Iterative Graph Feature Mining for Graph Indexing

Dayu Yuan*[1], Prasenjit Mitra*†[2], Huiwen Yu*[3], C. Lee Giles*†[4]
*Department of Computer Science and Engineering
†College of Information Sciences and Technology
Pennsylvania State University
Email:[1]duy113@psu.edu, [2]pmitra@ist.psu.edu, [3]huy117@psu.edu, [4]giles@ist.psu.edu

***Abstract*—Subgraph search is a useful and challenging query scenario for graph databases. Given a query graph q, a subgraph search algorithm returns all database graphs having q as a subgraph. To efficiently implement a subgraph search, *subgraph features* are mined in order to index the graph database. Many subgraph feature mining approaches have been proposed. They are all "mine-at-once" algorithms in which the whole feature set is mined in one run before building a stable graph index. However, due to the change of environments (such as an update of the graph database and the increase of available memory), the index needs to be updated to accommodate such changes. Most of the "mine-at-once" algorithms involve frequent subgraph or subtree mining over the whole graph database. Also, constructing and deploying a new index involves an expensive disk operation such that it is inefficient to re-mine the features and rebuild the index from scratch.**

We observe that, under most cases, it is sufficient to update a small part of the graph index. Here we propose an "iterative subgraph mining" algorithm which iteratively finds one feature to insert into (or remove from) the index. Since the majority of indexing features and the index structure are not changed, the algorithm can be frequently invoked. We define an objective function that guides the feature mining. Next, we propose a basic branch and bound algorithm to mine the features. Finally, we design an advanced search algorithm, which quickly finds a near-optimum subgraph feature and reduces the search space. Experiments show that our feature mining algorithm is 5 times faster than the popular graph indexing algorithm gIndex, and that features mined by our iterative algorithm have a better filtering rate for the subgraph search problem.

I. Introduction

Graph data has grown steadily in various scientific and commercial areas. Chemical molecules [1], proteins [2] and three-dimensional mechanical parts [3] are modeled as graphs. Graphs also have broad applications in such areas as computer vision and image processing [4].

Subgraph Search is one of the most popular graph retrieval models. In a graph dataset D, given a query graph q, a subgraph search algorithm retrieves all graphs in D containing q as a subgraph. Subgraph query processing is nontrivial because deciding if one graph is a subgraph of another, also referred to as the subgraph isomorphism problem, is shown to be NP-complete [5]. To solve the subgraph search problem, subgraph (subtree) features are commonly mined using several methods to build a graph index [6]–[11]. As shown in Figure 1, in a 8-graph dataset, three subgraph features are mined to build a graph index. Given a query q containing features p_1 and p_3, any supergraph of q should have both p_1 and p_3 as subgraphs. Therefore, only graphs $\{g_1, g_2\} = \{g_1, g_2, g_6, g_8\} \cap \{g_1, g_2, g_3, g_5, g_7\}$ are candidate graphs that need to be evaluated with subgraph isomorphism tests, and all the other database graphs are directly filtered out. The query processing time depends upon the number of subgraph isomorphism tests, which, in turn, depends on the filtering power of the feature set. As such an important aspect is a choice of good features. Many features, such as frequent and discriminative subgraph (subtree) features [6], [9]–[11], $\delta-$TCFG features [7] and MimR (maximum information and minimum redundancy) features [8] are mined to build the graph index and have certain significant filtering capabilities. However, these algorithms are all *mine-at-once algorithms*: they first mine the whole feature set and then use those features to construct a stable index. For these algorithms the index structure can only be slightly changed, otherwise all the features need to be re-mined and the index has to be reconstructed and deployed. Such changes are necessary when the underlying database is updated or extra memory becomes available for the index. In a recent study [12], roughly $4,000$ new structures are added into the SCI Finder database every day. Running the mine-at-once algorithms frequently to accommodate those changes is currently quite costly. In order to address the index updating problem, Zou, et al., propose a spectral coding method, $gCode$, to index graph databases without mining and using subgraph features [12]. However, Han, et al. using comprehensive comparison experiments show that gCode's filtering power is much lower than that of feature-based subgraph indexes [13].

In this paper, we introduce a light-weight subgraph feature mining algorithm suitable for mining indexing features for a dynamically updated graph database. We propose an iterative method to mine a subgraph feature and insert it into the index. Since our mining algorithm takes far less time than the

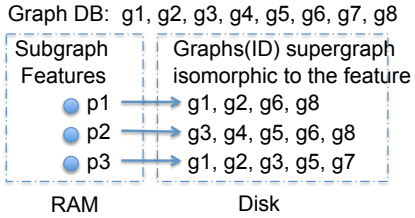


Fig. 1. Demo of subgraph query processing

existing mine-at-once algorithms, it can be called frequently to update the graph index accommodating any changes in the underlying database. Iterative mining of the graph features for classification and regression has been studied [14]–[17]. However, to the best of our knowledge, we are the first to propose an incremental, iterative, subgraph-search feature-mining algorithm.

Given a graph index I with its initial set of indexing features P_0, our mining algorithm searches for a new feature p which optimizes an objective function. This objective function is defined as the number of isomorphism tests saved over query log Q after adding this new feature p. Intuitively, the optimal feature p should be a frequent subgraph of the query log Q, infrequent in the graph database and not redundant to features in P_0. Therefore, we enumerate frequent subgraphs from the query log Q and then evaluate them. The query logs are seldom used in the previous subgraph search feature selection algorithms. However, as we show in subsection III-B that query logs are as important as the underlying graph database. The isomorphism-test saving (or filtering power) of current subgraph (tree) features, e.g., $\delta-$TCFG, depends on the query logs. Although we add training queries to model the objective function, our algorithm is general enough for any queries. We also show in subsection III-C that features selected according to the objective function are all discriminative features (irredundant to P_0). To alleviate the computation bottleneck of calculating the objective function, we further propose an upper bound and lower bound of the objective function in subsection III-D. Besides finding and inserting new subgraph features into the graph index, our proposed objective function can also be used to find and remove useless features from the index. Due to the lack of space, we do not discuss this in detail.

In order to quickly find the feature p maximizing the objective function, we propose two algorithms to prune the search space in section IV. We first derive an anti-monotone [1] branch upper bound and adopt the branch and bound paradigm to search for the optimal feature. As the branch upper bound is not tight (shown in Section IV-A), we propose the second algorithm, QueryGrouping, to further reduce the search space. QueryGrouping takes the following two steps: (1) assign the query logs Q into disjoint groups, such that graphs in each group share a common subgraph $r \in P_0$, (2) enumerate all frequent subgraphs in each group to search for an optimal feature. The query group with the longest response time are mined first. We observe that the optimal value mined from the first few groups is close to the global optimum; this is because graphs in each group are similar (sharing common subgraph r) and the queries with large candidate sets are analyzed first. Quickly identifying a near-optimal feature can help prune more branches thus reduce the search space. In addition, since the number of query graphs in one group is much less than that in the whole query log, enumerating frequent subgraphs over one small group of query graphs is much faster than doing it over the entire query log. Finally, features mined from different groups target at improving the processing of different queries and have little pair-wise redundancy, they can be inserted into the index simultaneously. Therefore, we select the top k features mined in one iteration, which further reduces the search time.

We show empirically that subgraph features generated by our iterative mining algorithm reduce the time taken to process subgraph search queries more than those mined by existing methods. Furthermore, our incremental algorithm runs significantly faster than algorithms that mine index features from scratch when there are database updates or changes in available memory.

Additionally, we propose a "mine-at-once + iterative mining" framework that alleviates the low-support frequent-subgraph-mining bottleneck. Current mine-at-once algorithms often first obtain all frequent subgraphs and then mine the indexing features out of them. To prevent missing any important feature, frequent features are mined with a very low minimum support, e.g., $0.01|D|$, although most of the selected features have much higher support. The time for frequent subgraph mining dominates the overall mining time. To overcome this bottleneck of feature mining, we first run the mine-at-once algorithm with a higher minimum support ($0.1|D|$), then run the iterative feature mining algorithm to identify the missing features that are important.

This paper is organized as follows: we introduce background and preliminaries in Section II. Then we present the objective function and prove its correctness in section III. In Section IV, we first introduce a basic branch and bound algorithm. Then we propose an advance iterative graph mining framework to reduce the search space. Results of the empirical study is given in Section V. A review of related work is given in Section VI. Finally, we conclude this paper in Section VII.

II. Preliminary

A graph $g\ (V, E, L)$ is defined on a set of vertices V and a set of edges E such that $E \subseteq V \times V$. Each node v, or edge e is associated with one label $l(u)$ or $l(e) \in L$. A *subgraph* of a graph $g(V, E, L)$ is a graph $h(V', E', L)$ whose vertex set $V' \subseteq V$, and $E' \subseteq E$. A graph g is a *supergraph* of h if h is a subgraph of g. We denote the subgraph relationship between g and h as $g \supset h$ and $h \subset g$.

Two graphs $g_1 = (V_1, E_1, L)$ and $g_2 = (V_2, E_2, L)$ are *isomorphic* to each other if there is a bijection between V_1 and V_2 that preserves labels and connectivity of the two graphs. This bijection is a mapping such that a pair of adjacent vertices u_1, v_1 in g_1 is mapped to a pair of adjacent vertices u_2, v_2 in g_2, where $L(u_1) = L(u_2)$, $L(v_1) = L(v_2)$, and $L(E(u_1, v_1)) = L(E(u_2, v_2))$, and vice-versa. A *subgraph isomorphism* between g_1 and g_2 is an isomorphism between g_1 and a subgraph of g_2. For clarity, we describe and evaluate our algorithms on labeled undirected connected graphs. However, our algorithms can be extended on other kinds of graphs via simple modifications.

[1] A function f is anti-monotone implies $x \leq y$ if and only if $f(x) \geq f(y)$. For the case of graphs, $x < y$ iff x is a subgraph of graph y.

Frequent subgraphs (subtrees) are commonly mined and used in various graph applications. In a graph database D, a subgraph sg is a *frequent subgraph* if and only if its support is greater than a pre-defined parameter, the minimum support, $\sigma|D|$. The *support* of a subgraph sg is the number of graphs $g \in D$ containing sg. The graphs containing sg in D comprise the *supporting set* of sg, $D(sg)$.

Subgraph-search queries are often processed based on a filter+verify paradigm. Given a set of graph features $P = \{p_1, p_2, ..., p_m\}$, the database graphs are first vectorized based on P. Each graph g is represented as an m dimensional vector $X_g = [x_1, x_2, ..., x_m]$, where $x_i = 1$ if $p_i \subset g$ and $x_i = 0$ otherwise. An inverted index is then built on all graphs in D where each dimension in the vector is a key in the index. Given a query q, if an indexed subgraph feature $p = q$, p's supporting set is returned directly as the answer set. Otherwise, the candidate set $C(q)$ can be derived as,

$$C(q) = \bigcap_{p_i \subset q}^{n} D(p_i) = \bigcap_{p_i \in maxSub(q,P)}^{n} D(p_i) = D(maxSub(q, P)), \tag{1}$$

where $maxSub(q, P)$ is defined as follows:

Definition 1 (Maxsub Features): Given a subgraph feature set P, the maximal subgraph features of a graph q in P is

$$\text{maxSub(q,P)} = \{p_i \in P | p_i \subset q, \nexists x \in P \text{ s.t. } p_i \subset x \subset q\}. \tag{2}$$

After obtaining the candidate set $C(q)$, a step of subgraph isomorphism test is performed to check whether the query q is contained in each candidate graph.

The time complexity of processing a query q is:

$$T_{resp}(q) = T_{filter}(q) + T_{verif}(C(q)). \tag{3}$$

Because of the subgraph isomorphism tests (which is NP-hard), the verification time $T_{verif}(C(q))$ dominates the overall response time. Our experiments over the AIDS dataset containing roughly 40 thousands chemical molecule graphs show that $T_{verif}/T_{resp} = 93.2\%$ for gIndex and 95% for FG-Index (the dataset is introduced in section V). We further make a reasonable assumption that T_{verf} is proportional to the number of candidate graphs obtained after the filtering step,

$$\underset{verif}{T}(C(q)) \propto \begin{cases} 0 & q \in P \\ |C(q)| = |D(maxSub(q, P))| & q \notin P. \end{cases} \tag{4}$$

The size of the candidate set depends on the feature set P used to build the inverted index. Because of the exponential number of possible subgraphs of all graphs in the database, not all subgraph features can be added into P. The size of P is usually confined by the size of available memory. In FG-Index [7], some features are also stored on disk, but they are mainly used for direct retrieval of answers when the query hits one of the on-disk features. The on-disk features are not used for filtering, because frequent swapping them in and out of memory involves expensive disk operations, which greatly increases the cost of index lookups. Therefore, in the following discussion, the feature set P only represents the in-memory features that are used for filtering. Again, we make a simplifying (but reasonable) assumption that the total memory consumption of the inverted index is proportional to $|P|$. In order to improve the efficiency of subgraph search under memory constraint, it is necessary to select a set of N graph features minimizing the expectation of the overall response time for all queries,

$$\begin{aligned} P &= \underset{|P|=N}{argmin} \sum_{\forall \text{ unique } q} T_{resp} \cdot Pr(q) \\ &\approx \underset{|P|=N}{argmin} \sum_{\forall \text{ unique } q} T_{verif} \cdot Pr(q) \\ &= \underset{|P|=N}{argmin} \sum_{\forall \text{ unique } q} |C(q)| \cdot Pr(q) \end{aligned} \tag{5}$$

where $Pr(q)$ is the probability that the query is isomorphic to q and $\sum_{\forall \text{ unique } q} Pr(q) = 1$. Existing mine-at-once methods take three steps to answer subgraph search queries: (1) mine subgraph features, (2) construct the index, and (3) process queries. In our iterative feature mining method, we assume that we have an initial graph index I with features P_0. Due to database **insertion, deletion**, or **increase** of the available memory, the index and features need to be updated. We run our iterative mining algorithm to find a small set of features P' and add them to the graph index I. We also remove features with limited filtering power from the index.

III. Objective Function of the Iterative Mining

In this paper, we model the iterative feature mining as an optimization problem. We first propose an objective function capturing the computational cost-saving of the isomorphism tests after adding a new feature p to the current feature set P_0. The goal of the iterative feature mining is to find a subgraph feature p maximizing the objective function.

The objective function depends on the probability distribution of the expected queries. Estimating query distribution is hard, so we take a practical approach, calculating the objective function based on a training query log. Query logs have not been widely used on the subgraph-search feature-selection problem. Current available indexing features, e.g., δ-TCFG (FG-Index) [7] and gIndex features [6], seem to be independent of the queries. However, as we show in subsection III-B, the isomorphism-test saving by using FG-Index and gIndex actually depends on the underlying queries, which demonstrates the necessity of considering the query logs. Since it is hard to get a real query work load, we use randomly generated queries in this paper. Our algorithm can be used to provide more accurate answers when real query loads are available. We also show that features selected according to the gain function are discriminative in subsection III-C.

A. Objective Function of A New Feature

Definition 2 (Iterative Graph Feature Mining): Given a graph database D and a graph index with a feature set P_0 of size $n-1$, find a new graph feature p, $p \notin P_0$ such that the

expectation of the verification cost $T_{verif}(\approx T_{resp})$, which is proportional to $\sum_{\forall \text{ unique } q} |C(q)| \cdot Pr(q)$, is minimized with the new feature set $\{p, P_0\}$ indexed, where $C(q)$ is the candidate set of the query q.

To distinguish the candidate set generated by using feature set P_0 from that by $\{p, P_0\}$, we add a new parameter $\mathcal{P}$ in $C(q, \mathcal{P})$, where $\mathcal{P}$ is the graph-feature set that is used to obtain the candidates $C(q)$.

We propose our objective function as the saving of the number of isomorphism tests (isomorphism-test saving) after bringing the new feature p into the feature set P_0:

$$gain(p, P_0) = \sum_{\forall \text{ unique } q} (|C(q, P_0)| - |C(q, \{p, P_0\})|) \cdot Pr(q). \tag{6}$$

In order to minimize the expectation of the verification cost T_{verf} ($\approx$ response time T_{resp}), the new feature should be selected as the one maximizing the isomorphism-test saving, $p = argmax\ gain(p, P_0)$.

From the frequentist point of view, $|C(q, P_0)| = |D| \cdot Pr_D(maxSub(q, P_0))$, where $Pr_D(maxSub(q))$ is the probability of graphs in the dataset D containing all features in $maxSub(q)$. Notice that $Pr(q)$ is the probability of a query graph isomorphic to q. And $Pr_D(maxSub(q)) = Pr(g \in D \text{ s.t. } g \supset maxSub(q))$. It is hard to estimate the distribution of $Pr(q)$, $Pr_D(maxSub(q, P_0))$ and $Pr_D(maxSub(q, \{p, P_0\}))$. Therefore, we take a practical approach, calculating the objective function over the graph database D and a training query log Q:

$$gain(p, P_0) = \frac{1}{|Q|} \sum_{q \in Q} [|C(q, P_0)| - |C(q, \{p, P_0\})|] \tag{7}$$

Since the objective function of a feature p only relates to the query $q = p$ and queries for which p is a maximal subgraph given that $\{p, P_0\}$ is indexed (indicated in Equation 1), we do not consider irrelevant queries while calculating the objective function. We denote queries that have a maximal subgraph p by the *minimal super queries* of p.

Definition 3 (MinSup Query): Given a query set Q and a subgraph feature $p \in P$, a graph $q \in Q$ is a minimal super query of the feature p if and only if the feature p is a maximal subgraph feature of q.

$$minSup(p, Q) = \{q \in Q | p \in maxSub(q, P)\}. \tag{8}$$

Therefore,

$$gain(p, P_0) = \frac{1}{|Q|} \sum_{q \in minSup(p,Q)} |C(q, P_0) - C(q, \{p, P_0\})| + \frac{1}{|Q|} \sum_{q \in Q} I(p = q)\, |C(q, P_0)|, \tag{9}$$

where I is the indicator function: $I(p = q) = 1$ iff p is isomorphic to q.

B. Importance of the query distribution

Our proposed objective function takes training queries Q into consideration for the purpose of estimating the expectation of the isomorphism-test saving over all possible queries. Previous works seem to be independent of the training query set Q and are generally applied to any possible queries. However, as we show in this section, FG-Index [7] has implicit assumptions on the queries. If we change the query distribution, the isomorphism-test saving of FG-Index will change significantly (from $O(|D|/log|D|)$ to $O(1)$). Similar analysis on gIndex and other algorithms draws the same conclusion (see Appendix B for details).

Statement 1 (FG-Index's isomorphism-test saving): FG-Index optimizes the worst case response time. Its isomorphism-test saving depends on query graph distribution, $Pr(q)$.

Proof: FG-Index assumes the difference between $C(q)$ and $D(q)$ is relatively small compared to $D(q)$. Hence,

$$\begin{aligned} E[\underset{resp}{T}] \approx E[\underset{verf}{T}] &\propto \sum_{\forall \text{ unique } q} |C(q)| \cdot Pr(q) \\ &\approx \sum_{\forall \text{ unique } q} |D(q)| \cdot Pr(q) \\ &= \int_1^{|D|} w \cdot Pr_Q(|D(q)| = w) dw, \end{aligned} \tag{10}$$

where $Pr_Q(|D(q)| = w)$ is the probability that the query q has support w in the graph database D. Although w is an discrete value, we approximate the summation by integration while $|D| \gg 1$.

Instead of decreasing the difference between $C(q)$ and $D(q)$, FG-Index saves computational cost by pre-calculating and storing the answer for queries with a large answer set $D(q)$, which are denoted by FG-queries. Assume that the threshold for FG-queries is set to δ, such that all queries q with $|D(q)| > \delta|D|$ are precomputed and can be answered directly. Hence, the worst case for the isomorphism tests decreases to $\delta|D|$. The total isomorphism-test saving of FG-Index is

$$\text{FGgain} = \int_{\delta|D|}^{|D|} w \cdot Pr_Q(|D(q)| = w) dw, \tag{11}$$

which actually depends on the query distribution $Pr_Q(|D(q)| = w)$. ■

We observe in our experiments that there is a power law relationship between queries and their support: the number of queries with support w equals to $a \cdot w^{-k}$ where $w \in [1, |D|]$ (see Appendix A for details). Then, the probability follows the distribution $Pr_Q(|D(q)| = w) = \frac{(k-1)w^{-k}}{1-|D|^{-k+1}}$.

$$\begin{aligned} \text{FGgain} &= \int_{\delta|D|}^{|D|} \frac{(k-1)w^{-k}}{1-|D|^{-k+1}} \cdot w dw = \frac{(k-1) \cdot |D|(\delta^{2-k} - 1)}{(k-2)(|D|^{k-1} - 1)} \\ &< E(w) = \frac{(k-1)(|D|^k - |D|^2)}{(k-2)(|D|^k - |D|)}. \end{aligned} \tag{12}$$

The above isomorphism-test saving depends largely on k. Assume $1 < k < 2$. When $k \to 1$, the cost savings is

$\lim_{k\to+1}$ FGgain $= \frac{|D|-\delta|D|}{\log|D|}$. When $k \to 2$, the cost savings is $\lim_{k\to 2}$ FGgain $= \frac{-|D|\cdot\log\delta}{|D|-1}$, which is small compared to $\lim_{k\to+1}$ FGgain.

Most of the previous indexing algorithms extract testing queries from the graph database D: they first enumerate all subgraphs for each database graph $g \in D$, then draw random samples (following the uniform distribution) to generate the query set Q. We show in Appendix A that based on our experiments on chemical molecules, the queries (frequent subgraphs) and their support follow a power law relationship with $k = 1.26$. Hence, following the Equation 12, the cost saving by FG-Index on chemical molecules is $\frac{0.34|D|}{D^{0.26}-1}$.

C. Objective function VS Discriminative ratio

In related works [6]–[9], a feature p, to be brought into the index, has to be discriminative. In this subsection, we show that a feature selected according to the proposed objective function is discriminative, and the discriminative ratio is modeled more precisely than the previous works.

gIndex [6] defines a feature f to be discriminative w.r.t. f's subgraphs $sub(f) = \{f' | f' \subset f\}$: a feature f is discriminative if $|D(sub(f))|/|D(f)| > \gamma$. In tree+$\delta$ [9], the discrimination of a subgraph feature f is evaluated w.r.t. both subtrees contained in f, $T_f = \{t' \text{ is a tree } | t' \subset f\}$, and one subgraph $f' \subset f$: a subgraph feature f is discriminative if $\frac{D(T_f)-D(f)}{D-D(f)} > \epsilon_0$ and $\frac{D(f)}{D(f')} < \sigma^*$. In FG-Index [7], a discriminative feature f is defined w.r.t. its supergraphs: a feature f is a δ-TCFG feature if $\forall f' \in P_0$ and $f' \supset f$, $|D(f')|/|D(f)| < 1-\delta$. The above three measurements of discrimination confine to f's supergraph or subgraph(subtree) only. In this subsection, we propose a general definition of discrimination w.r.t all features in P_0.

We first use an example to show the benefit of defining the discrimination of a feature f w.r.t to P_0. Assume P_0 contains 3 features $P_0 = \{p_1, p_2, p_3\}$, and the supporting set of each feature is: $D(p_1) = \{g_1, g_2, g_3, g_4\}$, $D(p_2) = \{g_1, g_2, g_3, g_5\}$, and $D(p_3) = \{g_1, g_2, g_5\}$. In adition, all the queries contain those three features. The discrimination of a new feature p (assume p is the supergraph of both p_1, p_2 and $D(p) = \{g_1, g_2\}$) is then evaluated. Assume we set the discriminative ratio (as in gIndex) to be $4/3$, then p is discriminative because $|D(p_1) \cap D(p_2)|/|D(p)| = 3/2 > 4/3$. However, if we consider the discrimination of p w.r.t P_0, then p is not discriminative enough to filter any false graphs. In other words, although p is not redundant with its subgraphs p_1 and p_2, p is redundant with p_3, which has no containment relationship with p.

The conditional probability $Pr_D(p|P_0)$ can not be used to define the discrimination of a new pattern p w.r.t. feature set P_0. This is because $Pr_D(P_0) = 0$ when P_0 is a set of heterogeneous subgraph features (there is no database graph containing all the subgraphs in P_0). In order to find a reasonable definition, we take queries into consideration and propose the following definition.

Definition 4 (Discrimination): A feature p is discriminative w.r.t P_0 if $dis(p, P_0) < \delta$, where

$$dis(p, P_0) = \sum_{\forall \text{ unique } q} Pr(q) \cdot dis(p, P_0; q), \quad (13)$$

$$\text{and } dis(p, P_0; q) = \begin{cases} 0 & p \not\subset q \\ Pr_D(p | maxSub(q, P_0)) & p \subset q. \end{cases} \quad (14)$$

Based on the above definition, the discrimination of p is an expectation of $dis(p, P_0; q)$, thus it is independent of any specific query graph. When $p \not\subset q$, p does not participant in filtering for query q, $dis(p, P_0; q)$ is zero. When $p \subset q$, $dis(p, P_0; q)$ is the probability of graphs containing p under the condition that graphs containing $maxSub(q, P_0)$. Here, only $maxSub(q, P_0)$, instead of P_0 is considered because only $maxSub(q, P_0)$ in the feature set P_0 participants in filtering for query q.

Statement 2 (Coherence): Features maximizing the objective function are discriminative to current features P_0 in the index.

Proof: We first expand the discrimination of p on q:

$$\begin{aligned} dis(p, P_0; q) &= Pr_D(p | maxSub(q, P_0)) \cdot I(p \subset q) \\ &= \frac{Pr_D(p, maxSub(q, P_0))}{Pr_D(maxSub(q, P_0))} \cdot I(p \subset q) \\ &= \frac{D(\{p, maxSub(q, P_0)\})}{D(maxSub(q, P_0))} \cdot I(p \subset q). \end{aligned} \quad (15)$$

Recall that $C(q, P_0) = D(maxSub(q, P_0))$, hence,

$$dis(p, P_0; q) = \frac{C(q, \{P_0, p\})}{C(q, P_0)} \cdot I(p \subset q). \quad (16)$$

Therefore, the first monomials of Equation 9 can be rewritten as,

$$\sum_{q \in minSup(p,Q)} C(q, P_0) \cdot (1 - dis(p; P_0, q)), \quad (17)$$

which is a weighted expectation of $dis(p; P_0, q)$. Thus, the definition of the objective function and the discriminative value are coherent: the smaller the discriminative value, the larger the objective function. For an indiscriminative feature with large $dis(p; P_0, q)$, its objective function value is small. ■

D. Evaluating the objective function

To evaluate the objective function for a new feature p (as in Equation 9), the algorithm needs to know both $C(q, P_0)$ and $C(q, \{P_0, p\})$. According to Equation 1, $C(q, P_0)$ can be easily obtained from the intersection of $maxSub(q, P_0)$'s supporting sets. However, $C(q, \{P_0, p\}) = C(q, P_0) \cap D(p)$ can only be evaluated after obtaining $D(p)$. Computing $D(p)$ (by searching the graph index) for each enumerated feature p is time-consuming because it involves subgraph isomorphism tests. Therefore, we propose an upper bound and a lower bound of the objective function $gain(p, P_0)$, which help in pruning the features without calculating $D(p)$. For example, given a new feature p, if its objective-function's upper bound is

smaller than the objective value of the current optimal feature p^* or the lower bound of p^*, then p can be pruned. Hence $D(p)$ don't have to be computed for each subgraph feature p. The upper bound and lower bound of the objective function can alleviate the bottleneck of computing $D(p)$, Algorithm 1 describes the iterative graph mining algorithm using the upper & lower bound of the objective function in detail.

Algorithm 1 Iterative Graph Feature Mining

Input: Current featuresP_0, Queries Q
Output: The optimum feature p^*

$p^* = null$
for each feature p in all enumerated subgraphs **do**
  if $gain(p^*, P_0)$ is not precisely calculated **then**
    if $Upp(gain(p, P_0)) < Low(p^*, P_0)$ **then**
      Continue
    else if $Low(gain(p, P_0)) < Upp(p^*, P_0)$ **then**
      $p^* = p$
    else
      Calculate $gain(p^*, P_0)$
    end if
  end if
  if $gain(p^*, P_0)$ is precisely calculated **then**
    if $Upp(gain(p, P_0)) < gain(p^*, P_0)$ **then**
      Continue
    else if $Low(gain(p, P_0)) > gain(p^*, P_0)$ **then**
      $p^* = p$
    else
      Calculate $gain(p, P_0)$, compare with $gain(p^*, P_0)$
    end if
  end if
end for

Statement 3 (Upper and Lower bound): The objective function $gain(p, P_0)$, has an easy-to-compute upper bound $Upp(p, P_0)$ and a lower bound $Lower(p, P_0)$.

$$Upp(p, P_0) = \frac{1}{|Q|} \sum_{q \in minSup(p,Q)} |C(q, P_0) - D(q)| + \frac{1}{|Q|} \sum_{q \in Q} I(p = q)\, |C(q, P_0)| \quad (18)$$

$$Low(p, P_0) = \frac{1}{|Q|} \sum_{q \in minSup(p,Q)} |C(q, P_0) - \frac{1}{\gamma} D(max(p))| + \frac{1}{|Q|} \sum_{q \in Q} I(p = q)\, |C(q, P_0)|, \quad (19)$$

where γ is an self defined parameter ($\gamma > 1$).

Proof: **Upper Bound:** The maximum gain of p is to reduce $C(q, P_0)$ to $D(q)$ (reduce the gap to zero). Hence the upper bound holds. **Lower Bound:** Since $C(q, \{p, P_0\}) = C(q, P_0) \cap D(p)$, $|C(q, \{p, P_0\})| \leq |D(p)|$. Also, in order to enumerate and select only discriminative features, the feature p have to satisfy $\frac{D(maxSub(p))}{D(p)} \geq \gamma$ [6]. Hence $|C(q, \{p, P_0\})| \leq |D(p)| \leq \frac{D(maxSub(p))}{\gamma}$. The lower bound holds. ■

IV. Prune the Search Space

In this section, we propose algorithms to enumerate subgraph features and search for the subgraph optimizing the objective function. Since the search space is exponential, we propose two methods, branch & bound and QueryGrouping, to bound and prune the search space.

A. Branch and Bound

To search for the optimal feature p, we use a brute force algorithm as the baseline: enumerate all subgraph features with the DFS code [18] and evaluate their objective function values. As in Leap Search [15], an iterative frequency-descending mining method can be applied: it first searches for an optimal feature p^*_σ with the minimum support σ, then searches again with $\sigma/2$; the search continues with a decreasing minimum support until the optimal objective function value found in each iteration converges. Pruning only by frequency is not efficient enough because the number of frequent subgraphs easily blows up when σ is small. *Can we use the objective function to prune the search space too?*

The objective function of p, $gain(p, P_0)$, is neither monotonic nor anti-monotonic with respect to feature p. Therefore, it cannot be used directly for pruning the search space. A common method to reduce the search space is to derive a branch upper bound for features that contain p, $BUpp(p)$. $BUpp(p)$ represents the upper bound of the objective function for any feature $p' \supseteq p$, i.e., $BUpp(p) \geq gain(p')$, $\forall p' \supset p$ [14]–[16]. Note that the branch upper bound is different from the upper bound of the objective function introduced in the previous section. The branch of features containing the feature p can be safely pruned if $BUpp(p)$ is smaller than the current best $gain(p^*)$. For example, in Figure 2, candidate features are enumerated with order p_1, p_2, p_4, p_5. If we find that $gain(p_2) > BUpp(p_4)$, then $\forall p' \supset p_4$, $gain(p_2) \geq gain(p')$. Hence p_5 can be pruned.

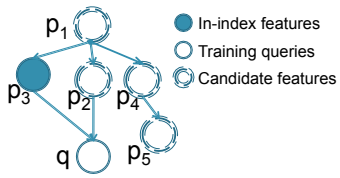


Fig. 2. One counterexample for the anti-monotonic of $minSup(p, Q)$

Statement 4 (Branch Upper Bound): For a feature p, a branch upper bound exists such that $\forall p' \supset p$, $gain(p') \leq BUpp(p)$, where

$$BUpp(p) = \frac{1}{|Q|}[\sum_{q \supset p} |C(q, P_0) - D(q)| + \max_{p' \supset p} |C(p')| \sum_{q \in Q} I(q = p')]. \quad (20)$$

Proof: For an arbitrary feature $p' \supset p$, we define the query set $Q_{super}(p') = \{q \in Q | q \supset p'\}$ as the *super-query set* of p'. It is easy to show that $Q_{super}(p') \subseteq Q_{super}(p)$ and $Q_{super}(p') \supseteq minSup(p', Q)$. And also, for each query

$q \in Q_{super}(p')$, the filtering effectiveness of feature p' is $C(q,P_0)-[C(q,P_0)\cap D(p)] \leq C(q,P_0)-D(q)$, since $D(q)$ is the lower bound of any candidate set, e.g., $[C(q,P_0)\cap D(p)]$. Therefore,

$$\begin{aligned}
&\sum_{q\in minSup(p',Q)} |C(q,P_0) - C(q,P_0)\cap D(p')| \\
&\leq \sum_{q\in minSup(p',Q)} |C(q,P_0) - D(q)| \\
&\leq \sum_{q\in Q_{super}(p')} |C(q,P_0) - D(q)| \\
&\leq \sum_{q\in Q_{super}(p)} |C(q,P_0) - D(q)|.
\end{aligned}$$

Also, $\sum_{q\in Q} I(p'=q)|C(q,P_0)| \leq \max_{p'\supset p}|C(p')| \sum_{q\in Q} I(p'=q)$. Therefore, the branch upper bound holds. ■

Although correct, this branch upper bound is not tight. One major cause of the loose bound is the change from $minSup(p,Q)$ in the objective function to $Q_{super}(p)$ in the branch upper bound. The reason that we use $Q_{supers}(p)$ instead of $minSup(p,Q)$ is that $minSup(p,Q)$ is not anti-monotonic, as can be seen in Figure 2. Figure 2 organizes features in a lattice, in which a direct edge starting from p_i pointing to p_j (or q_j) exists if and only if $p_i \subset p_j$ (or q_j) and $\nexists p_x \in P_0$, s.t., $p_i \subset p_x \subset p_j$ (or q_j). We prove $minSup(p',Q) \not\subset minSup(p,Q)$ when $p' \supset p$ by giving the following example: Given two features $p_1 \notin P_0$, $p_2 \notin P_0$ and $p_1 \subset p_2$, we can see that $q \in minSup(p_2,Q)$ but $q \notin minSup(p_1,Q)$ because $\exists p_3 \in P_0$, s.t., $p_1 \subset p_3 \subset q$. Therefore, $minSup(p_2,Q) \not\subset minSup(p_1,Q)$ when $p_2 \supset p_1$. Since the branch upper bound is not tight, its effectiveness in pruning the search space is limited. Therefore, we propose another search algorithm that can quickly identify a near optimal feature in the next subsection.

B. Query Grouping

The branch and bound search always starts from the root of the search tree and searches among frequent features of the whole query log according to the same search order, hence the search may be trapped into a local optimum. Besides deriving a tighter branch bound for the objective function, an alternative method for reducing the search space is to first quickly find a near-optimum feature, so that it can be used to prune the rest of the search tree. However, unlike other data, e.g., item sets, reordering the search order of frequent subgraphs is hard. In most "pattern growth" based methods, the search order is specifically designed to prevent one subgraph from being enumerated several times [19]. For example, gSpan [18] grows subgraph features according to a DFS-code tree and expands the right-most path or vertices on the right-most path, and Gaston [20] first enumerates all tree features and then grows them to graph features with loops.

We consider finding a near-optimal feature by confining the graphs we are mining. We first assign the queries into different groups, and then enumerate subgraph features over queries one group after another. Since we order the query groups according to their cumulative verification costs, we can always find a near-optimal feature after mining the first few groups. Although theoretical approximation ratio is hard to derive, the query grouping algorithm performs well in our empirical study.

1) Starting Point: For the sake of the completeness of the graph indices, normally the whole set of small subgraph features are included into the graph index. For example, in gIndex [6] all subgraph features with less than $minL$ edges are selected, and in FG-Index [7] all distinct edges are included into the in-memory index. If we organize all indexing features into a lattice as in Figure 3 (similar to the lattice in Figure 2), distinct edges or small features are all on the first level of the lattice. As a matter of fact, any new incoming subgraph

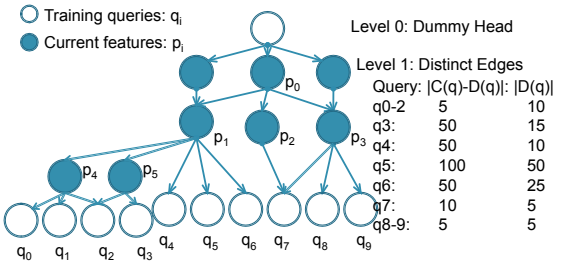


Fig. 3. Graph features organized in a lattice

feature p must be a supergraph of a feature in P_0. Given the above observation, we start searching for p from some feature in P_0, say p_1, instead of the root. But there are $|P_0|$ possible starting points. *Which $r \in P$ should we choose such that a near optimal feature can be quickly identified?*

Intuitively, a good starting point r should have the following properties:

1) A great proportion of the queries are supergraphs of r, otherwise there will be few queries using $p \supset r$ for filtering
2) The average size of the set of candidates for queries $\supset r$ are large, which means improvement over those queries is important.

At first glance, we may consider $BUpp$ to be a good starting point selection function, i.e., a feature $p_i \in P_0$ is a good starting root if its branch upper bound is high. However, this criterion chooses small features as the starting point. As shown in Figure 3, the feature p_0 is a subgraph of all the queries, and consequently its branch upper bound is the highest. But actually, it is not involved in the filtering of any query. If we start enumerating features from p_0, we may enumerate many subgraph features that are not used to answer any query, or only used for filtering the candidate set for queries q_{0-2}, q_8, q_9, which are already well filtered by the current feature set. Therefore, we consider defining a criterion based on both the branch upper bound and the objective function. We propose

the following starting-point function:

$$SPoint(r) = \sum_{q \in minSup(r,Q)} |C(q,P_0) - D(q)| + max_{p' \supset r}|C(p')| \sum_{q \in minSup(r,Q)} I(q = p'). \quad (21)$$

The above starting point function is consistent with the intuition stated above and additionally does not choose small features. For example, in Figure 3, instead of p_0, p_1 is chosen as the first starting point.

2) Subgraph Enumeration: After choosing the root r to start the search, the branch and bound search actually enumerates frequent subgraphs over queries that are supergraphs of r, $Q_{super}(r)$. But the search may still be trapped into a local optimum. For example, if we select the feature p_1 in Figure 3 to start the search, the search may easily enumerate a feature isomorphic to p_4 and keep on searching that branch. Since queries q_0, q_1 and q_2 all have very small $C(q)$, the search is trapped in a local optimum. To prevent this situation, we further reduce the mining set and enumerate frequent subgraphs only over r's minimal supergraph queries, $minSup(r, Q)$. However, since $minSup(r, Q)$ is not an anti-monotonic function, confining the mining over $minSup(p_1, Q)$ excludes the query q_3 in Figure 3. We argue that even if q_3 is not mined directly, a subgraph feature $p' \subseteq q_3$ can still be enumerated. Since the query q_3 is not well filtered given the feature p_5, one subgraph feature p' that can reduce $C(q_3)$ must be either a supergraph of p_5 or a sibling of p_5. (Two subgraphs a and b are *siblings*, a sib b, if they share a common subgraph and $a \not\subset b$, $b \not\subset a$.) If $p' \supset p_5$, p' can be enumerated latter when p_5 is chosen as the starting point. If p' sib p_5 and p' is generally effective for many queries, it is of high probability that p' will be enumerated in the search starting from p_1. When p' is enumerated, we find all of its minimal super queries (not confined to $minSup(p_1, Q)$) to calculate its objective function following the definition.

In summary, the search strategy we take is to first assign the training queries to groups, each of which is associated with an indexing feature $p_i \in P_0$, and queries in that group are $minSup(p_i, Q)$. We then enumerate all frequent subgraph features over queries within each group, and search for an optimal feature using the branch and bound algorithm proposed in the last section. Algorithm 2 describes the procedure.

Since query groups are searched according to the decreasing order of the $SPoint(p_i)$ function, a near-optimal feature may be identified after searching the first few groups. Besides this benefit, since the $minSup(p_i, Q)$ is much smaller than the overall training query set Q, enumerating the frequent subgraphs of $minSup(p_i, Q)$ is much faster. Furthermore, we usually set a low minimum support to prevent missing any interesting features in the mining over the whole training query log Q. But while searching over $minSup(p_i, Q)$, higher minimum support (e.g., $0.1|minSup(p_i, Q)|$), can be set. Otherwise, a feature with frequency $0.01|minSup(p_i, Q)|$ may cover too few queries. This relatively high minimum support setting is justified experimentally in section V.

Algorithm 2 Advanced Branch and Bound

Input: Current featuresP_0, Queries Q
Output: A new feature p

```
gain(p*) = 0
Sort all P_0 according to sPoint(p_i) function in decreasing order
for i = 1 to |P| do
    if branch upper bound of BUpp(r_i) < gain(p*) then
        break
    else
        Find the minimal supergraph queries minSup(r, Q)
        p*(r) = Branch & Bound Search(minSup(r, Q), p*)
        Find an optimal pattern p*(r) ⊃ r, gain(p*(r)) > gain(p*), update p* = p*_r.
    end if
end for
```

C. Multiple-feature Selection

Assigning queries to different groups and then mining each group separately is faster than the baseline branch and bound algorithm. However, in comparison to the "mine-at-once" algorithms, the iterative mining algorithm is still slow in finding N features since only one feature is added into the index in each iteration. To speed up the iterative mining, we propose to select the top k features after one iteration of the feature mining.

Simply choosing the k features with the highest objective function does not work. Recall that in section 3, the objective function of each new feature p is measured based on P_0. In each iteration, the top k features $\{p_1, p_2, ...p_k\}$ are all evaluated on the index with the feature set P_0. The benefit of adding p_j after p_i is minor when p_i and p_j's supporting set are similar and they are contained by the same queries, although both of them have high objective function values measured based on P_0.

To tackle the above difficulty, in each iteration of our multiple-feature selection algorithm, we collect the optimum feature p_i^* growing from each starting point r_i, and then choose the top k p_i^*s on the basis of the objective function to insert into the index. Besides setting k, we can also set a percentage threshold $r\%$, such that feature p_i^* with objective value at least $r\%$ of the best objective value are chosen. We observe that the top candidate subgraph features enumerated within the same group of queries tend to be redundant, but the redundancies between features enumerated in different groups are generally small. The reason is that subgraph features enumerated starting from different roots are structurally different hence they are used for filtering different sets of queries.

V. Experiment

In this section, we show the effectiveness and efficiency of our incremental feature mining algorithm on a chemical molecule dataset. We compare our iteratively mined features with three other classic subgraph features: discriminative

and frequent subgraphs (DF) [6], δ-Tolerant Closed Frequent Subgraphs (δ-TCFG) [7] and MimR features [8], which are reported to have high filtering rates. We set the parameters to the default values as suggested in [6]–[8], except where specified otherwise. For the branch and bound iterative feature mining algorithm (BB), we adopt a decreasing minimum support as in Leap Search [15], from $0.1|Q|$ to $0.01|Q|$ ($|Q|$ is the size of the query log). To have comparable results, we set the minimum support to be $0.1|Q_i|$ for the query grouping (QG) and the top k (TK) iterative algorithms, where Q_i refers to queries in each group i. For the TK algorithm, we choose all features with a gain value of least 80% of the gain of the best feature.

Since we do not have access to real query logs, we use simulated training and testing queries that are generated by first enumerating all subgraphs of a subset of randomly selected database graphs and then sampling using a uniform distribution or a normal distribution. The comparative results for both distributions are similar. Due to space restrictions, we only report the results on normal distribution. We evaluate the performance of our algorithms on the AIDS Antiviral Screen dataset[2], which contains $43,906$ graphs and has been commonly used to measure the effectiveness of subgraph querying features in previous work [6]–[8]. We also use data from the eMolecules dataset[3], which contains more than 5 million chemical structures.

A. Scenario 1: Updating the Index When Memory Increases

Typically, on average, the size of the memory taken by an index is proportional to the number of features in the index; thus we count the number of features as a proxy of the memory consumption. We first show how hard controlling the memory consumption is using the mine-at-once algorithms and tuning parameters. We run experiments on both gIndex and FG-Index with decreasing minimum support (from 0.1 to 0.01) over the AIDS dataset. As shown in Table I, each entry denotes the number of features that are newly discovered after decreasing the minimum support. For example, the entry $.1-.08$ counts features that are mined with minimum support 0.08 but not discovered with minimum support 0.1. As can be seen, there is no obvious relationship between the minimum support σ and the feature count. Therefore, given more memory, it is hard to tune a proper minimum support for current mine-at-once algorithms and build an index that utilizes the additional memory.

TABLE I
FEATURE COUNT WITH DECREASING MIN-SUPPORT σ

σ	$.1-.08$	.08-.06	.06-.04	.04-.02	.02-.01
DF	54	104	247	747	1655
$\delta-$TCFG	347	616	1441	5732	13434

Our incremental iterative mining algorithm can address the above problem. We first build three indices with the DF features DF$(.1)$, DF$(.05)$ and DF$(.02)$, where the number in the parentheses denotes the minimum support used to mine those features. As we can see in Figure 4(a), DF$(.05)$ includes more than 300 features than DF$(.1)$, but to achieve the same verification cost $C(q)$, the iterative algorithm only needs less than 100 new features. Based on the DF$(.05)$, we mine and add another 42 features. It performs even better than DF$(.02)$, which includes more than $1,100$ features than the DF$(.05)$, as in Figure 4(b). *Note:* Instead of comparing the real response time for subgraph search (which varies machine to machine), we use the candidate set $C(q)$ as the measurement of the response time (because of Equation 4).

We also observe that the QG algorithm and the TK algorithm sometimes perform better than the ground truth branch and bound algorithm, BB. This improvement is because BB algorithm may over-fit the training queries.

The running time for all our three algorithms is shown in Figure 4(c). TK is slightly faster than QG, and they are both about three times faster than BB. The fact that TK is only slightly faster than QG also shows that the high pruning power of the near-optimal features we obtain. To get the same verification-cost saving as from DF$(.05)$ to DF$(.02)$, the running time for QG or TK is significantly faster than mining DF$(.02)$. Also, as the available memory increases, our algorithm can be run to select features to incrementally augment the index. [4]

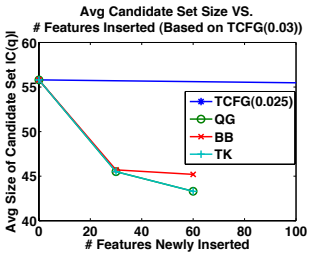


Fig. 6. Scenario one: Comparison between iterative methods and FG-Index(TCFG)

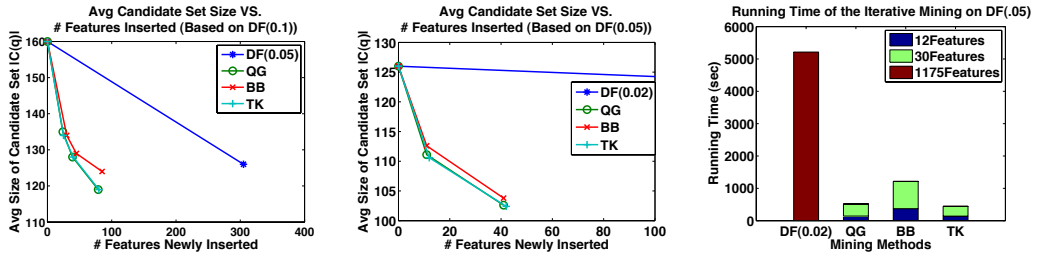
Our methods show similar improvements over an index constructed using the $\delta-$TCFG features. By decreasing the minimum support from .03 to .025, the number of $\delta-$TCFG features grows by $2,000$, and the number of frequent features stored on disk grows by $1,300$. We can achieve the same improvement by adding 30 features mined with the iterative mining algorithm, as in Figure 6.

B. Scenario 2: Updating the Index When Database Updates

In the second scenario, we simulate the update of the database. To accommodate this change, we mine additional features to insert into current index and remove equal number of less effective features out of the index. (The effectiveness of the deleted features is measured with the same objective

[2]http://dtp.nci.nih.gov/
[3]http://www.emolecules.com/

[4]The long running time for all algorithms is because we use a less memory-intensive mining strategy, in which we store all the supporting set of the features on disk and load them into memory when needed. But we fairly adopt this strategy on all examined algorithms.

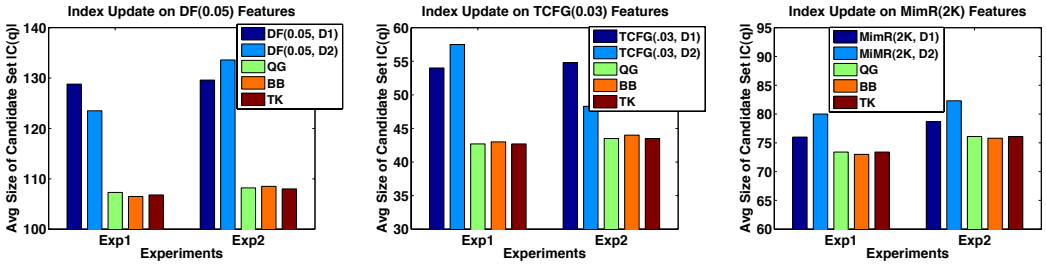


(a) Mining DF(0.05) VS. iterative methods base on DF(0.1)

(b) Mining DF(0.02) VS. iterative methods base on DF(0.05)

(c) Running time of four methods. They achieve the same candidate set size decrease

Fig. 4. For scenario one: Comparison between iterative methods and gIndex(DF).



(a) DF VS. iterative methods

(b) TCFG VS. iterative methods

(c) MimR VS. iterative methods

Fig. 5. For scenario two: Comparison between iterative mined features and re-mined features

function. Due to the page limitation, we do not discuss the feature-removing algorithm in detail.) In this experiment, we first generate a dataset D_1 by randomly choosing $10,000$ graphs from the AIDS dataset. Then we mine $DF(.05, D_1)$ and build a graph index $I_1 = I(DF(.05, D_1), D_1)$. To simulate the update of the graph database, we generate a new dataset D_2 by swapping $5,000$ graphs in D_1 with another $5,000$ graphs randomly chosen from the AIDS dataset (not in D_1). Then, a graph index $I_2 = I(DF(.05, D_1), D_2)$ is build with the same features as in I_1 but on D_2. To evaluate the goodness of an index over a graph dataset, we use the ratio $F = |C(q) - D(q)|/|D(q)|$. We observe the filtering ratio $F_{I_1} = 0.605$ and $F_{I_2} = 0.637$, which means the filtering power degrades while the database is updating. To accommodate this change, we run the "mine-at-once" algorithm again to mine $DF(.05, D_2)$ and build a new index $I_3 = I(DF(.05, D_2), D_2)$ and the filtering rate improves to $F_{I_3} = 0.574$. The feature mining and index construction take more than $1,500$ seconds. Using the iterative mining method, we can improve the filtering rate by adding 30 features (and removing 30 less effective features) as shown in Figure 5(a). Figure 7 shows that the TK algorithm takes 300 seconds, which is 0.2 times of the time taken by the gIndex algorithm to re-mine and re-construct an index

Interestingly, re-mining and re-constructing the index may sometimes degrade the performance of the index. In another experiment (exp2 in Figure 5(a)), we insert 2000 graphs randomly chosen from the eMolecules dataset, which is believed to have structurally different graphs than those in the AIDS dataset. As a result, if the DF features are re-mined, the performance of the new index is worse than the old one. But, for our iterative mining algorithms, since only features with positive objective-function values are mined and inserted into the index (and they all have higher objective function value than features deleted), it is guaranteed that the performance is better than the old index. Figure 5(b) and Figure 5(c) show that our methods have similar improvements over indices constructed using the $\delta-$TCFG features and MimR features. The mining and index construction running time is compared and reported in Figure 7.

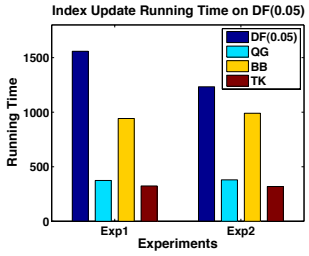


Fig. 7. Comparison of running time between re-mining DF and the iterative methods. The iterative methods achieve more candidate set size decrease than re-mining DF.

C. Scalability

Since evaluating the objective function of each feature involves finding the supporting set of that feature in the graph database, the running time of the iterative mining is linear in the size of the graph database. We also observe from Figure 8 that the BB algorithm is linear to the size of the training

queries, and the TK algorithm is sub linear, which further shows the merit of the TK algorithm on a large training set of queries when available. Also, we observe that enlarging the training query log does not improve the quality of the feature mining significantly. This shows that our algorithm can be trained using reasonably sized query logs.

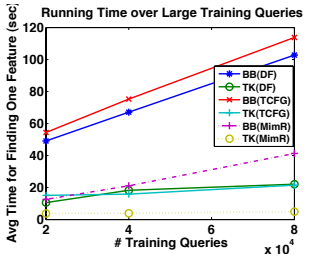


Fig. 8. Running time over varied training queries

VI. Related Work

The development of frequent subgraph feature mining algorithms such as AGM [21], FSG [22], gSpan [18] and Gaston [20] has created several feature-based graph indexing methods [6]–[11], [23]. gIndex [6] selects *frequent* and *discriminative* subgraphs; FG-Index [7] indexes all frequent subgraphs, which are partially stored on disk and partially in memory. Sun, Mitra, and Giles [8] have proposed an approach for mining *informative* and *irredundant* subgraphs; these subgraphs are selected to maximize the cumulative information gain and minimize pairwise redundancies, defined specific to a subgraph search problem. Also, to reduce the time required to mine the index features, other subtree-based approaches have been proposed [9]–[11].

All the above algorithms are "mine-at-once" algorithms that do not support incremental updating of the indexed feature set in response to database updates. The tree+δ algorithm tries to mine small non-tree graph features (δ) through mining each query q. But it is different from our algorithm, since (1) it only mines small non-tree graph features, (2) a graph feature is selected based on only one query, not the entire query log, and (3) it cannot accommodate any change to the graph structure or database.

The other category of the graph indexing methods is non-feature-based. In CTree [24], graphs in a database and their subgraphs or closure graphs are organized into an index tree. The answer set is retrieved by traversing this index tree, which is computationally costly because it involves expensive exact or inexact subgraph isomorphism tests. Williams, Huang, and Wang have proposed a Graph Decomposition Index (GDI) [25] which contains all graph decompositions of graphs in the database. Because of its large memory requirements, it is only effective for small databases of small and sparse graphs. GCode [12] encodes database graphs into spectrum codes and uses the interlacing theorem to prune the false candidate graphs. In iGraph [13], it is reported that GCode's filtering power is generally lower than that of others after comprehensive comparison experiments. GString [26] primarily targets at two-dimensional chemical structure graphs.

Besides mining subgraph features for indexing, graph features are also mined for classification and regression. Helma et al. use all frequent subgraph features for classification [27]. Other algorithms mining graph features with problem-specific objective functions are also proposed. Graph Boosting [14] proposes a branch and bound approach searching for graph features for classification based on the boosting algorithm. Saigo, Kramer, and Tsuda propose a similar algorithm solving the partial least squares regression [16]. Fan et al. propose a model-based search tree approach to find a small set of highly discriminative features for classification [17]. GraphSig [28] mines significant subgraphs in large graph databases by first converting graphs into feature vectors and then mining the vectors to obtain significant closed sub-feature vectors. Finally, graph features can be recovered from the closed sub-feature vectors. Yan et al. propose Leap Search [15] as a general framework for graph mining with specific objective functions. Besides finding an upper bound for the objective function and adopting the branch and bound paradigm, leap search also prunes a feature horizontally if its structure and frequency are similar to its siblings. However, this approach cannot be applied directly to our problem, because in our model, besides considering its variable structure and frequency, a new feature has to be irredundant to current in-index features.

VII. Conclusion

We investigated the iterative mining of graph features for the subgraph search problem. Compared with the existing mine-at-once algorithms, our iterative mining is effective in updating indices that accommodate changes, such as the update of the graph database or structure, a query change caused by users' interest shift, or extra memory becomes available for use. We observe that the top k feature selection process works best in terms of the feature quality and the mining time. For future work, one can consider the development of algorithms that estimate the objective function instead of calculating it precisely, which can further enhance the speed of an iterative mining algorithm.

References

[1] N.Trinajstic, *Chemical Graph Theory*, 2nd ed. CRC Press, 1992, vol. 1, 2.

[2] B. Schalkopf, K. Tsuda, and J. P. Vert, Eds., *Kernel Methods in Computational Biology*. MIT Press, 2004.

[3] M. El-Mehalawi and R. A. Miller, "A database system of mechanical components based on geometric and topological similarity." *J. CAD*, vol. 35, no. 1, pp. 83 – 94, 2003.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Graph matching applications in pattern recognition and image processing," in *ICIP*, 2003.

[5] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, 1971, pp. 151–158.

[6] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *SIGMOD*, 2004.

[7] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: Towards verification-free query processing on graph databases," in *SIGMOD*, 2007.

[8] B. Sun, P. Mitra, and C. L. Giles, "Irredundant informative subgraph mining for graph search on the web," in *CIKM*, 2009.

[9] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree + delta <= graph," in *VLDB*, 2007, pp. 938–949.

[10] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," *ICDE*, pp. 966–975, 2007.
[11] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *pVLDB*, 2008.
[12] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *EDBT*, 2008, pp. 181–192.
[13] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "igraph: a framework for comparisons of disk-based graph indexing techniques," *Proc. VLDB Endow.*, vol. 3, pp. 449–459, September 2010.
[14] T. Kudo, E. Maeda, and Y. Matsumoto, "An application of boosting to graph classification," in *NIPS*, 2004.
[15] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *SIGMOD*, 2008, pp. 433–444.
[16] H. Saigo, N. Krämer, and K. Tsuda, "Partial least squares regression for graph mining," in *KDD*, 2008.
[17] W. Fan, K. Zhang, H. Cheng, J. Gao, X. Yan, J. Han, P. Yu, and O. Verscheure, "Direct mining of discriminative and essential frequent patterns via model-based search tree," in *KDD '08*, 2008.
[18] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *ICDM*, 2002, p. 721.
[19] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*, 2nd ed. Morgan Kaufmann, 2006.
[20] S. Nijssen and J. N. Kok, "The gaston tool for frequent subgraph mining," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 1, pp. 77 – 87, 2005.
[21] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *PKDD*, 2000, vol. 1910, pp. 13–23.
[22] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," *Data Mining, IEEE International Conference on*, vol. 0, p. 313, 2001.
[23] D. Shasha, J. T. L. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching," in *PODS*, 2002.
[24] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *ICDE*, 2006, p. 38.
[25] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *ICDE*, 2007.
[26] H. Jiang, H. Wang, P. Yu, and S. Zhou, "Gstring: A novel approach for efficient search in graph databases," in *ICDE*, April 2007, pp. 566–575.
[27] C. Helma, T. Cramer, S. Kramer, and L. Raedt., "Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds," in *J. Chem. Inf. Comput. Sci*, 2004, pp. 1402–1411.
[28] S. Ranu and A. K. Singh, "Graphsig: A scalable approach to mining significant subgraphs in large graph databases," in *ICDE '09*, 2009, pp. 844–855.

APPENDIX

A. Distribution of Frequent Subgraphs in Database

In this section, we look at the distribution of subgraphs in the graph database because query independent approaches prefer to set this distribution as the probability distribution of the queries. We randomly choose $131,072$ (2^{17}) graphs from the eMolecules dataset that contains 5 millions chemical molecules graphs [5], and mine frequent subgraphs with the minimum support $0.01|D|$, where D is the dataset. We observe that there is a power law relationship between the number of subgraphs sg and its support, as shown in figure 9,

$$\text{Count}(D(sg) = w) = a \times w^{(-k)}, \tag{22}$$

where $\text{Count}(D(sg) = w)$ denotes the total number of subgraph sg (include duplications) with support w. One subgraph sg will be enumerated n times (sg will have n duplications in the query set) if its support is n. And from our empirical study, $k = 1.26$.

[5]http://www.emolecules.com/

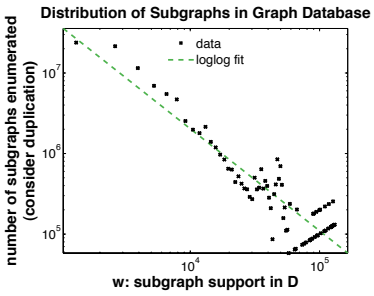


Fig. 9. The count of enumerated subgraphs VS. their support

B. gIndex's computation cost saving

Statement 5 (gIndex Cost Model): gIndex's isomorphism-test-cost depends on the query distribution $Pr_Q(q)$.

Proof: Assume we have the complete set of frequent subgraph features $\mathcal{F}$, and its power set is $\mathcal{S}(\mathcal{F})$. The isomorphism-test-cost of gIndex given feature set $\mathcal{F}$ is

$$\begin{aligned} \underset{verf}{T} &= \sum_{\forall \text{ unique } q} \sum_{q \notin \mathcal{F}} Pr_Q(q)|C(q)| \\ &= \sum_{\forall \text{ unique } q} Pr_Q(q)|C(q)| - \sum_{q \in \mathcal{F}} Pr_Q(q)|C(q)|. \end{aligned} \tag{23}$$

Since gIndex uses subgraph features $F = sub(q, \mathcal{F}) = \{f \subset q | f \in \mathcal{F}\}$ for filtering query q, $C(q) = \cap_{f \in sub(q,\mathcal{F})} D(f) = D(sub(q, \mathcal{F}))$. Hence,

$$\begin{aligned} &\sum_{\forall \text{ unique } q} Pr_Q(q)|C(q)| = \sum_{\forall \text{ unique } q} Pr_Q(q)|D(sub(q, \mathcal{F}))| \\ &= \sum_{\forall F \in \mathcal{S}(\mathcal{F})} Pr_Q(F) \cdot |D(F)|, \end{aligned} \tag{24}$$

where $Pr_Q(F)$ denotes the probability that a query graph q uses features F for filtering.

Since the complete set of frequent subgraph features $\mathcal{F}$ is too big to fit into the memory, gIndex only uses discriminative and frequent features $\mathcal{F}_d$ for indexing. Correspondingly, we use F_d to denote $\{f \in F | f \in \mathcal{F}_d\}$. Inevitably, the isomorphism-test-cost goes up because the candidate set generated by $\mathcal{F}_d$ is not as tight as $\mathcal{F}$. gIndex tries to minimize the inflation of isomorphism-test-cost, $O(\mathcal{F}_d)$, while reducing $\mathcal{F}$ to $\mathcal{F}_d$,

$$\begin{aligned} O(\mathcal{F}_d) &= \sum_{\forall F \in \mathcal{S}(\mathcal{F})} Pr_Q(F) \times |D(F)| \left(\frac{|D(F_d)|}{|D(F)|} - 1\right) \\ &= \sum_{\forall \text{ unique } q} Pr_Q(q) \times |D(sub(q, \mathcal{F}))| \left(\frac{|D(sub(q, \mathcal{F}_d))|}{|D(sub(q, \mathcal{F}))|} - 1\right) \end{aligned}$$

gIndex removes feature f from $\mathcal{F}$ if $|\cap D_{f' \subset f}(f')|/|D(f)| > \gamma$, which is a heuristic to confine $\frac{|D(sub(q,\mathcal{F}_d))|}{|D(sub(q,\mathcal{F}))|} < \gamma^n$ when $|sub(q, \mathcal{F})| - |sub(q, \mathcal{F}_d)| = n$. Both the isomorphism-test-cost inflation and its upper bound depends on the underlying query distribution. ■