World Wide Web (2008) 11:71–92
DOI 10.1007/s11280-007-0031-z

# Fast Frequent Free Tree Mining in Graph Databases

**Peixiang Zhao · Jeffrey Xu Yu**

Received: 5 January 2007 / Accepted: 10 May 2007 /
Published online: 15 August 2007

**Abstract** Free tree, as a special undirected, acyclic and connected graph, is extensively used in computational biology, pattern recognition, computer networks, XML databases, etc. In this paper, we present a computationally efficient algorithm *F3TM* (Fast Frequent Free Tree Mining) to find all frequently-occurred free trees in a graph database, $\mathcal{D} = \{g_1, g_2, \cdots, g_N\}$. Two key steps of *F3TM* are candidate generation and frequency counting. The frequency counting step is to compute how many graphs in $\mathcal{D}$ containing a candidate frequent free tree, which is proved to be the subgraph isomorphism problem in nature and is NP-complete. Therefore, the key issue becomes how to reduce the number of false positives in the candidate generation step. Based on our observations, the cost of false positive reduction can be prohibitive itself. In this paper, we focus ourselves on how to reduce the candidate generation cost and minimize the number of infrequent candidates being generated. We prove a theorem that the complete set of frequent free trees can be discovered from a graph database by growing vertices on a limited range of positions of a free tree. We propose two pruning algorithms, namely, automorphism-based pruning and canonical mapping-based pruning, which significantly reduce the candidate generation cost. We conducted extensive experimental studies using a real application dataset and a synthetic dataset. The experiment results show that our algorithm *F3TM* outperforms the up-to-date algorithms by an order of magnitude in mining frequent free trees in large graph databases.

**Keywords** structural pattern mining · graph database · free tree

P. Zhao (✉) · J. X. Yu
The Chinese University of Hong Kong, Hong Kong, China
e-mail: pxzhao@se.cuhk.edu.hk

J. X. Yu
e-mail: yu@se.cuhk.edu.hk

## 1 Introduction

The past decade has witnessed an explosive growth of the World Wide Web. As a global information potpourri, the Web flooded us with a substantial amount of data and information, which is frequently modeled as structural patterns like graphs, lattices and trees, etc. [3, 4]. Thus, structural pattern mining was introduced and fulled studied to help discover high quality information and uncover knowledge buried in the World Wide Web.

Graph, a general data structure to represent relations among entities, has been widely used in a broad range of areas such as computational biology, chemistry, pattern recognition, computer networks, etc. In recent years data mining on complex structural patterns has attracted a lot of research interests [11, 20]. The idea of discovering frequent patterns in large graph databases has led to several specialized algorithms for mining frequent subgraphs in graph databases. However, discovering frequent subgraphs from a large graph database comes with high cost. Two computationally expensive operations are unavoidable: (1) to check if a graph contains another graph (in order to determine the frequency of a pattern) is an instance of the subgraph isomorphism problem, which is NP-complete [10]; and (2) to tell if two graphs are isomorphic (in order to avoid creating a candidate multiple times) is an instance of the graph isomorphism problem, which is not known to be either P or NP-complete [10].

With the advent of World Wide Web and the need for mining semi-structured XML data, a particularly useful family of general graph—free tree, has been studied intensively in various areas. Free tree—the connected, acyclic and undirected graph, is a generalization of linear sequential patterns, so that it reserves plenty of structural information of datasets to be mined. At the same time, it is a specialization of general graph, therefore avoids undesirable theoretical properties and algorithmic complexity incurred by graph. As the middle ground between these two extremes, free tree has been widely used in networks, bioinformatics, chemistry, computer vision, etc.

In Web content mining, since XML allows the modeling of a wide variety of databases as tree-structured XML documents, XML thus forms an important data mining domain, and it is valuable to extract frequent patterns from such data. Given a set of such XML documents, lots of applications discover the commonly-occurred subtrees that appear in the collection [21, 26, 27].

In Web usage mining [5, 8], given a database of web access logs from a popular site, one can perform several mining tasks. The simplest is to ignore all link information from the logs, and to mine only the frequent sets of pages accessed by users. It is also possible to look at the entire forward accesses of a user, and to mine the most frequently accessed trees at that site.

In computer networking, multicast free trees are mined and used for packet routing [9]. When there are concurrent multicast groups in the network storing routing tables for all the groups independently, it usually requires considerable space at each router. One possible strategy is to partition the multicast groups and only build a separate routing table for each partition. Here frequent subtrees among the multicast routing trees of different multicast groups offer hints on how to form the partition.

Tree patterns also arise in bioinformatics. In analysis of molecular evolution, an evolutionary free tree, a.k.a. *phylogeny*, is used to describe the evolution history of certain species [12]. Rückert et al. [18] showed how additional constraints can be incorporated into a free tree miner for biochemical databases. Those frequently occurred fragments provide chemists more insight into Quantitative Structure–activity Relationships (QSARs). In pattern recognition, a free tree called *shape axis tree* is used to represent shapes [16]. Free tree has provided us a good compromise between the more expressive, but computationally harder general graph and the faster but less expressive path in data mining research.

## 1.1 Related work

In recent years data mining on complex structural patterns has attracted a lot of research interests [11, 20]. The idea of discovering frequent patterns in large graph databases has led to several specialized algorithms for mining frequent subgraphs in graph databases. Two algorithms by Inokuchi et al. [14] and Kuramochi et al. [15] take advantage of an *a priori*-like approach [1] to mine frequent subgraphs in a graph database. Yan et al. [24, 25] and Huan et al. [13] present subgraph mining algorithms based on a vertical mining approach, which traverses the search space in a depth-first fashion, as opposed to the breath-first traversal used inherently in *a priori*-like algorithms.

Compared with the frequent graph mining problem, mining frequent free trees is computationally less expensive and is proved to be viable in a wide range of applications. In particular, if the background database is composed of trees, determining whether a tree $t_1$ (a candidate frequent pattern) is contained in another tree $t_2$ (a member tree of a tree database) can be solved in $O\left(\frac{m^{3/2}n}{\log m}\right)$, where $m$ and $n$ ($m < n$) are the sizes of $t_1$ and $t_2$ [19]. Meanwhile, to determine whether a free tree, $t_1$, is isomorphic to the other, $t_2$, can be efficiently solved in linear time, w.r.t. the free tree size [2]. Termier et al. [21] present an algorithm to find a subset of frequent trees in a set of tree-structured XML data. Chi et al. [6, 7] present an *a priori*-like algorithm to discover all frequently occurred subtrees in a database of free trees .

If the background database is composed of graphs, there is a need to determine whether a tree pattern is isomorphic to some subtrees of a graph, i.e., a costly *tree-in-graph* testing which is still NP-complete. Rückert et al. propose an algorithm [18] for mining frequent free trees in graph databases. It is important to note that current graph mining algorithms can not be effectively used to tackle the free tree mining problem because they do not fully utilize the characteristics of free tree during mining process. New algorithms should be carefully designed for this special kind of graph to maximize performance.

## 1.2 Our approach

In this paper, we study the issue of mining frequent free trees in a graph database. As a specialized structural pattern mining problem, discovering the complete set of frequent free trees from a graph database has two key steps: *candidate generation* and *frequency counting*. For candidate generation, potential frequent free trees, a.k.a *candidates* are generated from the graph database. The primary operation in this step is free tree isomorphism checking, which can be efficiently done in linear time. For frequency counting, the occurrence of each candidate is computed by traversing the graph database. Infrequent free trees, a.k.a. *false positives*, are eliminated while frequent ones are preserved for further candidate generation. The primary operation in this step is the costly tree-in-graph testing, which is the NP-complete subgraph isomorphism problem in nature. Therefore, the focus of our paper is on candidate generation, because the more the false positives pruned, the less the costly tree-in-graph testings executed. We also observe that the cost of candidate generation itself can be high. In order to solve the free tree mining problem efficiently and effectively, we concentrate ourselves on reducing both the candidate generation cost and the number of candidates to be generated.

The main contributions of our work are summarized below. We propose a vertical mining algorithm, called *F3TM* for **F**ast **F**requent **F**ree **T**ree **M**ining to discover the complete set of frequent free trees from a graph database by a *pattern-growth* approach. Pattern-growth means *F3TM* generates candidates by growing one vertex each time on a frequent free tree. First, we prove that all frequent free trees can be discovered by pattern-growth on the *extension frontier*, a limited range of positions, of frequent free trees. Second, the *automorphism-based pruning* algorithm is proposed to assist us to determine if we need to generate a new candidate from a given frequent free tree by checking the tree itself. Third, the *canonical mapping-based pruning* algorithm is presented during candidate generation. Since there exists a large number of mappings from a free tree to each of its occurrences in a graph, to ensure the completeness of frequent free tree patterns being mined, existing algorithms have to generate a new candidate based on all possible mappings. With canonical mapping, we ensure that there is a unique mapping from a free tree to each of its occurrences in a graph. Therefore, the redundant cost of candidate generation and frequency counting can be significantly reduced. Our extensive experimental studies confirm that our algorithm *F3TM* significantly outperforms the up-to-date algorithms [6, 7, 18] by an order of magnitude in mining frequent free trees from large graph databases.

### 1.3 Roadmap

The rest of the paper is organized as follows. In Section 2, we give our problem statement. Section 3 outlines two existing algorithms for mining frequent free trees in tree and graph databases. We discuss our new algorithm *F3TM* in Section 4, and report results of our extensive performance studies in Section 5. Finally, Section 6 concludes our work.

## 2 Problem statement

A *graph* $G = (V, E, \Sigma, \lambda)$ is defined as a undirected labeled graph where $V$ is a set of vertices, $E$ is a set of edges (unordered pairs of vertices), $\Sigma$ is a set of labels, and $\lambda$ is a labeling function, $\lambda : V \cup E \rightarrow \Sigma$, that assigns labels to vertices and edges. A *free tree*, denoted *ftree*, is a special undirected labeled graph that is connected and acyclic. A *rooted unordered tree* is a *ftree* where a special vertex, which is often referred to as *root*, is distinguished from among vertices in $V$. A *rooted ordered tree* is a *ftree* where root is oriented and the order among children of each vertex is explicitly predefined. In a rooted unordered tree, if vertex $p$ is on the path from the root to vertex $c$ then $p$

is an *ancestor* of $c$ and $c$ is a *descendent* of $p$. If in addition $p$ and $c$ are adjacent, then $p$ is the *parent* of $c$ and $c$ is a *child* of $p$. Notice that a *ftree* is a *unrooted unordered* tree. Below, we call a *ftree* with $n$ vertices a *n-ftree*.

Let $t$ and $s$ be two *ftrees*, and $g$ be a graph. $t$ is a *subtree* of $s$ (or $s$ is the *supertree* of $t$), denoted $t \subseteq s$, if $t$ can be obtained from $s$ by repeatedly removing vertices with degree 1, a.k.a *leaves* of the tree. Similarly, $t$ is a *subtree* of a graph $g$, denoted $t \subseteq g$, if $t$ can be obtained by repeatedly removing vertices and edges from $g$. *Ftrees* $t$ and $s$ are *isomorphic* to each other if there is a one-to-one mapping from vertices of $t$ to vertices of $s$ that preserves vertex labels, edge labels, and adjacency. An *automorphism* is an isomorphism that maps from a *ftree* to itself. A *subtree isomorphism* from $t$ to $g$ is an isomorphism from $t$ to some subtree(s) of $g$.

Given a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_N\}$ where $g_i$ is a graph ($1 \leq i \leq N$). The problem of frequent *ftree* mining is to discover the complete set of frequent *ftrees*. A *ftree*, $t$, is *frequent* if the ratio of graphs in $\mathcal{D}$, that has $t$ as its subtree, is no less than a user-given threshold $\phi$. Formally, let $t$ be a *ftree* and $g_i$ be a graph. We define

$$\varsigma(t, g_i) = \begin{cases} 1 & \text{if } t \subseteq g_i \\ 0 & \text{otherwise} \end{cases}$$

and

$$\sigma\big(t, \mathcal{D}\big) = \sum_{g_i \in \mathcal{D}} \varsigma(t, g_i)$$

$\sigma(t, \mathcal{D})$ denotes the *frequency* of $t$ in $\mathcal{D}$, which is also known as *support* of $t$. A *ftree* $t$ is frequent if (1) holds.

$$\sigma\big(t, \mathcal{D}\big) \geq \phi N \tag{1}$$

In a frequent *ftree* mining algorithm, two main tasks are candidate generation and frequency counting. Candidate generation is to generate potential frequent *ftrees* (candidates) in the graph database. Frequency counting is to calculate $\sigma(t, \mathcal{D})$, where $t$ is a candidate *ftree*. The most important issue of mining frequent *ftrees* in a graph database is to reduce the number of candidate frequent *ftrees* to be generated and counted with minimum overhead, which is the focus of our paper to be studied.

## 3 Two existing algorithms

In this section, we introduce two existing frequent *ftree* mining algorithms. The algorithm proposed by Chi et al. [6] is to discover frequent *ftrees* in a tree database. The algorithm proposed by Rückert et al. [18] is to find frequent *ftrees* in a graph database. We call the former ***FT*** (free tree mining in tree databases), and the latter ***FG*** (free tree mining in graph databases) in this paper. It is worth noting that we study mining frequent *ftrees* in a graph database. But our algorithm can be naturally extended to solve the problem of mining frequent *ftrees* in a tree database. We outline the two algorithms below in brief.

***FT*** is an *a priori*-based algorithm. A conceptual enumeration lattice is built to enumerate all frequent *ftrees* in the graph database. In each step, *FT* generates candidate frequent *ftrees* level-wise. That is, in level ($n$+1) of the enumeration lattice, all candidates of frequent ($n$+1)-*ftrees* are generated using frequent *n-ftrees* found in level $n$. In detail, *FT* follows a pattern–join approach to generate candidate frequent *ftrees*. Two frequent *n-ftrees* which share the same frequent ($n$–1)-*ftree*, which is often referred to as the *core*, are joined to generate a candidate ($n$+1)-*ftree*. *FT* leverages the anti-monotone property of frequent patterns in pruning certain branches of the search space. According to this property, no super trees of an infrequent *ftree* can be frequent.

*FT* is initially proposed to discover all frequent *ftrees* in a database of labeled *ftrees*. When applied in graph databases, this *a priori*-based algorithm is not likely to achieve good performance because the exponential growth of potential frequent *ftrees* will inevitably require a huge memory consumption in the mining process.

***FG*** is a vertical mining algorithm which follows the pattern-growth principle to generate candidate frequent *ftrees*. Given a frequent *n-ftree* $t$, *FG* first counts its frequency by traversing the graph database. At the same time, all vertices at the bottom layer of $t$ are selected as *extension points* to generate candidate $(n+1)$-*ftrees*. A data structure called *extension table* is maintained to record extension points and support values w.r.t. $t$. *FG* builds an enumeration tree upon which the mining algorithm depth-first traverses to discover all frequent *ftrees*. The enumeration tree guarantees an easy pattern-growth process to generate candidate frequent *ftrees* and a backtracking process to efficiently prune the search space. In detail, the mining process may be terminated and backtracked on some branches of the enumeration tree, if the same frequent *ftree* has been found already or the candidate *ftree* is infrequent.

The advantage of the vertical mining algorithm *FG* over the *a priori*-based algorithm *FT* is its relatively small memory consumption. Notice that in order to enumerate candidate frequent $(n+1)$-*ftrees* by *FT*, all frequent *n-ftrees* must be held in memory. The large memory consumption costs a great number of physical page swaps between main memory and disk. However, there still exist several disadvantages for *FG*. First, the candidate generation process to grow vertices on the extension points of a frequent *ftree* may generate redundant candidates, which incurs repetitive computations if no pre-pruning is provided. Second, during each database scan for frequency counting, all occurrences of a *ftree* in the graph database should be computed from scratch, if no further optimization techniques are provided.

## 4 A new fast algorithm: *F3TM*

In this section, we present our frequent *ftree* mining algorithm *F3TM* (**F**ast **F**requent **F**ree **T**ree **M**ining). *F3TM* is a vertical mining algorithm using the pattern-growth approach for candidate generation. We focus ourselves on efficient pruning strategies during candidate generation. Two pruning algorithms, *automorphism-based pruning* and *canonical mapping-based pruning* are proposed to facilitate the candidate generation process and they contribute a dramatic speedup to the final performance of frequent *ftree* mining. Our algorithm *F3TM* can be up to an order of magnitude faster than *FT* and *FG* in mining frequent *ftrees* in large graph databases.

Below, we discuss the following issues of *F3TM*: (1) canonical forms of *ftree*, (2) the enumeration tree, (3) candidate generation, and (4) frequency counting.

## 4.1 Canonical form

Because a *ftree* is unrooted and unordered, it can be possibly represented in multiple ways (We have different choices for the root, and for each non-leaf vertex of the *ftree*, the order of its children is also undetermined). Among multiple representations, we want to select one as *canonical form* to uniquely represent the *ftree* . The canonical form of *ftrees* have the following property: two *ftrees* $t_1$ and $t_2$ share the same canonical form if and only if $t_1$ is isomorphic to $t_2$. The concept of canonical form greatly facilitates storage, indexing and manipulation of *ftrees*. More importantly, during frequent *ftree* mining, only *ftrees* in their canonical form need to be considered, while *ftrees* in non-canonical forms are redundancy and hence can be efficiently pruned.

A canonical form of a *ftree* can be obtained in a two-step algorithm: (1) normalizing a *ftree* to be a rooted ordered tree; (2) assigning a string, as its code, to represent the normalized rooted ordered tree. The final code derived from the two-step algorithm is the canonical form of the *ftree*. Both steps of the algorithm are $O(n)$, for a *n-ftree* [2]. The details are listed below.

**Normalization** The normalization of a *ftree* is based on a root-orientation, children-ordering procedure to transform the tree into a rooted ordered tree. First, the *center*(*s*) of a *ftree t* is distinguished from among vertices of *t* and is selected as root. Then, children of each non-leaf vertex are ordered and rearranged in a pre-defined order. Figure 1 shows two examples: a *central ftree* with one center (Figure 1a) and a *bicentral ftree* with two centers (Figure 1b). Recall that a *ftree* is either a central *ftree* or a bicentral *ftree* [23]. In Figure 1a and b, the left trees are original *ftrees* and center(s) are represented as grey node(s). The middle trees are rooted unordered trees after the center(s) of *ftrees* is distinguished as root (represented as grey node). The right trees are the normalized trees which are both rooted and ordered. The time complexity of normalization of a *n-ftree* is $O(n)$ [2].

**Code Assignment** In the literature, there exist several codes, also known as *canonical string representations* for rooted ordered trees, such as *BFCS* (Breadth-first Canonical String), *DFCS* (Depth-first Canonical String) [7], *canonical form* [18] and *isomorphic code* [23] etc. The canonical string sequentializes normalized *ftrees* for the purpose of efficient manipulation and compact storage. Our algorithm is not sensitive

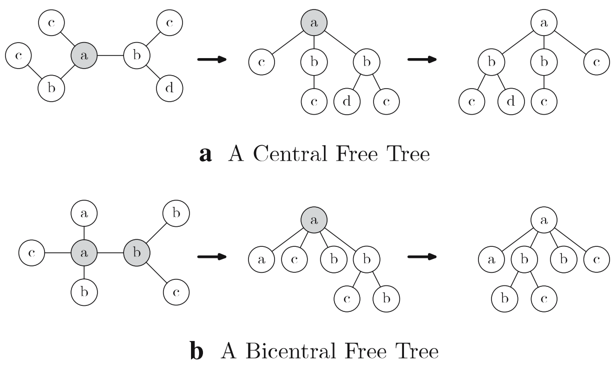


**Figure 1** Central/bicentral ftrees and their canonical forms.

to canonical string representations. Any canonical string can be used in our algorithm effectively.

Given a *ftree* in its canonical form, a.k.a. *canonical ftree*, without loss of generality, we assume that all edge labels of the *ftree* are identical. Because each edge connects a vertex with its parent and we can consider an edge, together with its label, as a part of the child vertex. (For the root of the *ftree*, we can assume that there is a *null* edge connecting to it from above.) In the reminder of the paper, if not specified purposefully, a *ftree* is a canonical *ftree*, and its canonical form or canonical string representation can be used interchangeably.

## 4.2 The enumeration tree

An enumeration tree, $T(V, E)$, is a tree structure representing all frequent *ftrees* of a graph database. Here, $V$ is a set of vertices representing frequent *ftrees*, $E$ is a set of edges representing a subtree-supertree relationship between frequent *ftrees*. The root of $T$ is a virtual vertex, which represents the 0-*ftree*. Children of the root are all frequent 1-*ftrees*. According to the pattern-growth principle, a vertex can be appended to an 1-*ftree* $t$ and candidate frequent 2-*ftrees* originated from $t$ are generated. This procedure continues iteratively, i.e., all $(n+1)$-*ftrees* originated from a $n$-*ftree* $t$ can be obtained from $t$ by the patten-growth approach. In the enumeration tree, the level $n$ contains all frequent $n$-*ftrees* of the graph database. By a depth-first traversal of the enumeration tree, *F3TM* can discover all frequent *ftrees* in the graph database.

Figure 2 shows an example of the enumeration tree. Here, the root node is labeled with ∅, which has three children representing three *ftrees* with label `a`, `b`, and `c`, provided that they are only possible labels. The 1-*ftree* with vertex `a` can grow to three 2-*ftrees*, `a`-`a`, `a`-`b`, and `a`-`c`. As shown in Figure 2, there may be duplicated *ftrees* in the enumeration tree, which means that a *ftree* can be possibly grown from different ancestors in the enumeration tree. For example, the shaded 2-*ftree*, `a`-`b`, can be grown either from 1-*ftree*, `a`, by appending a vertex with label `b`, or from 1-*ftree*, `b`, by appending a vertex with label `a`.

Our *F3TM* algorithm traverses the enumeration tree $T$ in a depth-first manner. When it finds that a *ftree* has been discovered before as a frequent *ftree*, it does not need to mine the duplicated *ftree* and all its descendants again. As discussed above, the key issue is to reduce the possibility of generating duplicated *ftrees* to minimum. We will discuss it in the next section on candidate generation.

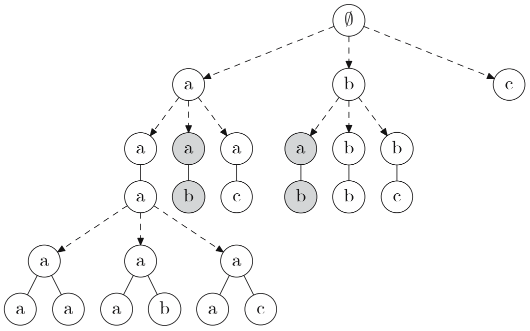


**Figure 2** The enumeration tree.

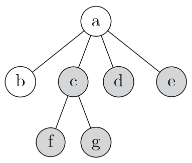


**Figure 3** The extension frontier of a ftree *t*.

## 4.3 Candidate generation

Given a frequent *n-ftree t*, a naive pattern-growth algorithm to generate candidate frequent (*n*+1)-*ftrees* is to add one vertex on each of *n* vertices of *t*. Obviously, this naive algorithm will generate a number of duplicated candidate frequent (*n*+1)-*ftrees*. In the following, we prove that the completeness of frequent *ftrees* in a graph database can be guaranteed even if we grow vertices on the predefined positions of frequent *ftrees*, which are called *extension frontier*. Similar techniques were also applied in [7, 18], which reduces the number of duplicated candidate frequent *ftrees*. To our best knowledge, there is no proof in the reported studies. Below, we first define concepts of *leg*, *last-leg*, and *extension-frontier* of a *ftree*.

**Definition 1** Given a *ftree t*, leaves at the bottom level are called legs. Among all legs, the rightmost leaf is the last leg and the parent of the last leg is denoted as $pl(t)$.

**Definition 2** Given a *ftree t*, the extension-frontier of *t* is composed of three parts:

(1) all legs;
(2) the parent of the last leg, $pl(t)$, and
(3) leaves at the second but last level, whose order are no less than $pl(t)$, or in other words, appear after $pl(t)$.

As shown in Figure 3, the legs of the *ftree t* are vertices `f` and `g`. The last leg is vertex `g`, and the parent of the last leg is vertex `c` $= pl(t)$. The leaves at the second last level, whose order are no less than $pl(t)$, are vertices `d` and `e`. So the extension-frontier of the *ftree t* is composed of vertices `c`, `d`, `e`, `f` and `g`.

**Lemma 1** *Given a n-ftree t, for $n > 2$. Let $t'$ be a canonical $(n–1)$-ftree after deleting the last-leg l from t. The position of vertex $pl(t)$ of t in $t'$ can only have three possibilities: (Case-1) $pl(t)$ is a leg of ftree $t'$, (Case-2) $pl(t) = pl(t')$ and (Case-3) $pl(t)$ appears in the extension-frontier of $t'$, where Case-1 and Case-2 do not hold.*

Figure 4 shows the three cases. We prove Lemma 1 in Appendix.

**Theorem 1** *All frequent ftrees of a graph database can be found by growing vertices on the extension frontier of ftrees in the enumeration tree.*

*Proof* We prove Theorem 1 by induction. It is obvious that the extension frontier of a frequent 1-*ftree* is its sole root, and all frequent 2-*ftrees* can be grown on the extension frontiers of 1-*ftrees*. Assume that all frequent *n-ftrees* can be grown a vertex on the extension frontiers of their corresponding frequent (*n*–1)-*ftrees*. Given a frequent (*n*+1)-*ftree*, *t*, after deleting the last leg *l* from *t*, we get a canonical *n-ftree*, $t'$. Based

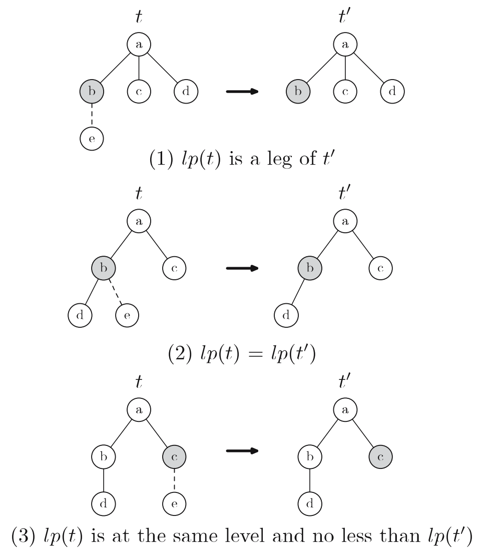

t
t′
a
b
c
d
e
(1) $lp(t)$ is a leg of $t'$
(2) $lp(t) = lp(t')$
(3) $lp(t)$ is at the same level and no less than $lp(t')$

**Figure 4** Three possibilities of $lp(t)$ in $t'$.

on Lemma 1, $pl(t)$ is located on the extension frontier of $t'$. So $t$ can be grown with one more vertex $l$ on the extension frontier of $t'$. Therefore, Theorem 1 holds. □

Theorem 1 demonstrates all legal positions of a frequent *n-ftree* $t'$ on which an additional vertex, i.e., the *last leg*, can be grown to achieve a new candidate frequent ($n$+1)-*ftree* $t$, while no frequent *ftrees* are omitted during this *frontier-extending* process.

## 4.4 Automorphism-based pruning

Based on the specialized pattern-growth approach mentioned above, we can generate the complete set of frequent *ftrees* of a graph database. Suppose there is a set of candidate frequent *ftrees* found already, denoted $\mathcal{T}$. Given a candidate *ftree* $t \in \mathcal{T}$, in order to reduce the cost of frequency counting, we firstly check if there is a *ftree* $t' \in \mathcal{T}$ such as $t = t'$ (*ftree* isomorphim). If it is true, there is no need to count it again. Note: *ftree* isomorphism can be computed in polynomial time, whereas tree-in-graph testing is NP-complete. In other words, it is cost-effective to check whether $t = t'$, for every $t' \in \mathcal{T}$. However, when $\mathcal{T}$ becomes large, the cost of checking $t = t'$ for every $t' \in \mathcal{T}$ can possibly become the dominating cost.

Figure 5 illustrates a concrete example. The leftmost *ftree* $t$ is a frequent 7-*ftree*, where vertices are identified with a unique number as *vertex id*. Since vertex 3 and vertex 5 of $t$ are both located in the extension frontier, we can grow a vertex on either of them to generate a candidate frequent 8-*ftree*. Growing a new vertex on vertex 3 of $t$ generates a candidate frequent 8-*ftree* $t' \in \mathcal{T}$, shown in the middle of Figure 5. While growing the same vertex on vertex 5 of $t$ generates another candidate frequent 8-*ftree* $t''$, shown on the right of Figure 5. Since $t', t'' \in \mathcal{T}$ and $t' = t''$ in the sense of *ftree* isomorphism, the candidate $t''$ can be pruned for further frequency checking. However, if $\mathcal{T}$ is large, the cost of *ftree* isomorphism checking can be prohibitive.

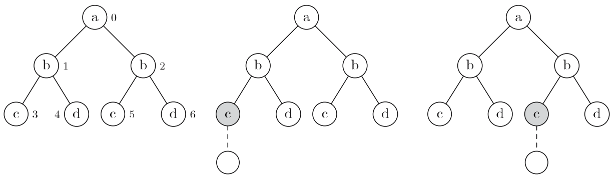


**Figure 5** Automorphism-based pruning.

Based on the observation mentioned above, we need a way to make it possible to grow a new vertex from either vertex 3, or vertex 5 of *t*, but not both, without checking all candidate frequent *ftrees* of $\mathcal{T}$ in advance. Because the candidate frequent *ftree* set, $\mathcal{T}$, may be large. We propose an automorphism-based pruning algorithm in this section to efficiently prune redundant candidates in $\mathcal{T}$ and avoid checking if a candidate frequent *ftree* has existed in $\mathcal{T}$ already, repetitively. Below, we define an equivalence relation of *ftree* based on *ftree* automorphism.

**Definition 3** Two vertices, *u* and *v*, of a *ftree t* satisfy an equivalence relation based on automorphism, if and only if (1) *u* and *v* are at the same level of *t*, (2) subtrees rooted from *u* and *v* are isomorphic, and (3) *u* and *v* share the same parent or their parents hold the equivalence relation based on automorphism.

It is easy to see that the equivalence relation based on automorphism is reflexive, symmetric and transitive. An *equivalence class* is a set of vertices of a *ftree* where every two vertices in the same equivalence class hold the equivalence relation based on automorphism. All vertices in a given *ftree* can be *partitioned* into different equivalence classes. Figure 6 illustrates how to partition vertices of a *ftree* (left tree) into four equivalence-classes (right tree).

It is worth noting that during candidate generation, we only need to grow vertices from one representative of each equivalence class of the *ftree*, if vertices of an equivalence class are in the extension frontier of the *ftree*. In other words, given two vertices *u* and *v* of a frequent *n*-*ftree t*, and assume that they are in the extension-frontier of *t* and in the same equivalence-class, any candidate frequent (*n*+1)-*ftree* that is generated by growing a vertex *x* on *u* is isomorphic to a candidate frequent (*n*+1)-*ftree* that is generated by growing *x* on *v*.

The equivalence classes of a *ftree* can be obtained based on the normalization procedure mentioned in Section 4.1. We explain it using Figure 6. The left *ftree* is normalized where every vertex is associated with a number representing its relative order in a given level of the *ftree*. To compute equivalence classes, we breadth-first traverse the canonical *ftree* and do the following operations in a top-down fashion: first, the root of the *ftree* is the only member of an equivalence class; second, all vertices with the same order at a given level belong to the same equivalence class if their parents belong to the same equivalence class. The procedure continues

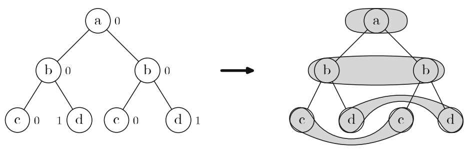


**Figure 6** Equivalence classes of a ftree.

recursively until all vertices of the *ftree* are partitioned to some equivalence classes. Obviously, we can compute equivalence classes of a *n-ftree* in $O(n)$.

The similar concept of equivalence class is also used in *FT* [6]. The equivalence class in *FT* is to guide how to join two frequent *ftrees* and is to ensure the completeness of candidate frequent *ftrees* to be generated. However, it does not necessarily mean that any candidate frequent *ftree* being generated will appear as a subtree in the graph database, i.e., a lot of false positives may be generated during candidate generation, which deteriorates the final performance of the mining algorithm. In our case, the equivalence class based on automorphism is used for pruning. When we select a representative vertex of an equivalence class which is located on the extension frontier of a frequent *ftree* for candidate generation, there is at least one graph in the database that contains the newly generated candidate as subtrees. The cost for counting a *ftree* that does not even appear in the graph database can be significantly reduced in our algorithm.

## 4.5 Canonical mapping-based pruning

With the concept of equivalence class, we show which vertex of a frequent *ftree* is selected to be grown during candidate generation. The selection of such a vertex as a representative of its equivalence class can be determined entirely by the *ftree* itself. As shown in Figure 5, vertex 3 and vertex 5 are in the extension frontier of a *ftree* and in the same equivalence class. We can select either of the two, but not both, to generate candidate frequent *ftrees*. The blank vertex in Figure 5 is the new vertex to be grown, which can be attached with any possible label of the graph database. In this section, we discuss how to assign a reasonable label to, for example, the blank vertex in Figure 5.

A label of a vertex to be grown can be selected from all possible labels appeared in every graph of the graph database, i.e. the alphabet set $\Sigma$. However, this method is infeasible because the number of labels in a graph database might be very large while a majority of candidates generated in this way have little chance to be frequent in the graph database. So a huge number of unnecessary tree-in-graph testings are introduced which is costly as we have discussed above.

In practice, existing algorithms select labels that are truly appeared in the graph database. We outline the idea below. Consider a frequent *n-ftree* $t$ and a graph $g_i \in \mathcal{D}$ and suppose that $t$ occurs $k$ times in $g_i$. The algorithms maintain *mappings* from *ftree* $t$ to all its $k$ occurrences in $g_i$. Based on these mappings, it is possible to know which labels, that appear in graph $g_i$, can be selected and assigned to generate a candidate frequent $(n{+}1)$-*ftree*.

Figure 7 illustrates a simple graph database with two graphs, $\mathcal{D} = \{g_1, g_2\}$, and a frequent *ftree* $t$. Each vertex in *ftree* $t$ and graph $g_1, g_2$ is identified by its vertex id. Every *mapping* between $t$ and an occurrence of $t$, denoted $t'$, a subtree of $g_i$ ($i = 1, 2$) is shown in a mapping list in Figure 7. Suppose the pre-order sequence of vertex ids of $t$ is in a form of $[u_1, u_2, \cdots]$, an entry of the mapping list is shown as $[id(g_i) : v_1, v_2, \cdots]$, where $id(g_i)$ is the graph identifier of $g_i \in \mathcal{D}$ and $v_i$ is the vertex id of $g_i$ so that $u_i$ of $t$ is mapped to $v_i$ of $g_i$. Namely, one mapping entry records one occurrence information between $t$ and $t' \subseteq g_i$. Since the pre-order sequence of vertex ids of $t$ is $[1, 2, 3]$, there are four mappings from $t$ to its two occurrences in $g_1$ (the first four entries in the mapping list) and two mappings from $t$ to its one occurrence

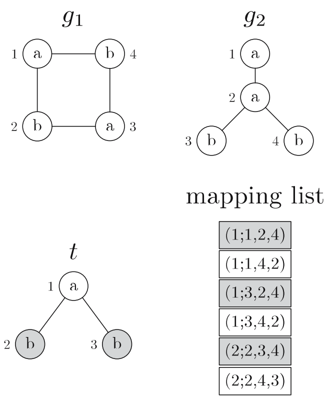


**Figure 7** Mapping and canonical mapping.

in $g_2$ (the last two entries in the mapping list). Existing algorithms will pick up the label of a vertex that connects to the frequent *ftree* and use it as a legal vertex for candidate generation. This method can significantly reduce the number of potential labels assigned to frequent *ftrees* for candidate generation. For example, consider *ftree* $t$ in Figure 7, if we grow a vertex from `b(2)` of $t$ (this representation format means a vertex has a label `b` and its vertex id is 2), the label to be assigned can be `a` based on the first entry in the mapping list, where label `a` is the label of vertex 3 of $g_1$.

However, as shown in Figure 7, there are redundant mappings in the mapping list. Since vertices `b(2)` and `b(3)` of $t$ are in the same equivalence class, the first and second entry in the mapping list: $(1:1,2,4)$ and $(1:1,4,2)$ describe the same occurrence of $t$ in $g_1$ with the only difference of mapping order. It will significantly improve the efficiency of candidate generation, if we remove such redundant mappings. In this section, we propose the concept of *canonical mapping* to efficiently avoid multiple mappings from a *ftree* to the same occurrence of a graph $g_i \in \mathcal{D}$.

**Definition 4** Two mappings are equivalent from *ftree* $t$ to the same occurrence $t'$ of $t$ in a graph $g$, based on all equivalence classes, $C_1, C_2, \cdots$, of $t$, if the set of vertices in $C_i$, for $i = 1, 2, \cdots$, maps to the same set of vertices in $t'$. A canonical mapping is one of such equivalent mappings.

As shown in Figure 7, for the *ftree* $t$, `a(1)` itself is in an equivalence class, $C_1$, and `b(2)` and `b(3)` are in the same equivalence class, $C_2$. There are two mappings from $t$ to an occurrence of $t'$ in graph $g_1$, $(1:1,2,4)$ and $(1:1,4,2)$. The two mappings are equivalent, because in two mappings, `a(1)` in $C_1$ of $t$ maps to the same `a(1)` of $g_1$, and both `b(2)` and `b(3)` in $C_2$ of $t$ map to the same vertices `b(2)` and `b(4)` of $g_1$. We use $(1:1,2,4)$ as the canonical mapping based on the sorting order of each equivalence class. After orienting frequent *ftree* $t$ to its canonical mapping $t'$ of $g_i \in \mathcal{D}$, We can select potential labels from graph $g_i$ for candidate generation, while other non-canonical mappings of $t$ are efficiently pruned.

Given a *n-ftree* $t$, and assume that the number of equivalence classes of $t$ is $c$, and the number of vertices in each equivalence class $C_i$ is $n_i$, for $1 \le i \le c$. The number of mappings between a *ftree* $t$ and an occurrence $t'$ in graph $g_i$ is up to $\prod_{i=1}^{c}(n_i)!$, i.e., the multiplication of all possible permutations of vertices from all equivalence classes of $t$. When either the number of equivalence classes, or the number of vertices in some equivalence class is large, or $t$ frequently occurs in the graph database $\mathcal{D}$, the number of mappings of $t$ in graphs of $\mathcal{D}$ can be huge. With the concept of canonical mapping, we only need to consider one out of $\prod_{i=1}^{c}(n_i)!$ mappings for candidate generation.

## 4.6 Frequency counting

Given a *ftree* $t$, the frequency counting step is to check whether $\sigma(t, \mathcal{D}) \geq \phi N$ Equation (1) holds, which needs to compute tree-in-graph testing, $t \subseteq g_i$ for $g_i \in \mathcal{D}$. Ullmann's backtracking algorithm [22] or Mckay's Nauty algorithm [17] can be applied to tackle this NP complete problem. We design our frequency counting procedure based on Ullmann's approach, but it combines the tree-in-graph testing and candidate generation into one process, thus saving a lot of computations.

**Algorithm 1** *F3TM* ($\mathcal{D}$, $\phi$).

```
Input: the graph database $\mathcal{D}$, the minimum support threshold $\phi$
Output: all frequent ftrees $\mathcal{T}$ satisfying (1)
1: let $\mathcal{T}$ contain all frequent 1-ftrees and 2-ftrees found in $\mathcal{D}$;
2: let $L(t)$ be a list of graph identifiers $id(g_i)$ for the frequent 2-ftree $t$, such that
   $t \subseteq g_i$ and $g_i \in \mathcal{D}$;
3: for all frequent 2-ftree $t \in \mathcal{T}$ do
4:     V-Mine($\mathcal{D}$, $\mathcal{T}$, $t$);
5: return $\mathcal{T}$
```

## 4.7 The *F3TM* algorithm

We outline our *F3TM* algorithm for mining frequent *ftrees* from a graph database in Algorithm 1. It takes two parameters, a graph database $\mathcal{D}$ and a minimum threshold $\phi$. In Algorithm 1, we first find all frequent 1-*ftrees* and 2-*ftrees* and maintain them in $\mathcal{T}$, the set of frequent *ftrees*. Since the number of frequent 1-*ftrees* and 2-*ftrees* is not large, we scan the graph database once to find and count all distinct vertices and edges instead of following the candidate generation and frequency counting procedure, which is time-consuming (line 1). For a frequent *ftree* or a candidate frequent *ftree* $t$ in the mining algorithm, we maintain a list of graph identifiers $id(g_i) \in L(t)$, if $t \subseteq g_i$, for $g_i \in \mathcal{D}$ (line 2). For each frequent 2-*ftree* $t$, we call *V-Mine* to discover all frequent *n-ftrees* $t'$ ($n > 2$) that is a supertree of $t$ by a depth-first traversal of the enumeration tree. Here *V-Mine* implies vertical mining (lines 3–4). The algorithm *V-Mine* takes three parameters, the graph database $\mathcal{D}$, the frequent *ftrees* currently found in $\mathcal{T}$, and the frequent *ftree* $t$ to be examined. *V-Mine* returns the complete set of frequent *ftrees* in line 5.

The *V-Mine* is outlined in Algorithm 2, which recursively generates candidate frequent *ftrees* with one-vertex-growth each time and counts frequency for each candidate iteratively. Suppose the frequent *ftree* $t$ passed to *V-Mine* is a *n-ftree*. Let $\mathcal{X}$ denote the set of all candidate ($n$+1)-*ftrees* that are supertrees of $t$ with one-vertex-growth. Initially, $\mathcal{X}$ is set $\emptyset$ (line 1). In line 2, the algorithm partitions the extension frontier of $t$ into different equivalence classes $F_i$, for $i = 1, 2, \cdots$, which is discussed in Section 4.4. Because we only need to find frequent $(n+1)$-*ftrees* originated from $t$, *V-Mine* considers the graphs in $L(t)$ as the projected graph database for both candidate generation and frequency counting. The candidate generation is given in lines 3–14, and frequency counting is given in lines 15–18.

**Candidate Generation** the algorithm considers all graphs $g$ if $id(g) \in L(t)$ as the projected database for candidate generation (line 3). For each occurrence of $t$, i.e., $t'$ in graph $g$, canonical mapping (Section 4.5) ensures that it only needs to consider one mapping from $t$ to $t'$, and all other non-canonical mappings are efficiently pruned. (line 5). For each extension frontier $F_i$, the algorithm attempts to grow one additional vertex on $F_i$ of $t$. First, it determines a *ftree* $t_c$ to be a candidate *ftree* with one additional vertex $v$ (with no label assigned yet) to be grown on a vertex $u$ in $F_i$ of

**Algorithm 2** *V-Mine*$(\mathcal{D}, \mathcal{T}, t)$.

```
Input: the graph database $\mathcal{D}$, the set of frequent ftrees $\mathcal{T}$, and a frequent ftree $t$;
Output: all frequent ftrees originated from $t$;
 1: $\mathcal{X} \leftarrow \emptyset$;
 2: partition the extension frontier of $t$ into equivalence classes, $F_i$, for $i = 1, \cdots$;
 3: for all graph $g$ if $id(g) \in L(t)$ do
 4:    for all occurrences $t'$ of $t$ in $g$ do
 5:       if the mapping from $t$ to $t'$ is a canonical mapping then
 6:          for all $F_i$ of $t$ do
 7:             let $t_c$ be a candidate ftree with one additional vertex $v$ grow from a
                vertex, $u$, in $F_i$;
 8:             for all potential label $l$ do
 9:                let $v$ in $t_c$ be assigned with label $l$;
10:                canonicalize $t_c$;
11:                $L(t_c) \leftarrow \emptyset$;
12:                if $t_c \notin \mathcal{T}$ then
13:                   $L(t_c) \leftarrow L(t_c) \cup \{id(g)\}$;
14:                   $\mathcal{X} \leftarrow \mathcal{X} \cup \{t_c\}$;
15: for all $t_c \in \mathcal{X}$ do
16:    if support$(t_c) \geq \phi|\mathcal{D}|$ then
17:       $\mathcal{T} \leftarrow \mathcal{T} \cup \{t_c\}$;
18:       V-Mine$(\mathcal{D}, \mathcal{T}, t_c)$;
```

$t$ (line 7). This step can be achieved by selecting one representative from among vertices of $F_i$ and appending the unlabeled vertex upon it. Second, the algorithm selects and assigns a label $l$ to the vertex $v$ grown on $t$ from the potential label set of the corresponding neighbor vertices of the occurrence $t'$ in $g$, based on the canonical mapping (line 9). The algorithm canonicalizes $t_c$ into its canonical form(line 10), and creates an empty list of graph identifiers $L(t_c)$ for the graphs that contain $t_c$ (line 11). If $t_c$ has been found in $\mathcal{T}$ already, there is no need to process it further. Otherwise, the graph identifier $id(g)$ is added to $L(t_c)$, and $t_c$ is appended into $\mathcal{X}$, the candidate frequent *ftree* set, for further frequency counting.

**Frequency Counting** For each candidate frequent *ftrees* $t_c$ in $\mathcal{X}$, the frequency counting procedure computes its frequency, support($t_c$), on $L(t_c)$, its projected graph database. If it is frequent (line 16), the algorithm adds $t_c$ to $\mathcal{T}$, and call *V-Mine* recursively. Otherwise, infrequent *ftrees* are pruned to avoid further consideration.

### 4.8 Algorithm analysis

In this section, we sketch a rough complexity analysis for our frequent *ftree* mining algorithm *F3TM*. Notice that one of the most important operations in *F3TM* is *ftree* manipulation. Given a *n-ftree* $t$, the time complexities of canonicalizing $t$, computing equivalence class of $t$, and identifying canonical mappings of $t$ can all be obtained in $O(n)$.

Let $|\mathcal{T}|$ be the number of frequent *ftrees* in $\mathcal{D}$. Algorithm 2 will be called $|\mathcal{T}|$ times. Let $M$ be the maximum number of canonical mappings of a *ftree* in any graphs of $\mathcal{D}$, $F$ be the maximum number of equivalence classes of the extension frontier of a frequent *ftree*, and $K$ be the maximum number of labels to be assigned to a vertex in an equivalence class of the extension-frontier. The number of tree operations during candidate generation is bounded by a factor of $|\mathcal{T}| \cdot |\mathcal{D}| \cdot M \cdot F \cdot K$. Since $F$ can be bounded by $n$, where $n$ is the maximum size (measured by vertex number) of frequent *ftrees* discovered in $\mathcal{D}$, while $K$ can be bounded by $|\Sigma|$, the operations in candidate generation can be bounded by a factor of $|\mathcal{T}| \cdot |\mathcal{D}| \cdot M \cdot n \cdot |\Sigma|$.

In Algorithm 2, let $|\mathcal{X}|$ be the maximum number of candidate *ftrees* for a given frequent *ftree* $t$, the number of tree-in-graph testing is bounded by a factor of $|\mathcal{T}| \cdot |\mathcal{X}|$. As mentioned above, given a frequent *ftree* $t$, for each equivalence class of its extension frontier, we can grow an additional vertex to generate a candidate for frequency counting. Since the maximum number of equivalence classes of the extension frontier of $t$, i.e., $F$, is bounded by $n$, the maximum size of frequent *ftrees* discovered in $\mathcal{D}$, and the possible number of potential labels is bounded by $|\Sigma|$, so $|\mathcal{X}|$ can be bounded by $n \cdot |\Sigma|$. The final tree-in-graph testing during frequency counting can be bounded by a factor of $|\mathcal{T}| \cdot n \cdot |\Sigma|$.

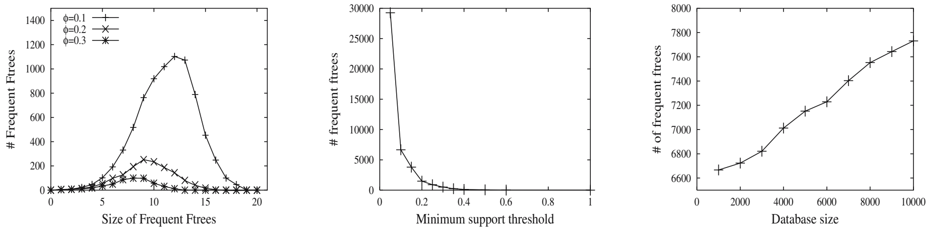
## 5 Performance studies

In this section, we report a systematic performance study that validates the effectiveness and efficiency of our frequent free tree mining algorithm: *F3TM*. We use both a real dataset and a synthetic dataset in our experiments. We implemented *FT* based on [6] and *FG* based on [18] for performance comparison. All experiments were done on a 3.4 GHz Intel Pentium IV PC with 2 GB main memory, running MS Windows XP operating system. All algorithms are implemented in C++ using the MS Visual Studio compiler.

### 5.1 Real dataset

The experiments described in this section use the AIDS antiviral screen dataset from the Developmental Theroapeutics Program in NCI/NIH.[1] This 2D structure

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



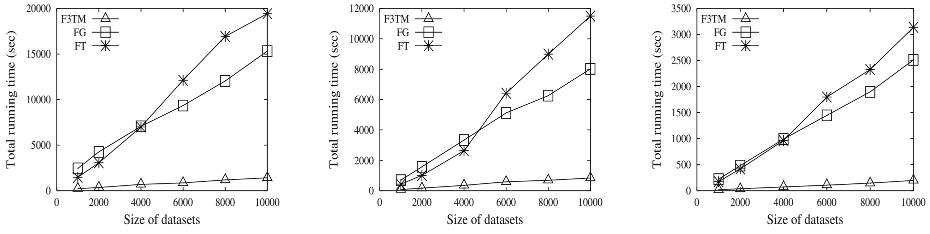
**a** $N = 1000$ **b** $N = 1000$ **c** $\phi = 0.1$

**Figure 8** Characteristics of the real dataset.

dataset contains 42,390 compounds retrieved from DTP's Drug Information System. There are total 63 kinds of atoms in this dataset, most of which are $C$, $H$, $O$, $S$, etc. Three kinds of bonds are popular in these compounds: single-bond, double-bond and aromatic-bond. We take atom types as vertex labels and bond types as edge labels. On average, compounds in the dataset has 43 vertices and 45 edges. The graph of maximum size has 221 vertices and 234 edges. In the following experiments, $n$ is the size of frequent *ftrees* represented as vertex number. $N$ denotes the database size, i.e., $|\mathcal{D}|$ and $\phi$ denotes the minimum threshold.

We show the characteristics of the real dataset being tested. First, we randomly generated a dataset with $N = 1{,}000$ graphs from the AIDS antiviral screen database, and show the number of frequent *ftrees* that have a certain number of vertices for a given minimum threshold $\phi$ in Figure 8a. As shown in Figure 8a, most frequent *ftrees* have 13, 9 and 7 vertices, when $\phi = 0.1, 0.2, 0.3$. respectively. Two conclusions can be made: (1) the number of small *ftrees* with vertex number less than 5 is quite limited and most of them are frequent; (2) although the number of *ftrees* with vertex number greater than 17 grows exponentially, few of them are frequent in the graph database. Second, we use the same randomly selected 1,000 graphs, and show how the number of frequent *ftrees* decreases while varying the minimum threshold $\phi$ 0.05 to 1 in Figure 8b. Third, we fix $\phi = 0.1$ and increase the number of graphs sampled from the AIDS antiviral screen database, and show that the number of frequent *ftrees* increases linearly with the size of the graph database, when $N$ increases up to 10,000.

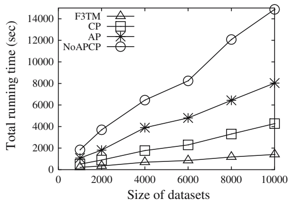
We examine the mining performance of our *F3TM* with *FT* and *FG*, and report the findings in Figure 9. We increase the number of graphs, sampled from the AIDS antiviral screen dataset, from 1,000 to 10,000. Figure 9a, b and c show the



**a** $\phi = 0.1$ **b** $\phi = 0.2$ **c** $\phi = 0.5$

**Figure 9** Performance comparisons on the real dataset.

**Figure 10** Performance comparisons between F3TM and AP, CP, NOAPCP.



mining performance of three algorithms with three different minimum thresholds, $\phi = 0.1, 0.2, 0.5$, respectively. All the three algorithms scale linearly with the size of the graph database. However, *F3TM* outperforms *FT*-algorithm and *FG*-algorithm by an order of magnitude in all experimental settings. These experiments also confirm that *F3TM* can successfully handle large real application data with a broad range of support thresholds.

We also test how much our pruning techniques contribute to the final performance of our mining algorithm. We set $\phi = 0.1$ and vary the graph database size from 1,000 to 100,000. In Figure 10, we use *AP*, *CP*, and *NoAPCP* for automorphism-based pruning enabled, canonical mapping-based pruning enabled, and neither of such pruning techniques enabled, respectively, in addition to *F3TM*, which enables two pruning strategies. As shown in Figure 10, the automorphism-based pruning brings 2.5 to three times speedup for the final performance of our mining algorithm, whereas the canonical mapping-based pruning boosts the final mining performance for four to six times, w.r.t. the trivial mining algorithm, *NoAPCP*.

## 5.2 Synthetic dataset

We generated our synthetic datasets using the widely-used graph generator [15]. The synthetic dataset is characterized by the following parameters: $|\mathcal{D}|$, the number of graphs in the database; $T$, the average size of each graph (in terms of the edge number); $I$, the average size of frequent subgraphs (in terms of edge number); $L$, the number of frequent subgraphs with average size $I$ and $V$, the number of distinct vertex labels in the dataset. Since during graph generation, $L$ is always set to 200, we omit this factor.

Figure 11 shows characteristics of the synthetic database. First, we fix $T = 10$, $I = 5$, $V = 5$, and $\phi = 0.1$. We show in Figure 11a that the number of frequent *ftrees* increases linearly with the size of datasets, when $|\mathcal{D}|$ increases from 1,000 to 10,000.

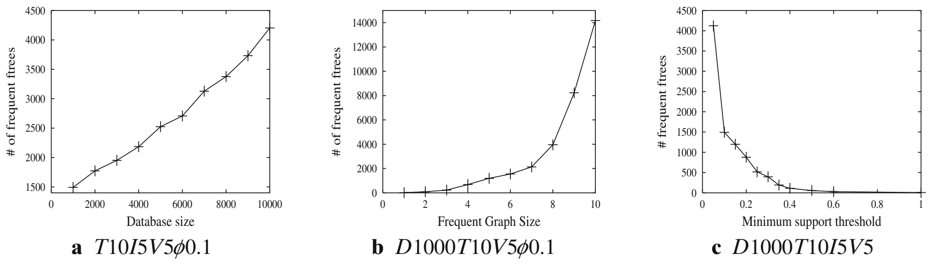
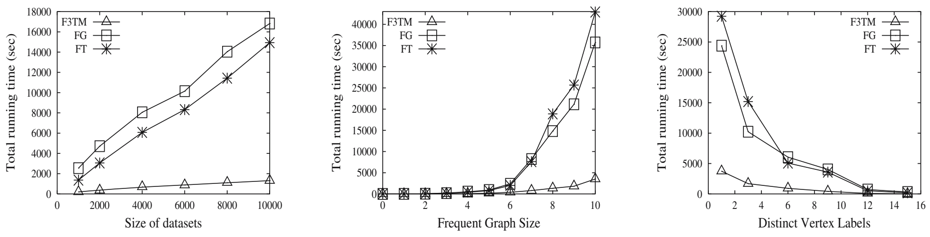


**Figure 11** Characteristics of the synthetic dataset.



**a** $T10I5V5\phi0.1$ **b** $D10000T10V5\phi0.1$ **c** $D10000T10I5\phi0.1$

**Figure 12** Performance comparisons by varying $|\mathcal{D}|$, $I$ and $\phi$.

Second, we fix $|\mathcal{D}| = 1{,}000$, $T = 10$, $V = 5$ and $\phi = 0.1$, and show in Figure 11b that the number of frequent *ftrees* grows exponentially when $I$ increases from 1 to 10. Recall: when the average size of frequent subgraphs increases, all their underlying subtrees become frequent in graph datasets. Third, we fix $|\mathcal{D}| = 1{,}000$, $T = 10$, $I = 5$ and $V = 5$, and show in Figure 11c that the number of frequent free trees drops exponentially when $\phi$ increases, which is similar to the results we have got in the real dataset experiments.

We test the mining performance of *F3TM*, and compare it with *FT* and *FG* in various experimental settings. First, we show the runtime comparison among these three algorithms with fixed $T = 10$, $I = 5$, $V = 5$, and $\phi = 10\%$, when the size of datasets varies from $|\mathcal{D}| = 1{,}000$ to 10,000. As shown in Figure 12a, all three algorithms scale linearly with the size of graph datasets, while *F3TM* outperforms the other two mining algorithms roughly in an order of magnitude. Second, we test the impact of $I$ upon the runtime of three algorithms. As shown in Figure 12b, all three algorithms grow exponentially when $I$ increases. According to the algorithm analysis proposed in Section 4.8, the running time depends on the number of frequent *ftrees* in the graph database (the factor $|\mathcal{T}|$). When $I$ increases, the total number of frequent *ftrees* grows exponentially and hence affects the running time noticeably. The final experiment presents how the running time of algorithms is affected by $L$, i.e., the size of $\Sigma$ in our problem definition. When $|\Sigma|$ is small, both the number of frequent *ftrees* and the canonical mappings between a *ftree* and subtrees in a graph become large, therefore it has a great impact on the total running time. As shown in Figure 12c, we set $|\mathcal{D}| = 10{,}000$ $T = 10$, $I = 5$, and $\phi = 0.1$. When $L$ increases, the running time drops exponentially.

## 6 Conclusion

In this paper, we investigate issues of mining frequent free trees in a graph database. Free tree has several computational advantages over general graph, which make it a suitable candidate for computational biology, pattern recognition, computer networks, XML databases, etc.

We proposed a novel frequent free tree mining algorithm *F3TM* to discover all frequent free trees in a graph database with the focus on reducing the cost for candidate generation. We proved an theorem to guarantee that the complete set of frequent free trees can be achieved based on our pattern-growth approach. We also proposed two pruning algorithms in candidate generation: automorphism-based pruning and canonical mapping based pruning. They both facilitate the frequent free tree mining process drastically. Our experimental studies demonstrates that *F3TM* outperforms the up-to-date existing free tree mining algorithms by an order of magnitude, and *F3TM* is scalable to mine frequent free trees in a large graph database with a low minimum support threshold.

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

## Appendix

In this section, we give a detailed proof of Lemma 1 used in proving Theorem 1. Before presenting the lemma, we introduce some basic concepts in graph theory, which will be used in the proof. Given a free tree $t$, any two vertices $u$, $v$ of $t$ share a unique path and the length of path is defined as the number of edges in the path. The length of the path from vertex $u$ to $v$ is denoted $d(u, v)$, accordingly. The eccentricity of a vertex $u$ is defined as $e(u) = \max_{v \in V[t]}\{d(u, v)\}$. The radius of $t$, denoted as $rad(t)$, is defined as $rad(t) = \min_{u \in V[t]}\{e(u)\}$. The center of $t$ is defined as $center(t) = \{u | e(u) = rad(t)\}$. As mentioned in Section 4, the center of $t$ is either one vertex or two adjacent vertices of $t$, and correspondingly $t$ is denoted as central tree or bicentral tree. We prove Lemma 1 by examining how eccentricity of each vertex changes after $l$ is removed from $t$ and determining the new root of $t'$ and the final positions of $lp(t)$ in $t'$.

*Proof* If $l$ is the unique leg of $t$, we denote the path from $r$ (the root of $t$) to $l$ as $p$ and the length of $p$, i.e., $d(r, l)$ is equal to $c$. So $d(r, lp(t)) = c - 1$. Before $l$ is deleted from $t$, all vertices in $t$, except those on the path $p$, have eccentricities strictly larger than $c$; $e(r)$ is equal to $c$ and all vertices in $p$ except $r$ have eccentricities no less than $c$. After $l$ is deleted, all vertices in $t$, except those on the path $p$, decrease their eccentricities uniformly by 1, so their eccentricities are strictly larger than $(c - 1)$. $r$'s eccentricity changes to $(c - 1)$ and all vertices in $p$ except $r$ do not change their eccentricities, i.e., their eccentricities are still no less than $c$. Above all, the minimum value of eccentricities for all vertices of $t'$ is $(c - 1)$, and the center of $t'$ is $r$. Since $d(lp(t), r) = c - 1$, $lp(t)$ must be at the bottom level of $t'$, i.e., $lp(t)$ is a leg of $t'$ (Case-1).

If there are more than one leg besides $l$ in $t$, we denote them as $l_1, l_2, \cdots, l_k$. There are paths from root $r$ to $l_1, l_2, \cdots, l_k$, we denote them as $p_1, p_2, \cdots, p_k$. So $d(r, l_1) = d(r, l_2) = \cdots d(r, l_k) = d(r, l) = c$, i.e., the length of path $p_1, p_2, \cdots, p_k$ and $p$ are uniformly equal to $c$ and $d(r, lp(t)) = c - 1$. Vertices on each path $p_i$ $(1 \leq i \leq k)$ adjacent to $r$ are referred to as $a_1, a_2, \cdots, a_k$. Before $l$ is deleted from $t$, all vertices in $t$ have eccentricities no less than $c$, and $e(r) = c$. After $l$ is deleted from $t$, all vertices which are not in path $p$ as well as $p_1, p_2, \cdots, p_k$, do not change their eccentricities, i.e., their eccentricities are still no less than $c$ in $t'$. Vertices on path $p$ excluding $r$ do not change their eccentricities either. $e(r)$ is not changed and $e(r) = c$. Vertices in path $p_1, p_2, \cdots, p_k$ may (or may not) change their eccentricities but their eccentricity value are strictly greater than $(c - 1)$, i.e., $rad(t) = c$. So in $t'$, $e(r) = c$ and $e(a_i) = c$ or $(c + 1)$, $(1 \leq i \leq k)$. If (1) $\forall a_i$, where $1 \leq i \leq k$, $e(a_i) = (c + 1)$, then $r$ is selected as root of $t'$. Since $d(r, lp(t)) = c - 1$, $lp(t)$ is still in the second but last level and the relative order of $lp(t)$ at this level of $t'$ does not decrease. If there is some child $l'$ of $lp(t)$ ($l' \neq l$) in $t$, after $l$ is deleted, $l'$ becomes the last leg of $t'$, i.e., $lp(t) = lp(t')$ (Case-2). Otherwise, $l$ is the sole child of $lp(t)$. After $l$ is deleted, $lp(t)$ satisfies Case-3. If (2) $\exists a_i$ ($1 \leq i \leq k$), $e(a_i) = c$, and if $a_i$ is selected as root of $r'$, since $d(a_i, lp(t)) = c$, $lp(t)$ is located at the bottom layer of $t'$, i.e., $lp(t)$ is a leg of $t'$ (Case-1). □

## References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of 20th International Conference on Very Large Data Bases (VLDB94), pp. 487–499 (1994)
2. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Boston, MA (1974)
3. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web (WWW98), pp. 107–117 (1998)
4. Chakrabarti, S., Dom, B.E., Kumar, S.R., Raghavan, P., Rajagopalan, S., Tomkins, A., Gibson, D., Kleinberg, J.: Mining the web's link structure. Computer **32**(8), 60–67 (1999)
5. Chen, Z., Lin, F., Liu, H., Liu, Y., Ma, W.-Y., Wenyin, L.: User intention modeling in web applications using data mining. World Wide Web **5**(3), 181–191 (2002)
6. Chi, Y., Yang, Y., Muntz, R.R.: Indexing and mining free trees. In: Proceedings of the Third IEEE International Conference on Data Mining (ICDM03), Washington, DC, p. 509. IEEE Computer Society, Los Alamitos, CA (2003)
7. Chi, Y., Yang, Y., Muntz, R.R.: Canonical forms for labelled trees and their applications in frequent subtree mining. Knowl. Inf. Syst. **8**(2), 203–234 (2005)
8. Cooley, R., Mobasher, B., Srivastava, J.: Web mining: information and pattern discovery on the world wide web. In: Proceedings of the 9th International Conference on Tools with Artificial Intelligence (ICTAI97), Washington, DC, p. 558. IEEE Computer Society, Los Alamitos, CA (1997)
9. Cui, J.-H., Kim, J., Maggiorini, D., Boussetta, K., Gerla, M.: Aggregated multicasta comparative study. Cluster Comput. **8**(1), 15–26 (2005)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-completeness. Freeman, New York (1979)
11. Han, J., Yan, X., Yu, P.S.: Mining and searching graphs and structures. In: Proceeding of the 22th International Conference on Data Engineering (ICDE06), Philadelphia, PA. IEEE Computer Society Press, Los Alamitos, CA (2006)
12. Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. Discrete Appl. Math. **71**(1–3), 153–169 (1996)
13. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: Proceedings of the Third IEEE International Conference on Data Mining (ICDM03), Washington, DC, p. 549. IEEE Computer Society, Los Alamitos, CA (2003)
14. Inokuchi, A., Washio, T., Motoda, H.: An a priori-based algorithm for mining frequent substructures from graph data. In: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD00), pp. 13–23. Springer, Berlin Heidelberg New York (2000)
15. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM01), pp. 313–320. IEEE Computer Society, Los Alamitos, CA (2001)
16. Liu, T.-L., Geiger, D.: Approximate tree matching and shape similarity. In: International Conference of Computer Vision (ICCV99), pp. 456–462 (1999)
17. Mckay, B.D.: Nauty user's guide. In: Technical Report TR-CS-90-02, the Department of Computer Science. Australia National University (1990)
18. Rückert, U., Kramer, S.: Frequent free tree discovery in graph data. In: Proceedings of the 2004 ACM symposium on Applied computing (SAC04), pp. 564–570. ACM, New York (2004)
19. Shamir, R., Tsur, D.: Faster subtree isomorphism. In: Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems (ISTCS97), Washington, DC, p. 126. IEEE Computer Society, Los Alamitos, CA (1997)

20. Shasha, D., Wang, J.T.L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS02), pp. 39–52. ACM, New York (2002)
21. Termier, A., Rousset, M.-C., Sebag, M.: Treefinder: a first step towards XML data mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM02), Washington, DC, p. 450. IEEE Computer Society, Los Alamitos, CA (2002)
22. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. Assoc. Comput. Mach. **23**(1), 31–42 (1976)
23. Valiente, G.: Algorithms on Trees and Graphs. Springer, Berlin Heidleberg New York (2002)
24. Yan, X., Han, J.: gspan: graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM02), Washington, DC, p. 721. IEEE Computer Society, DC, Los Alamitos, CA (2002)
25. Yan, X., Han, J.: Closegraph: mining closed frequent graph patterns. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD03), pp. 286–295. ACM, New York (2003)
26. Zaki, M.-M.J.: Efficiently mining frequent trees in a forest: algorithms and applications. IEEE Trans. Knowl. Data Eng. **17**(8), 1021–1035 (2005)
27. Zhao, Q., Bhowmick, S.S., Mohania, M., Kambayashi, Y.: Discovering frequently changing structures from historical structural deltas of unordered XML. In: Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM04), pp. 188–197 (2004)