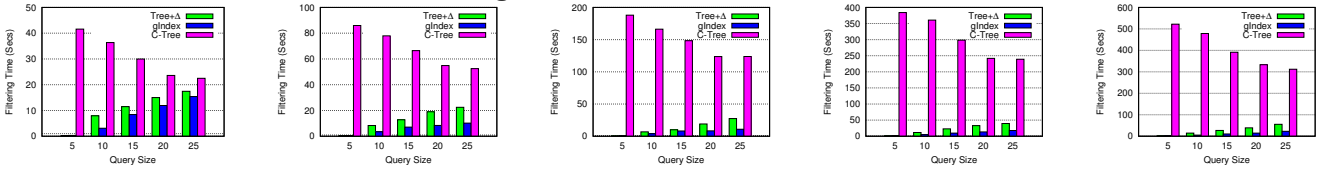
In comparison to the path-based indexing approach, there exists another mechanism using graphs as basic indexing features, which is often referred to as *graph-based* indexing approach. A distinguished example of this approach is `gIndex` [23]. `gIndex` takes advantage of a graph mining procedure to discover frequent graphs from $\mathcal{G}$, by which the index is constructed. In order to scale down the exponential number of frequent graphs, `gIndex` selects only *discriminative* ones as indexing features. `gIndex` has several advantages over `GraphGrep`. First, structural information of graph is well preserved, which is critical to filter false positives in the verification phase; Second, the number of discriminative frequent graph-features is much smaller than path-features, so that the index is compact and easy to be accommodated in main memory; Third, discriminative frequent graphs are relatively stable to database updates, which makes incremental index maintenance feasible. Experimental results show that `gIndex` has 10 times smaller index size than that of `GraphGrep`, and outperforms `GraphGrep` by $3-10$ times in terms of the candidate answer set size.

The disadvantages of `gIndex` are obvious. First, because index construction is a time-consuming graph mining procedure, the computationally expensive (sub)graph isomorphism testings are unavoidable. The index construction cost can be even high when $|\mathcal{G}|$ is large or graphs in $\mathcal{G}$ are large and diverse. Second, `gIndex` assumes that discriminative frequent graphs discovered from $\mathcal{G}$ are most likely to appear in query graphs, too. However, since a user may submit various queries with arbitrary structures, it is much more valuable to index common structures of query graphs than those of $\mathcal{G}$. If most subgraphs of a query are luckily indexed, `gIndex` can return the answer efficiently. Otherwise, the query performance can be deteriorated because few indexing features can be used, needless to say there may be some trivial features, such as vertices, edges or simple paths, which contributes little to pruning. Therefore it is both costly and unnecessary to discover a complete set of discriminative frequent graphs.



(a) N=1000 (b) N=2000 (c) N=4000 (d) N=8000 (e) N=10000

Figure 10: False Positive Ratio



(a) N=1000 (b) N=2000 (c) N=4000 (d) N=8000 (e) N=10000

Figure 11: Filtering Cost

`C-Tree` [9] is another graph-based indexing mechanism using *graph closure* as indexing features. The graph closure is a "bounding box" of constituent graphs which contains discriminative information of their descendants. Although the costly graph mining is avoided in index construction, `C-Tree` shares similar disadvantages to `gIndex`, as mentioned above.

7. EXPERIMENTAL STUDY

In this section, we report our experimental studies that validate the effectiveness and efficiency of our (Tree+Δ) algorithm. (Tree+Δ) is compared with `gIndex` and `C-Tree`, two up-to-date graph-based indexing algorithms. We use two kinds of datasets in our experiments: one real dataset and a series of synthetic datasets. Most of our experiments have been performed on the real dataset since it is the source of real demand. All our experiments are performed on a 3.4GHz Intel PC with 2GB memory, running MS Windows XP and Redhat Fedora Core 4. All algorithms of (Tree+Δ) are implemented in C++ using the MS Visual Studio compiler.

7.1 AIDS Antiviral Screen Dataset

The experiments described in this section use the antiviral screen dataset from the Developmental Theroapeutics Program in NCI/NIH[1]. This 2D structure dataset contains 42390 compounds retrieved from DTP's Drug Information System. There are total 63 kinds of atoms in this dataset, most of which are C, H, O, S, *etc.* Three kinds of bonds are popular in these compounds: single-bond, double-bond and aromatic-bond. We take atom types as vertex labels and omit edge labels because `C-Tree` does not support edge-labeled graphs. On average, compounds in the dataset has 43 vertices and 45 edges. The graph of maximum size has 221 vertices and 234 edges.

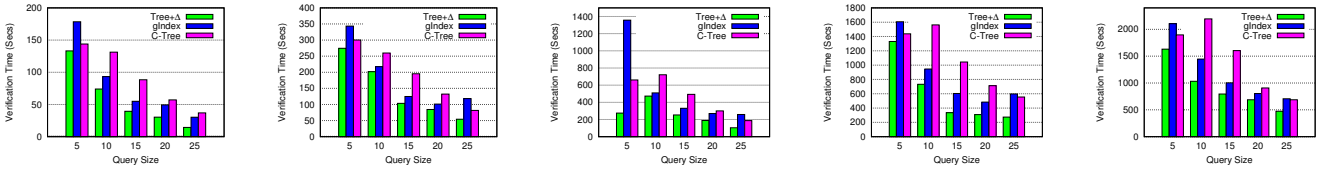
We set the following parameters in (Tree+Δ), `gIndex` and `C-Tree` for our experimental studies. In (Tree+Δ) and `gIndex`, the maximum feature size $maxL$ is set 10. For (Tree+Δ), the minimum discriminative ratio ε_0 is set 0.1; the minimum support threshold σ is set 0.1, and σ^* is set 0.8 for discriminative graph selection during query processing. In `C-Tree`, we set the minimum number of child vertices $m = 20$ and the maximum number $M = 2m - 1$. We use the NBM method to compute graph closures. All these experimental settings are identical to author-specified values in [9, 23].

The first test is on index size $|\mathcal{F}|$ and index construction cost $\mathcal{C}_{FS}$ of three different indexing algorithms: (Tree+Δ), `gIndex` and `C-Tree`. The test dataset consists of N graphs, which is randomly selected from the antivirus screen database. Figure 9 (a) depicts the number of frequent features indexed in (Tree+Δ) and `gIndex` with the test dataset size N varied from 1,000 to 10,000 (`C-Tree` does not provide the explicit number of indexing features, so we omit `C-Tree` in this experiment). The curves clearly show that frequent features of (Tree+Δ) and `gIndex` are comparable in quantity. Figure 9 (b) illustrates the index size, *i.e.* $|\mathcal{F}|$ of three different indexing algorithms. Although (Tree+Δ) has more indexing features than `gIndex`, the memory consumption for storing trees are much less than that for storing graphs held in `gIndex` and `C-Tree`. The curves illustrate that (Tree+Δ) has a compact index structure which can easily be held in main memory. In Figure 9(c), we test the index construction time, *i.e.*, $\mathcal{C}_{FS}$, measured in seconds. Both (Tree+Δ) and `C-Tree` outperforms `gIndex` by an order of magnitude, while (Tree+Δ) and `C-Tree` have similar index construction costs. Finally, the index construction time is averaged for each frequent feature and shown in Figure 9(d), measured in milliseconds. As the figure illustrates, for each indexing feature, $\mathcal{C}_{FS}$ of (Tree+Δ) is much smaller than that of `gIndex` and `C-Tree`.

Having verified the index size $|\mathcal{F}|$ and index construction cost $\mathcal{C}_{FS}$ of `gIndex`, `C-Tree` and (Tree+Δ), we now evaluate their query performances. Given a query graph q, the query cost is characterized by the candidate answer set size, $|C_q|$. Since $|sup(q)|$ is the tight lower bound of $|C_q|$, an algorithm achieving this lower bound can match the queries in the graph database precisely. We denote $|C(q)|/|sup(q)|$, the *false positive ratio*, as a measure for the pruning ability of different graph indexing algorithms. The smaller the false positive ratio, the better pruning ability an indexing algorithm has.

We select our dataset by randomly sampling graphs from the antivirus screen database with size varying from 1,000 to 10,000. Five query sets are tested, each of which has 1,000 queries. The query sets are generated from the antivirus screen database by randomly selecting 1,000 graphs and then extracting connected subgraphs (subtrees, subpaths) from them. We remove vertices with the smallest degree

[1]http://dtp.nci.nih.gov/docs/aids/aids_data.html



(a) N=1000 (b) N=2000 (c) N=4000 (d) N=8000 (e) N=10000

Figure 12: Verification Time

from a graph and this process proceeds until we get a query with a specified size. If all vertices of a graph have the same degree, one of them are removed to break the tie and the vertex-deletion procedure continues. This query generation approach secures that query graphs are connected and diverse enough to include different kind of graph structures(path, nonlinear tree and cyclic graph). We generate Q_5, Q_{10}, Q_{15}, Q_{20} and Q_{25}, with the digits denoting the query size.

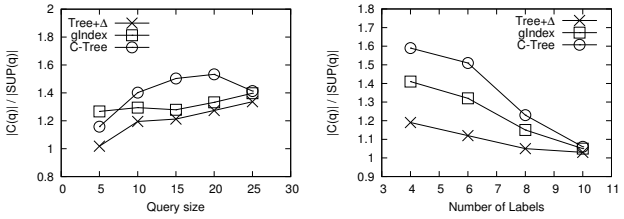
Figure 10 presents the false positive ratios of three graph indexing algorithms on different databases. Additionally, we use only frequent trees of $\mathcal{G}$, *i.e.*, $\mathcal{F}_T$ as indexing features and denote this indexing algorithms as `Tree`. `Tree` is similar to (Tree+Δ) but does not find discriminative graphs back from $\mathcal{G}$ during query processing. As shown in the figure, In all experimental settings and for every graph query set, (Tree+Δ) outperforms `gIndex`, `C-Tree` and `Tree`. When the query size is small, the filtering power of four algorithms is close to each other and when the graph database becomes large and the query size increases, the advantages of (Tree+Δ) become more apparent, *i.e.*, (Tree+Δ) performs especially well on large graph databases when query graphs are large and diverse. Interestingly, the gap between false positive ratios of `gIndex` and `Tree` is not large. It means that the pruning power of tree-feature is close to that of graph-feature, but tree is still less powerful than graph for pruning purpose. This evidence well justifies our analysis on pruning ability of different frequent features in $\mathcal{F}$, as mentioned in Section 3. Meanwhile, it also proves that discriminative graphs play a key role in false positive pruning, and there is a good reason for us to find them back from the graph database $\mathcal{G}$ during query processing.

Figure 11 illustrates the filtering cost $\mathcal{C}_f$ of three different graph indexing algorithms, measured in seconds. As shown in the figure, in all different databases, $\mathcal{C}_f$ of `C-Tree` is much larger than (Tree+Δ) and `gIndex`. Meanwhile, $\mathcal{C}_f$ of (Tree+Δ) is larger than `gIndex` because when queries become large and complex, (Tree+Δ) has to select discriminative graphs and reclaim them from the graph database, if the discriminative graph-features are not held in Δ. When query is fairly small (Q_5), there is little chance for a query graph to contain cycles, so $\mathcal{C}_f$ of (Tree+Δ) is smaller than that of `gIndex`. It also demonstrates that (Tree+Δ) is quite efficient to answer acyclic graph queries.

Figure 12 presents the verification cost of three graph indexing algorithms, *i.e.*, the factor $|C_q| \times \mathcal{C}_v$ of the query cost model in Section 2.3. As illustrated in the figure, (Tree+Δ) needs less time to remove false positives from the candidate answer set because $|C_q|$ is smaller than that obtained by `gIndex` and `C-Tree`. Based on Figure 11 and Figure 12, we are confirmed that (Tree+Δ) outperforms `gIndex` and `C-Tree` because our final query cost ($\mathcal{C}_f + |C_q| * \mathcal{C}_v$) is minimum, in comparison with `gIndex` and `C-Tree`, and our (Tree+Δ) shows a good scalability *w.r.t.* the database size.

7.2 Synthetic Dataset

In this section, we conduct our performance studies on a synthetic database. This database is generated by the widely-used graph generator [12] described as follows: first a set of S seed fragments are generated randomly, whose size is determined by a Poisson distribution with mean I. The size of each graph is a Poisson random variable with mean T. Seed fragment are then randomly selected and inserted into a graph one by one until the graph reaches its size. A typical database may have the following experimental settings: it has $10,000$ graphs and $1,00$ seed fragments with 5 distinct labels. On average, each graph has 50 edges and each seed fragment has 10 edges. We denote this dataset with the above settings as $D10kI10T50S100L5$.



(a) Varying Query Size (b) Varying #Labels

Figure 13: False Positive Ratio

We test the false positive ratio on the synthetic database mentioned above with 5 query sets Q_5, Q_{10}, Q_{15}, Q_{20}, Q_{25}, which are generated using the same method described in Section 7.1. As shown in Figure 13(a), the filtering power presented by (Tree+Δ) is still better than that provided by `gIndex` and `C-Tree`.

We then test the influence of vertex labels on the query performance of different indexing algorithms. When the number of distinct labels is large, the synthetic dataset is much different from the real dataset. Although local structural similarity appears in different graphs, there is little similarity existing among each graph. This characteristic results in a simpler index structure. For example, if every vertex in one graph has a unique label, we only use vertex labels as index features. This is similar to the inverted index technique in information retrieval. In order to verify this conclusion, we vary the number of labels from 5 to 10 in the synthetic dataset $D10kI10T50S100$ and test the false positive ratios of three algorithms using the query set Q_{10}. Figure 13(b) shows that they are close with each other when L is growing large.

8. CONCLUSIONS

Graph indexing plays a critical role in graph containment query processing on large graph databases which have gained increasing popularity in bioinformatics, Web analysis, pattern recognition and other applications involving graph structures. Previous graph indexing mechanisms take

paths and *graphs* as indexing features and suffer from overly large index size, substantial index construction overhead and expensive query processing cost. In this paper, we have proposed a cost-effective graph indexing mechanism for addressing the graph containment query problem: index based on frequent tree-features. We analyze the effectiveness and efficiency of tree as indexing feature, *w.r.t.* path and graph from three critical perspectives: feature size $|\mathcal{F}|$, feature selection cost $\mathcal{C}_{FS}$ and feature pruning power $|C_q|$. In order to achieve better pruning ability than path-based and graph-based indexing mechanisms, we deliberately select a small portion of discriminative graph-features on demand, which consist of our additional index structure Δ related only to query graphs. Our analysis and performance studies confirm that the proposed graph indexing algorithm, (Tree+Δ), is a better choice than up-to-date path-based or graph-based indexing algorithms. (Tree+Δ) holds a compact index structure, achieves good performance in index construction and most importantly, provides satisfactory query performance for answering graph containment queries over large graph databases.

9. ACKNOWLEDGMENTS

We thank Prof. Jiawei Han and Dr. Xifeng Yan for providing their valuable comments and `gIndex` code, Dr. Huahai He and Prof. Ambuj Singh for providing `C-Tree` code, and Dr. Michihiro Kuramochi and Prof. George Karypis for providing the synthetic graph data generator.

This work was supported by a grant of RGC, Hong Kong SAR, China (No. 418206).

10. REFERENCES

[1] National Library of Medicine. http://chem.sis.nlm.nih.gov/chemidplus.

[2] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. 1974.

[3] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[4] S. Berretti, A. D. Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(10):1089–1105, 2001.

[5] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok. Frequent subtree mining - an overview. *Fundam. Inf.*, 66(1-2):161–198, 2005.

[6] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM'99*, pages 251–262, 1999.

[7] K. S. Fu. A step towards unification of syntactic and statistical pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(3):398–404, 1986.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[9] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 38, 2006.

[10] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proceedings AAAI'94 of the Workshop on Knowledge Discovery in Databases (KDD'94)*, 1994.

[11] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM'03)*, page 549, 2003.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, pages 313–320, 2001.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 10th European software engineering conference (ESEC/FSE'05)*, pages 286–295, 2005.

[14] L.Sheng, Z.M.Ozsoyoglu, and G.Ozsoyoglu. A graph query language and its query processing. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 572–581, 1999.

[15] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[16] T. Milo and D. Suciu. Index structures for path expressions. In *Proceeding of the 7th International Conference on Database Theory (ICDT'99)*, pages 277–295, 1999.

[17] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'04)*, pages 647–652, 2004.

[18] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):435–447, 1997.

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th Very Large Data Bases (VLDB'99)*, pages 302–314, 1999.

[20] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'02)*, pages 39–52, 2002.

[21] S. Srinivasa and S. Kumar. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. In *Proc. of Very Large Data Bases (VLDB'03)*, pages 975–986, 2003.

[22] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 721, 2002.

[23] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD'04)*, pages 335–346, 2004.

[24] P. Zhao and J. X. Yu. Fast frequent free tree mining in graph databases. In *Proceedings of the 2nd International Workshop on Mining Complex Data (MCD'6)*, 2006.