TreePi: A Novel Graph Indexing Method

Shijie Zhang, Meng Hu, Jiong Yang
EECS Dept., Case Western Reserve University
10900 Euclid Avenue, Cleveland, OH 44106
{shijie.zhang, meng.hu, jiong.yang}@case.edu

Abstract

Graphs are widely used to model complex structured data such as XML documents, protein networks, and chemical compounds. One of the fundamental problems in graph databases is efficient search and retrieval of graphs using indexing techniques. In this paper, we study the problem of indexing graph databases using frequent subtrees as indexing structures. Trees can be manipulated efficiently while preserving a lot of structural information of the original graphs. In our proposed method, frequent subtrees of a database are selected as the feature set. To save memory, the set of feature trees is shrunk based on a support threshold function and their discriminative power. A tree-partition based query processing scheme is proposed to perform graph queries. The concept of Center Distance Constraints is introduced to prune the search space. Furthermore, a new algorithm which utilizes the location information of indexing structures is used to perform subgraph isomorphism tests. We apply our method on a wide range of real and synthetic data to demonstrate the usefulness and effectiveness of this approach.

1 Introduction

Database systems are increasingly being used to manage complex structured data like sequences, trees, and graphs. Graphs are among the most complicated and general form of structures, and they are heavily used to represent compounds, proteins and relationship networks in chem-informatics and bio-informatics datasets.

Graph databases are used in developing search and registration systems for chemical and biological structures. In chemistry, a large number of newly discovered chemical molecules are studied, classified, and recorded every year. ChemIDplus, a free data service offered by the National Library of Medicine, provides users with structural and nomenclatural information of chemical molecules. It helps to identify subsets of molecules related to query structures and shortens the discovery cycle in drug design and other scientific activities. Bio-informatics also requires efficient mechanisms to query a large number of biological pathways and protein interaction networks, which are complicated with embedded multilevel structures. Web XML files are typically graph databases as well. Furthermore, graph databases are widely used in the computer vision and pattern recognition. All these applications indicate the importance of graph databases, as well as the importance of effective graph databases search.

In many cases, the success of a graph database application is directly dependent on the efficiency of the graph query processing. The classical graph query problem is to find all supergraphs of the query graph from a graph database. Obviously it is inefficient to perform a sequential scan on every graph in the database, because the subgraph isomorphism test is expensive.

Therefore, it is necessary to construct graph indices to accelerate the graph query processing. Many indexing methods [3, 7, 14] have been developed for the XML query, which is a simple type of graph query. In order to process arbitrary graph queries, GraphGrep [14] and several other path-based graph indexing approaches are proposed. The general idea of them is as follows: First, they enumerate all existing paths in a database up to a maximum length and index them. Then, they uses the indices to identify every graph that contains all the paths in the query graph. However, the size of index path set could increase drastically with the size of graph database.

gIndex [18] is developed to solve the exact graph query problem by using general frequent patterns as basic index structures. Grafil, another frequent pattern based indexing method, is proposed for inexact graph query in [17]. However, while Grafil works well for approximate graph queries, frequent pattern based approaches are not efficient enough to retrieve the exact matches for the query graph from the database, because index patterns are irregular and the subgraphs isomorphism tests are slow. All the above suggest that further improvements are necessary for both index structures and algorithms.

The tree, as a special form of graphs, has many beneficial properties. In this paper, instead of using paths or arbitrary frequent subgraphs, we use frequent subtrees as

1-4244-0803-2/07/$20.00 ©2007 IEEE.

the index structures. Trees are more complex patterns than paths and trees can preserve almost equivalent amount of structural information as arbitrary subgraph patterns. Besides, the frequent subtree mining is relatively easier than general frequent subgraph mining [4, 6, 9, 16, 19], which enables us to construct indices more efficiently.

Our frequent tree based indexing approach works as follows: first, we perform the frequent tree mining on the graph database, and then select a set of frequent trees as index patterns. In the query processing, for a query graph q, we enumerate the frequent subtrees in q and identify the graphs in the database which contain those subtrees. As the canonical form of any tree can be calculated in polynomial time, the first screening operation can be performed very fast. Moreover, by applying the Center Distance Constraints, which is described in a later section, we can shrink the graph database even closer to the actual support set of the query graph, thus to reduce the search space significantly. In the final verification phase, we take advantage of the location information partially stored with the feature trees. A novel algorithm for the subgraph isomorphism test is devised based on the location information.

The choice of frequent subtrees as index structure is not a simple tradeoff between paths and frequent subgraphs. First, trees are more compact forms to preserve the structural information in graph database, especially for chem-informatics and bioinformatics data sets. Second, the canonical form of any tree can be quickly computed, which facilitates index searching. More importantly, the symmetric nature of trees makes it possible to partially keep the location information of the index structures. To the best of our knowledge, our algorithm is the first graph indexing method to utilize specific location information, and hence to maximize the benefits from graph mining results, i.e., the frequent substructures.

The remainder of the paper is organized as follows. Section 2 is the related work. Section 3 defines the preliminary concepts. Section 4 discusses the selection of index structures. Section 5 describes the query processing. Section 6 reports the performance study of our proposed algorithm. Discussion is presented in Section 7 and Section 8 concludes our work.

2 Related Work

The problem of graph query processing has been widely studied in many fields. In content-based image retrieval, Petrakis et al.[13] indexes graphs in high dimensional space by R-trees. In 3D protein structure search, algorithms using hierarchical alignments and geometric hashing are proposed by Madej et al.[10]. However, these methods are restricted to their own disciplines and are not efficient for graph queries in large graph databases.

In semistructured XML databases, query languages built on path expressions became very popular. Kaushik et al. introduces new techniques to efficiently extract exact answers to regular path queries in [7, 8]. APEX [2] uses adaptivity of index structures to fit the query load. In [1], Bruno et al. compares two main path query processing methods. To perform full scale graph retrieval, Shasha et al. [14] develops GraphGrep, a path-based algorithm, which is also used in Daylight system [5]. However, although paths are easier to manipulate, they also lose a large amount of structural information. Furthermore, the number of paths in a database could grow drastically when the graphs are large and diverse. Therefore, some new structures are proposed lately for XML query [11, 12].

Recently some graph indexing methods began to utilize graph mining results. Yan et al. proposes gIndex [18] and adopts frequent structures as index patterns. For any query graph q, gIndex enumerates the frequent subgraphs of q and identifies the graphs in the database which contain those subgraphs. Then, gIndex performs naive subgraph isomorphism tests for final verification. However, the calculation of the canonical form of an arbitrary graph can be very time-consuming. Since gIndex checks a large number of subgraphs of the query graph, the matchings between index patterns and those subgraphs are far from efficient. Although gIndex can reduce the search space a lot by intersecting the support sets of indexed subgraphs before the final verification, the reduced search space could still be as large as three times the actual support set of the query graph. Also, the final verification of gIndex, which uses naive graph isomorphism tests, is inefficient.

During the index construction, gIndex obtains every occurrence of frequent subgraphs in the graph database. However, due to memory limitations and asymmetry of index patterns, gIndex is not able to store and use this useful location information.

3 Preliminaries

In this section, we introduce the terminology used in this paper and give the formal problem definition. As an important data structure, the labeled graph is used to model complicated structures and schemaless data, e.g., XML is a kind of directed labeled graph, and chemical compounds are undirected labeled graphs. In this paper, we investigate the graph indexing methods for undirected labeled graphs, however it is easy to extend our method to directed labeled graphs. Some definitions are presented as follows:

Definition 1 *A* **labeled graph** *G is a five element tuple $G = \{V, E, \Sigma_V, \Sigma_E, l\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of undirected edges. Σ_V and Σ_E are the sets of vertex labels and edge labels respectively. The labeling function l defines the mappings $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$.*

Definition 2 *A labeled graph $G = (V, E, \Sigma_V, \Sigma_E, l)$ is* **isomorphic** *to another graph $G' = \{V', E', \Sigma'_V, \Sigma'_E, l'\}$, denoted by $G \approx G'$, iff there exists a bijection $f : V \rightarrow V'$ such that*

1. $\forall u \in V$, $(l(u) = l'(f(u)))$,

2. $\forall u, v \in V, ((u, v) \in E \Leftrightarrow (f(u), f(v)) \in E')$, *and*

3. $(u, v) \in E$, $(l(u, v) = l'(f(u), f(v)))$.

The bijection f is called an isomorphism between G and G'. We also say that G is isomorphic to $G'$and vice versa. A graph **automorphism** *is an isomorphism from G to itself.*

Definition 3 *A labeled graph G is* **subgraph isomorphic** *to a labeled graph G', denoted by $G \subseteq G'$, iff there exists a subgraph G'' of G' such that G is isomorphic to G''.*

Definition 4 *Graph $G = \{V, E, \Sigma_V, \Sigma_E, l\}$ and $G' = \{V', E', \Sigma_{V'}, \Sigma_{E'}, l'\}$ are non-edge-overlapping iff $E \bigcap E' = \varnothing$, two non-edge-overlapping graphs G and G' are matched iff $V \cap V' \neq \varnothing$, i.e, G and G' intersect only on vertices.*

Definition 5 *Given a graph q, a partition of q is a set of non-edge-overlapping subgraphs $\{s_1, s_2, ..., s_m\}$, such that $q = \bigcup_{i=1}^{m} s_i$.*

Definition 6 *Given a graph database $D = \{g_1, g_2, ..., g_n\}$, the support set of graph g, denoted by D_g, is defined as the subset of D to which g is subgraph isomorphic.*

$$D_g = \{g_i | g \subseteq g_i, g_i \in D\}$$

We denote $|D_g|$ as the size of D_g.

Problem Statement: In this paper, we investigate the following **graph query** problem. For a graph database D and a query graph q, find $D_q \in D$ that support q.

Figure 1 shows an example graph database and Figure 2 shows an example query. The support set of the query graph is {b,c}. This example database and query graph will be used as the running example throughout this paper.

The processing of graph queries in our paper can be divided into two major steps:

1. **Database Preprocessing** This is the index construction step in which we enumerate and select frequent subtrees in graph database D as feature trees. The feature tree set is denoted by T_D. For any $t_i \in T_D$, we also calculate the support set of t_i, D_{t_i}, in the preprocessing step.

2. **Query Processing** This step includes three sub-steps: 1.**Partition**, in which we partition the query graph into a set of feature trees $SF_q = \{tf_1, tf_2, ..., tf_n\}$, $tf_i \subseteq q$. 2.**Filtering & Pruning**, which is a two-step screening operation. First we project the graph database to a small set, $P_q = \bigcap_t D_t (t \in SF_q, t \in T_D)$, which consists of all graphs containing all the feature trees in SF_q. Then Center Distance Constraints are applied to reduce P_q to P'_q. 3.**Verification**, in which we devise a new subgraph isomorphism algorithm to verify whether or not q is contained by the graphs in P'_q by reconstruction from TP_q.

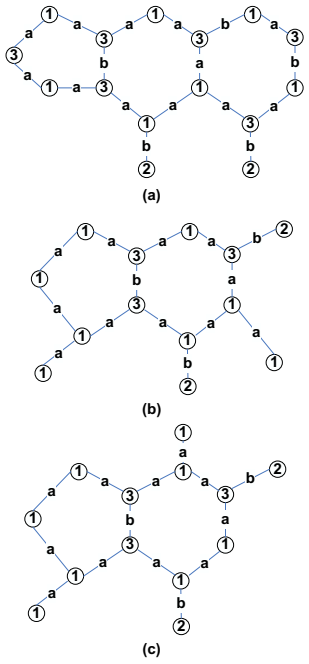


Figure 1. A Sample Graph Database

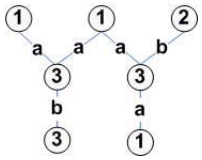


Figure 2. A Sample Query Graph

In the query processing step, the verification sub-step is most time-consuming. Therefore, the goal of the indexing method is to shrink the space as much as possible before the final verification.

4 Database Preprocessing

4.1 Feature Tree Selection

In the paper, we use trees as index structures for three reasons. First, operations on trees, such as isomorphism and normalization are asymptotically simpler than graphs, which are usually NP-complete. Second, a large number of important structures in biology and chemistry applications are really trees, e.g, structures in RNA. Last but not the least, trees can be adapted well in the reconstruction based framework of our verification algorithm, as shown in a later section.

In our paper, a feature tree is a selected frequent tree in the graph databases. A tree t is σ frequent if $|D_t| \geq \sigma$.

Figure 3 shows some 3-frequent trees of the example graph database. In the preprocessing procedure, we select a part of frequent trees of different frequency and sizes as the feature tree set. How to select the set of feature trees is discussed in section 4.1.3.

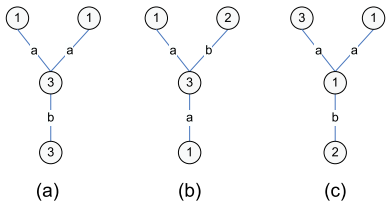


Figure 3. Frequent Trees

Due to the limit of memory space, we can not store all the frequent trees of a very low frequent threshold σ. So we use two following methods to make T_D more compact, but preserve as much information as possible.

4.1.1 $\sigma(s)$ Function

The number of different trees will grow exponentially when the size of the trees increases, e.g., there are $(n+1)^{n-1}$ different n edge trees if vertices have different labels. Therefore, it is not suitable to use a uniform threshold for different size feature trees. Here we introduce support function $\sigma(s)$.

In order to guarantee the completeness of the indexing, we set $\sigma = 1$ for single edge trees so that we can successfully partition the query graph into feature trees in the worst case. On the other hand, a large frequent tree is more likely to have same support set as its subtrees. Then, it is not meaningful to store large trees with low support. In our method, we select the following non-decreasing function $\sigma(s)$, in which s is the edge size of the feature trees.

$$\sigma(s) = \begin{cases} 1 & \text{if } s \leq \alpha \\ 1 + \beta s - \alpha\beta & \text{if } \eta \geq s > \alpha \\ +\infty & \text{if } s > \eta \end{cases} \quad (1)$$

where α, β, η are positive parameters which can be tuned based on the graph database. The $\sigma(s)$ function can not only ensure the completeness but also reduce the size of the feature tree set.

4.1.2 Shrinking

However, when the graph database is large and diverse, the feature tree set could still be very large. Therefore we need to develop other methods to shrink the feature tree set.

For any feature tree r with support set D_r, suppose the set of subtrees of r, except r itself, is $\{r_1, r_2, ..., r_n\}$. According to the definition of support set, we have $|\bigcap_i D_{r_i}| \geq |D_r|$. When $|\bigcap_i D_{r_i}|$ and $|D_r|$ are equal, D_r is not representative any more, as it can be calculated from the supports sets of its subtrees. Based on the above observation, we adopt the following shrinking mechanism to remove less important tree patterns.

If for any feature tree r, $|\bigcap_i D_{r_i}|/|D_r| \leq \gamma$, we would remove r from the feature tree set, where γ is called shrinking parameter .

4.1.3 Feature Tree Selection Algorithm

First, all the frequent trees according to the σ function are discovered by any level wise edge-increasing graph mining method [4, 6, 9, 16, 19]. Then, we shrink the frequent tree set according to the shrinking parameter γ introduced above. The remained frequent trees are selected as feature trees. The size of the feature tree set can be adjusted by variable α, β, η and γ.

While the setting of these parameters relies on the graph database and query graphs, there are some heuristics for choosing the values. Generally, the range of α is from $\bar{s_q}/4$ to $\bar{s_q}/2$, where $\bar{s_q}$ is the estimated average size of query graphs. When most frequent substructures below 10 edges in the database, β should be set to a lower value, e.g, 1 or 2. Otherwise, we can set β relatively larger, e.g., 5 or 6. In this paper, η is set to $min\{\bar{s_q}, \bar{s_D}\}$, where $\bar{s_D}$ is the average size of graphs in the database. The range of γ is from 1 to 3. When the memory size is too small to store all feature trees, we gradually decrease η and α, and increase the shrinking parameter γ, until the feature tree set can fit in the memory.

4.2 Index Construction

4.2.1 The Center of a Tree

Unlike other graph patterns, every tree has a unique center. From the center of the tree, we can find the canonical form of a tree in polynomial time, while it takes exponential time for regular graphs. Figure 4 demonstrates the center of an example graph. We introduce the theorem below.

Theorem 1 *The center of a tree consists of one vertex or two adjacent vertices, i.e., the center of a tree can be represented by a vertex or an edge.*

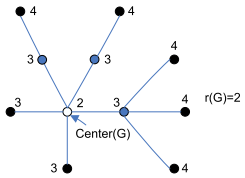


Figure 4. Center of a Tree

Due to the space limitations, we will omit the proof in this paper. In the preprocessing phase, we divide the feature trees into two groups by whether the center of the tree is a vertex or an edge. For each selected feature tree t, if the center of t is a vertex, in any graph g containing t, we use one bit for every vertex in g. We set the bit 1 if the corresponding vertex is the center of at least one subtree isomorphic to

t, 0 if not. The same procedure can also be applied to the edges when the center is an edge.

4.2.2 Tree Canonical Form

In this subsection, a canonical form of trees is presented. A unique string representation of a tree can be obtained from its canonical form, and used to index feature trees and test tree isomorphism. A method to transform a labeled tree into its canonical form is given.

Given a tree, its center(s) can be found by repeatedly removing leaves. Starting from a tree, we keep removing leaf nodes until only one node or two adjacent nodes are left. The remaining node(s) are considered as the center(s) of the tree. This process can be done in $O(n)$ time, where n is the number of vertices in the tree. This process is demonstrated in Figure 4, where leaf nodes removed in each round are marked differently.

After obtaining the center(s) of a tree, all nodes are sorted in polynomial time according to a defined order. First each node in the tree is represented in a 2-tuple, (L_e, L_v), where L_e is the label of the edge connecting the node to its parent, and L_v is the label of the node. For the root node, L_e is given an empty value. The order of two nodes at the same level is defined recursively as follows: First their L_e values are compared. If they are the same, the L_v values are compared. If both of them are the same, their subtrees from left to the right are compared, until the order is solved. According to the order defined above, we can sort all nodes in a rooted tree to obtain its canonical form. For example, in Figure 5, the original form and the canonical form of a rooted tree are given. The labels on each node and each edge are the node labels and edge labels. The 2-tuple of each node is shown beside each node.

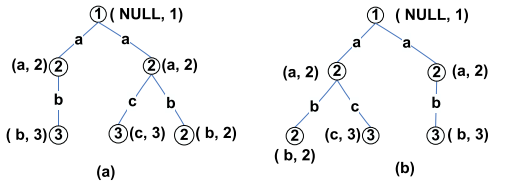


Figure 5. Canonical Forms

After transforming a rooted tree to its canonical form, a unique string representation of the tree can be constructed. A breadth-first-search starting from the root node is performed to obtain the string representation. Obviously, the construction of string representation can be done in $O(V)$ time.

After the string representation for each feature tree is obtained, a prefix tree based indexing is used to index all feature trees. Since all feature trees are transformed to strings, other traditional indexing techniques, such as B+ tree, can also be applied here.

5 Query Processing

In this section, we present the second part of the TreePi algorithm. We illustrate the design and implementation of the algorithm in three subsections: (1) Partition (2) Filtering & Pruning and (3) Verification.

5.1 Partition Query Graph

A partition p of query graph q is a set of non-edge-overlapping subgraphs as in Definition 6. If all subgraphs in the partition p are feature trees, we define p as a Feature-Tree-Partition. Suppose the feature tree set of the example database is the same as the frequent trees in Figure 3, Figure 6 demonstrates an example Feature-Tree-Partition of the example query graph. It is obvious that any query graph has at least one Feature-Tree-Partition, since in the worst case it can be partitioned into all one edge trees, which are always selected to be feature trees. A Feature-Tree-Partition p of query graph q is minimum if the size of p, denoted as $|p|$, is the smallest among all the Feature-Tree-Partitions of q.

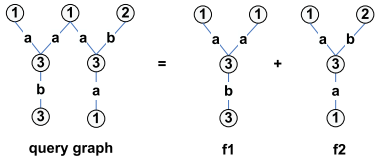


Figure 6. Feature Tree Partition

Since it is impossible to find the minimum Feature-Tree-Partition in polynomial time, a randomized algorithm is adopted to find a solution, which can also generate a group of additional feature subtrees of the query graph at the same time.

In the procedure $RP(q)$, if q is not a feature tree in the index list, q is randomly partitioned into two matched subgraphs q_1 and q_2, and if q_1 or q_2 is still not a feature tree, we will continue to partition the subgraph into two matched graphs. The procedure terminates when all partitioned subgraphs are feature trees.

Given a query graph q, the procedure $RP(q)$ is executed δ times, after which δ groups of different Feature-Tree-Partitions $\{TP_{q_1}, TP_{q_2}, ..., TP_{q_\delta}\}$ are obtained. Then we set the minimum partition among these δ Feature-Tree-Partitions, $TP_{q_i}, (|TP_{q_i}| \leq |TP_{q_j}|, 1 \leq j \leq \delta, j \neq i)$ as TP_q. TP_q will be used in the verification step. All the δ groups of Feature-Tree-Partitions are united into a feature subtree set of q, i.e. $SF_q = \bigcup_{j=1}^{\delta} TP_{q_j}$. SF_q will be used to reduce the search space in the following step.

5.2 Filtering & Pruning

5.2.1 Filtering by Intersection

Having obtained the feature subtree set SF_q, we intersect the support sets of all feature trees in SF_q, and obtain the filtered set P_q. The underlying intuition is that, if graph

g does not contain any of q's subtree, g will not contain q either. Therefore, query graph q will not be embedded in any graph outside P_q.

Algorithm 1 *Filtering*

Input: Graph database D, Feature tree set F, Query graph q, and maximum feature tree size L
Output: Filtered set P_q

```
1: let P_q ← D
2: for each t ∈ SF_q do
3:     if t ∈ T_D then
4:         P_q ← P_q ⋂ D_t
5:     end if
6: end for
```

In order to avoid processing unnecessary feature subtrees, the algorithm is optimized by the following strategy: if a subtree t is in the feature tree set, it is unnecessary to process any feature subtrees contained in t. In most cases, filtering can effectively reduce the search space. In the next subsection, a further pruning method is introduced.

5.2.2 Pruning by Center Distance Constraints

After the filtering, we adopt a more powerful pruning technique based on the constraints imposed by the distances between feature subtrees centers in the query graph. The concept of Center Distance Constraint is as follows: If q is subgraph isomorphic to graph g, then at least one set of feature subtrees $TP'_q = \{tp'_1, tp'_2, ..., tp'_m\}$ is embedded in g, which satisfies $tp_i \approx tp'_i, 1 \leq i \leq m$, and $d_q(center(tp_i), center(tp_j)) \geq d_g(center(tp'_i), center(tp'_j)), 1 \leq i \neq j \leq m$. The rationale is that, if $q \subseteq g$, there exists at least one subgraph q' of g, and $q \approx q'$, q' should also be able to be partitioned into a set of feature subtrees in the same way as q. In q', the center distance between any pair of partitioned feature subtrees should be the same as that of q. Therefore, according to the definition of distance in graphs, the distance between the centers of two partitioned feature subtrees in g, in which q' is embedded, should not be greater than its counterpart in q.

Figure 7 shows a simple example of pruning by Center Distance Constraints. Suppose the example query graph is partitioned into feature subtree f_1 and f_2 as in Figure 6, and the distance between the centers of f_1 and f_2 in q is 2. In both Figure 7(a) and (b), a pair of f_1 and f_2 are shown. In Figure 7(a), the distance between the centers of f_1 and f_2 is 4 while the distance is 2 in Figure 7(b). Therefore, Figure 7(a) will be pruned since it can not satisfy the Center Distance Constraint, even though it is in the filtered set.

Applying the concept of Center Distance Constraints, we develop Algorithm 2 to further reduce the size of the filtered set. If there does not exist any set of feature subtrees in graph g, then g will be deleted from P_q. Otherwise, all possible sets of feature subtrees satisfying Center Distance Constraint will be used as the input of final verification in next subsection. It should be addressed that after Center_Prune, for each graph g in P'_q and the residual set of P_q, we obtain groups of features subtrees, from which query graph q may be reconstructed in g. However, at this point only the center positions of those feature subtrees in each group are known, and the exact position of each vertex of the subtrees are unknown.

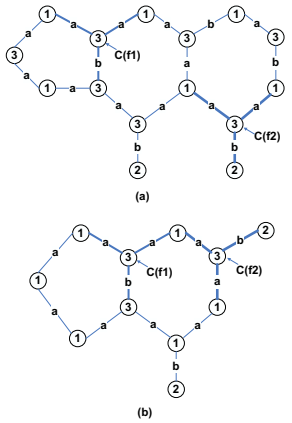


Figure 7. Center Distance Constraint

The novelty of this pruning mechanism is that it utilizes the location information, which is entitled by the nature of trees. Arbitrary substructure patterns adopted by gIndex do not have a unique center to perform any pruning operation. Since the center distances impose much stronger constraints on the candidate graphs, the algorithm can filter most of the remaining false positive graphs in the filtered set.

Algorithm 2 $Center\_Prune$

Input: Feature tree set F, Query graph q, TP_q, P_q
Output: Reduced filtered set P'_q

```
1: P'_q ← ∅
2: for each graph g in P_q do
3:     for each TP'_q, tp_i ≈ tp'_i, tp'_i ⊆ g do
4:         if TP'_q satisfies center distance constraint then
5:             label g
6:         end if
7:     end for
8:     if g is labeled then
9:         P'_q ← P'_q + g
10:    end if
11: end for
```

5.3 Verification

In this subsection we provide a new subgraph isomorphism algorithm for verification. The advantage of this method over verification based on brute force search is that it utilizes the location information stored in the preprocessing step, and at the same time, the new canonical reconstruction form avoids any isomorphism test.

In gIndex, for any graph g and query graph q, since we do not have any location information of q in g, a naive subgraph isomorphism test has to be performed between g and q, which is very time consuming. Based on the result of $Center\_Prune$, first a depth first search is used to retrieve all the possible feature subtrees centered in the stored positions, then we test if we can reconstruct the query graph q from the retrieved feature subtrees.

5.3.1 Canonical Reconstruction Form

In this subsection, we present a new canonical reconstruction form, which determines whether or not two graphs are isomorphic without isomorphism tests.

Given two pairs of matched graphs s_1 and t_1, s_2 and t_2, $s_1 \approx s_2, t_1 \approx t_2$, in order to find if $s_1 \cup t_1 \approx s_2 \cup t_2$, we first construct two graphs s and t, s.t., $s \approx s_1, s \approx s_2$ and $t \approx t_1, t \approx t_2$. Then a unique random number is assigned to every vertex in s and t, and denote $s(v_i)$ or $t(v_i)$ as the number assigned to v_i in s or t. We extend the operation to arrays, e.g, $s(array[v_1, v_2, ..., v_l]) = array[s(v_1), s(v_2), ...s(v_l)]$.

Suppose s has a automorphisms $\{f_{s1}, f_{s2}, ..., f_{sa}\}$ and t has b automorphisms $\{f_{t1}, f_{t2}, ..., f_{tb}\}$. Regarding $p(s_i \cap t_i)$ as a permutation of vertex set $(s_i \cap t_i)$, $f_{sj}(p(s_i \cap t_i))$ is also a permutation of vertices, each entry of which is mapped by f_{sj} from the origin entry in $p(s_i \cap t_i)$. we define the canonical reconstruction form $crf[s_i \cup t_i, s_i, t_i]$ as $(min_{i,j,p}[f_{sj}(s(p(s_i \cap t_i))), f_{tk}(t(p(s_i \cap t_i)))], s, t), 1 \leq j \leq a, 1 \leq k \leq b$. The definition of minimum could be based on any kind of partial orders on $2l$ arrays.

If $crf[s_1 \cup t_1, s_1, t_1]$ is equal to $crf[s_2 \cup t_2, s_2, t_2]$, then we can find a bijection from $s_1 \cup t_1$ to $s_2 \cup t_2$. According to definition 6, $s_1 \cup t_1$ and $s_2 \cup t_2$ are isomorphic. Figure 8 shows three different unions of feature trees f_1 and f_2. The corresponding canonical reconstruction forms can be (a): [4,3,f_1, f_2]; (b): [1,1,f_1, f_2]; (c):[1 2,1 4,f_1, f_2].

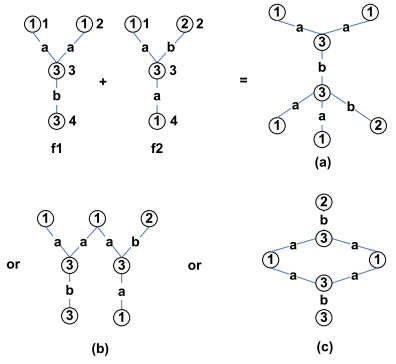


Figure 8. Canonical Reconstruction Form

5.3.2 New Reconstruction Based Subgraph Isomorphism Algorithm

Now we present our new subgraph isomorphism algorithm to determine whether the query graph q is subgraph isomorphic to graph g. It starts from any possible set of feature subtrees TP'_q embedded in g, which is obtained from the filtering and pruning step. However, before the algorithm is performed, we only know every tp'_i in TP'_q is isomorphic to tp_i in q, and its center vertex $center(tp'_i)$ in g. A depth first search is performed first to find the exact feature subtrees rooted in the stored center vertices. Then we reconstruct q by joining subtrees one by one. In each step, we use the canonical reconstruction form to decide if the joining with a new subtree is isomorphic to its counterparts in q. If they are not isomorphic to each other, we will directly move on to the next set of feature subtrees. Otherwise, if the union of all the subtrees in T'_{q_p} turns out to be isomorphic to q, then q is subgraph isomorphic to g.

Algorithm 3 $Verification$

Input: Feature tree set F, Query graph q, TP_q, graph g
Output: Whether $q \subseteq g$

```
1: for each TP'_q satisfied center distance constraint do
2:     if the union of the graphs in TP'_q is isomorphic to q then
3:         Return TRUE
4:         Break
5:     end if
6: end for
7: Return FALSE
```

While the naive subgraph isomorphism test is just an exhaustive search, our algorithm is more effective by making use of the location information obtained from preprocessing phase. For any frequent pattern, the fact is that we already know exactly where it occurs in the graph database during the index construction. However, all previous methods have to discard this important location information due to the limit of memory space and asymmetry nature of indexed frequent patterns. Here, although we only store the centers of frequent trees, the acyclic nature of trees imposes constraints strong enough to allow fast retrieval of the frequent subtrees in the graph database. We only need to compare the union of these retrieved subtrees with the corresponding subgraphs of query graph q, where naive isomorphism tests are avoided by the canonical reconstruct form.

6 Experiments Results

In this section, we will report our experimental results that validate the effectiveness and efficiency of the TreePi algorithm. The performance of TreePi is compared with that of gIndex.

We use two types of datasets in our experiments: one real dataset and a series of synthetic datasets. The real dataset is an AIDS antiviral screen dataset containing 43,905 classified chemical molecules. This dataset is available publicly on the web site of the Developmental therapeutics Program. The synthetic data generator is the same as that in [9]. The generator allows the users to specify the number of graphs, their average size, the number of seed graphs, the average

size of seed graphs, and the number of distinct labels.

All our experiments were performed on a 2.8GHZ, 2GB memory, Intel PC running on RedHat 9.0. Both gIndex and TreePi are compiled with gcc/g++.

6.1 AIDS Antiviral Screen Dataset

In this subsection, we report the experimental results on the antiviral screen dataset. The following parameters are set for gIndex and TreePi. In gIndex, the maximum fragment size $maxL$ is 10, the minimum discriminative ratio γ_{min} is 2.0, and the maximum support Θ is 0.1N. The size-increasing support function $\psi(l)$ is 1 if $l < 4$, in all other cases, $\psi(l)$ is $\sqrt{\frac{1}{maxL}}\Theta$. In TreePi, we set $\alpha = 5$, $\beta = 2$, $\eta = 10$ for the support threshold function, and $\gamma = 1.5$. As we now use a randomized algorithm to partition the query graph, we set $\delta = |q|$, which is relatively large. The same maximum size of features and equivalent number of features are chosen in gIndex and TreePi so that a fair comparison between them can be performed.

We first examine the index size of gIndex and TreePi. The test dataset consisting of N graphs is denoted by Γ_N, which is randomly selected from the antiviral screen database. Figure 9 depicts the number of features used in these two algorithms with the test dataset size varying from 1,000 to 16,000. The curve shows that even though lower thresholds are used in TreePi, the index size of TreePi is still smaller than that of gIndex, which indicates that using frequent tree set as the index structure will result in a smaller index size. As the size of the test dataset increases, the index size of TreePi remains small and stable just as gIndex.

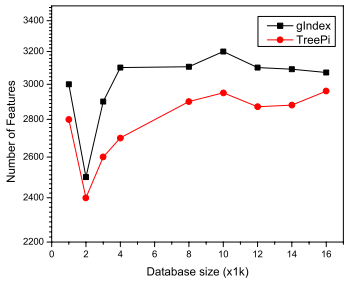
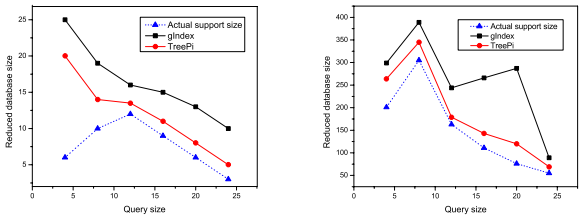


Figure 9. Index Size

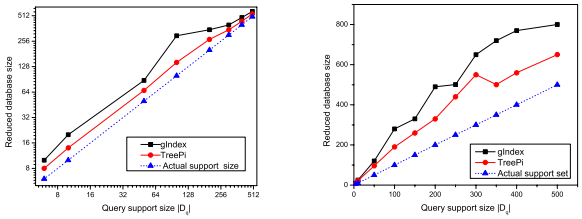
Having verified the index size of gIndex and TreePi, we now examine their effectiveness of pruning false positive candidate graphs. The goal is to reduce the graph database as much as possible to the actual support set of the query graph, since the final verification is relatively more time-consuming. TreePi reduces the graph database in two ways: filtering the database to the intersection of the support sets of feature subtrees, and by the Center Distance Constraints, while gIndex only uses the projection.

In this experiment, $\Gamma_{10,000}$ is selected as the test dataset. For gIndex, six query sets are tested, each of which has



(a) Low Support Queries (b) High Support Queries

Figure 10. Pruning Performance



(a) Real Dataset (b) Synthetic Dataset

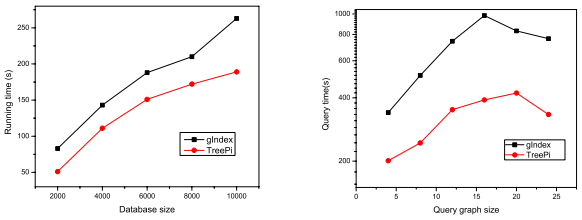
Figure 11. Prune Effectiveness

1,000 queries. We randomly select 1,000 graphs from the antiviral screen dataset and then extract a connected m edge subgraph from each graph randomly. These 1,000 subgraphs are taken as query set, denoted by Q_m. m is selected from 4 to 24. The query sets are divided into two groups, low support group if its support is less than 50 and high support group otherwise. Since frequent patterns are used as index structures, it is crucial to show the index algorithm can perform well on both high support and low support queries.

Figure 10(a) and Figure 10(b) present the pruning performance of gIndex and TreePi on low support queries and high support queries, respectively. We also plot the average size of the actual support set of the query graphs, which is the optimal performance of a pruning algorithm. As shown in the figures, TreePi surpasses gIndex in all query sets of different sizes, especially in large query sets.

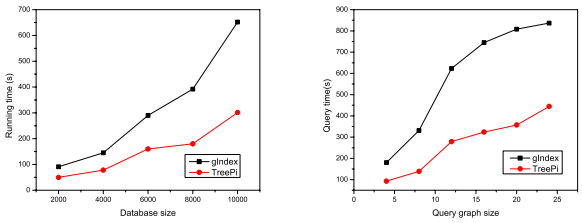
Figure 11(a) shows the pruning effectiveness of both methods on the real dataset. X-axis represents $|D_q|$, the average size of support set of query graphs. Y-axis represents the average size of reduced databases, i.e., $|C_q|$ in gIndex and $|P'_q|$ in TreePi. TreePi outperforms gIndex with $|D_q|$ below 500. The gap between $|D_q|$ and $|P'_q|$ is at least 50% smaller than that between $|D_q|$ and $|C_q|$, which shows TreePi is more powerful in pruning the search spachen $|D_q|$ grows, both gIndex and TreePi perform well.

Figure 12(a) presents the running time in constructing index patterns in both gIndex and TreePi. The database size varies from 2,000 to 10,000 and the index is constructed from scratch for each database. The index construction time of the two methods is both approximately proportional to



(a) Index Construction Time (b) Query Processing Time

Figure 12. Real Dataset



(a) Index Construction Time (b) Query Processing Time

Figure 13. Synthetic Dataset

the database size, while TreePi is relatively faster due to the following reasons: (1) The frequent subtree mining is simpler than the frequent subgraph mining; (2) Computing the canonical forms of trees is much more efficient than that of arbitrary graphs; (3) it is faster to check if a tree T_a is a subtree of tree T_b than to check if an arbitrary graph G_a is a subgraph of G_b, so the index structure shrinking of TreePi is also faster.

In Figure 12(b), the running time for query processing in both gIndex and TreePi is presented. The X-axis represents the edge size of the query graph, and the Y-axis represents the running time to find the corresponding support sets of query graphs. For each edge size, we randomly generate 1000 graphs as query graphs and a graph database of size 6000 is used. Obviously TreePi is much faster than gIndex, since it reduces more than half of the running time for the query graphs of almost all the edge sizes. There are three reasons for this: (1) It takes only polynomial time to retrieve the canonical form of any subtree, so TreePi can prune false positive graphs more efficiently; (2) By applying the Center Distance Constraints, TreePi can reduce the database more effectively as shown in Figure 11(a); (3) TreePi uses a subgraph isomorphism test method, which adopts location information, and is also faster than the naive subgraph isomorphism test in gIndex. Therefore, TreePi is more efficient than gIndex for the exact graph query processing.

6.2 Synthetic Dataset

In this subsection, we present the performance comparison on synthetic datasets. The synthetic graph dataset is generated as follows: First, a set of 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 fragments are then randomly selected and inserted into a graph one by one until the graph reaches its desired size. More details about the synthetic data generator are available in [9]. A typical dataset may have the following settings: it has 8,000 graphs and uses 1,000 seed fragments with 40 distinct labels. On average, each graph has 20 edges and each seed fragment has 10 edges. This dataset is denoted by D8kI10T20S1kL40.

When the number of distinct labels is large, the synthetic dataset is very different from the AIDS antiviral screen dataset. This characteristic results in a simpler index structure. Both gIndex and TreePi work well at this time. However, when the number of distinct labels decreases, more and more vertices and edges share the same labels. The dataset becomes much more difficult to index and search.

Figure 11(b) shows the pruning effectiveness of both methods. The X-axis represents D_q and Y-axis represents the average size of reduced datasets. The testing dataset is D8kI10T20S1kL4. As observed in the figure, the pruning effectiveness of TreePi is about two-fold as that of gIndex on average.

Next the running time for constructing index patterns in both gIndex and TreePi is examined. The database size is varied from 2,000 to 10,000, and there are 5 distinct labels, i.e., the denotation of the datasets is from D2kI10T20S1kL5 to D10kI10T20S1kL5. Figure 13(a) shows the performances of both methods. When the size of the database grows, TreePi can construct the index in a much shorter time.

Last the query time of both gIndex and TreePi is examined. We select a D8kI10T20S1kL5 dataset and vary the edge size of query graphs from 4 to 16. 500 query graphs are randomly generated. As shown in Figure 13(b), in most cases TreePi is more than two times faster than gIndex, which is compatible with the results from AIDS antiviral screen dataset.

7 Discussion

7.1 Insert/Delete Maintenance

In the previous sections, we mostly focus on static indexing, i.e., we did not pay too much attention to updates. Our method can be easily extended to the dynamic indexing. When graph g is inserted to the database, we simply update the support sets and center positions of the existing feature trees. If g contains feature tree t, we will add g into the support set of t, and record t's center position(s) in g. When a graph g' is deleted from the database, we first generate all subtrees below η of g'. If the subtree turns out to be a feature tree, we simply delete all the information of g' stored with the feature tree. This operation is very efficient. From

the experiment results, we can conclude that as long as the graphs in the database are homologous, i.e. representing similar objects, the feature tree set will not change dramatically when the graph database varies. So the maintenance operation is also effective. However, in the case when there are too many insert/delete operations having performed on the database, e.g., one quarter of graphs in the databases are changed, we can reconstruct the feature tree set entirely to ensure the quality of the index.

7.2 Directed Graph Database

When the graphs in the database are directed graph, we should adapt TreePi in following aspects. In the database preprocessing phase, first, the existing graph mining methods should be extended to mine frequent directed trees. Then, while the centers of the trees remain same, the canonical forms of trees should also be adjusted to keep the directions of edges in the preprocessing phase. In query processing phase, we need not make any modification of TreePi. The algorithm adapts well in filtering, pruning, and verification for directed graph queries.

8 Conclusion

Graph querying is a critical problem in the graph database management. In this paper, a new indexing scheme based on frequent subtrees is proposed for graph databases. Trees can preserve more structural information of graphs than simple paths, but are easier to manipulate than other substructures. In our approach, frequent subtrees are first mined from the graph database, then selected as the feature set according to a support threshold and a measure of their representative power. The size of indexed feature set can be controlled by these two parameters, while still capture most information of the original database. In addition, we propose a tree-partition based method for query processing. The search space can be significantly reduced by filtering the database according to the Feature-Tree-Partitions of the query graph. A Center Distance Constraints is introduced to further prune the filtered set. Last, a new subgraph isomorphism test algorithm is devised to perform final verification by utilizing the location information of the indexing structures. Both synthetic and real data sets are used to test our method. The experimental results show that our approach outperforms other graph indexing methods with a wide margin.

9 Acknowledgement

This work was partially supported by NSF CNS-0551603.

References

[1] N. Bruno, L. Gravano, N. Koudas and D. Srivastava. Navigation- vs. index-based XML multi-query processing, *Proc. of ICDE*, 2003.

[2] C. Chung, J. Min, and K. Shim. APEX: an adaptive path index for XML data, *Proc. of SIGMOD,* 2002.

[3] B. Cooper, N. Sample, M. Franklin. A fast index for semistructured data, *Proc. of VLDB,* 2001.

[4] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data, *Proc of PDKK,* 2000.

[5] C. A. James, D. Weininger, and J. Delany. Daylight theory manual daylight version 4.82. Daylight Chemical Information Systems, Inc, 2003.

[6] J. Huan, W. Wang, J. Prins, and J. Yang. SPIN: mining maximal frequent subgraphs from graph databases, *Proc. of KDD,* 2004.

[7] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data, *Proc. of ICDE,* 2002.

[8] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries, *SIGMOD,* 2002.

[9] M. Kuramochi, and G. Karypis. Frequent subgraph discovery, *ICDE,* 2001.

[10] T. Madej, J. Gibrat, and S. Bryant. Threading a database of protein cores, *Proteins,* 1995.

[11] S. Paparizos, and H. Jagadish. Pattern tree algebras: sets or sequences? *VLDB,* 2005.

[12] Z. Chen, J. Gehrke, F. Korn, N. Koudas, and J. Shanmugasundaram. Index structures for matching XML twigs using relational query processors, *ICDE,* 2005.

[13] E. Petrakis, and C. Faloutsos. Similarity Searching in Medical Image Databases, *IEEE TKDE* 1997.

[14] D. Shasha, J. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching, *PODS,* 2002.

[15] A. Shokoufandeh, S. J. Dickinson, and K. Siddiqi. Indexing using a spectral encoding of topological structure, *Proc. of CVPR,* 1999.

[16] X. Yan and J. Han. gSpan: graph-based substructure pattern mining, *ICDM,* 2002.

[17] X. Yan, P. Yu, and J. Han. Substructure similarity search in graph databases, *Proc of SIGMOD,* 2005.

[18] X. Yan, P. Yu, and J. Han. Graph indexing: a frequent structure-based approach, *Proc of SIGMOD,* 2004.

[19] M. Zaki. Efficiently mining frequent trees in a forest: algorithms and applications, *IEEE TKDE,* 2005.