# Fast Frequent Free Tree Mining in Graph Databases

Peixiang Zhao Jeffrey Xu Yu
Chinese University of Hong Kong, Hong Kong, China
{pxzhao,yu}@se.cuhk.edu.hk

## Abstract

*Free tree, as a special graph which is connected, undirected and acyclic, is extensively used in domains such as 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 discover all frequent free trees in a graph database. We focus ourselves on how to reduce the cost of candidate generation and minimize the number of candidates being generated. We prove a theorem that the completeness of frequent free trees can be guaranteed by growing vertices from a limited range of vertices in a free tree. Two pruning techniques, automorphism-based pruning and pruning based on canonical mapping are proposed which significantly reduce the cost of candidate generation. We conducted experimental studies on a real application dataset and we show that our F3TM outperforms the up-to-date algorithms by an order of magnitude.*

## 1. Introduction

Graph, as 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 interest [6, 14] and has led to several specialized algorithms for mining frequent subgraph patterns in a graph database[9, 10, 16, 8]. However, discovering frequent subgraphs 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 graph pattern) is an instance of the *subgraph isomorphism* problem, which is NP-complete [5]; and (2) to check if two graphs are isomorphic (in order to avoid creating a graph pattern multiple times) is an instance of the *graph isomorphism* problem, which is not known to be in either P or NP-complete [5].

With the advent of XML and the need for mining semi-structured data, a particularly useful family of general graph — free tree, has been studied intensively and applied extensively in various areas. Free tree — the connected, undirected and acyclic graph, is a generalization of linear sequential patterns, and hence 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 various areas such as bioinformatics, chemistry, computer vision, networks, etc. In analysis of molecular evolution, an evolutionary free tree, a.k.a. *phylogeny*, is used to describe the evolution history of certain species [7]. Rückert et al. [13] showed how additional constraints can be incorporated into a free tree miner for biochemical databases. In pattern recognition, a free tree called *shape axis tree* is used to represent shapes [11]. In computer networking, multicast free trees are mined and used for packet routing [4]. 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.

In this paper, we fully study the problem of mining frequent free trees in a database of labeled graphs. We propose a vertical mining algorithm, called *F3TM* for **F**ast **F**requent **F**ree **T**ree **M**ining in a graph database by a *pattern-growth* approach. The main contributions of our work are summarized below. First, we prove that all frequent free trees can be discovered in a graph database by growing vertices from the *extension frontier*, a limited range of vertices in a free tree. Second, we propose an *automorphism-based pruning* technique that assists us to determine if we need to generate a new candidate from a given frequent free tree by checking the given tree itself. Third, we propose a pruning technique based on *canonical mapping* which ensures that there is a unique mapping from a free tree to each of its occurrences in a graph. Therefore, the unnecessary candidate generation and frequency counting operations can be significantly reduced. Our extensive experimental studies confirm that our algorithm significantly outperforms the up-to-date algorithms [2, 3, 13] by an order of magnitude.

The rest of the paper is organized as follows. In Section 2, we give our problem statement. We discuss our new algorithm *F3TM* in Section 3, and report the results of our extensive performance studies in Section 4. Finally, Section 5 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. 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 the vertices of $t$ to the 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 find the set of all frequent *ftrees* where a *ftree*, $t$, is *frequent* if the ratio of graphs in $\mathcal{D}$, that has $t$ as its subtree, is greater than or equal to 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

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

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

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

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 for mining frequent *ftrees* in a graph database is to reduce the number of candidate frequent *ftrees* to be counted with minimum overhead, which is the focus of our paper to be studied.

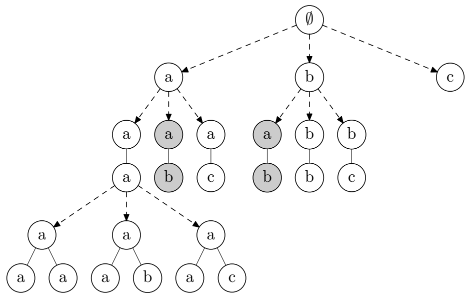


**Figure 1. The enumeration tree**

## 3. A New Fast Algorithm: *F3TM*

In this section, we present our frequent *ftree* mining algorithm *F3TM* (Fast Frequent Free Tree Mining). Below, we discuss the following issues: (1) canonical form of *ftree*, (2) enumeration tree, (3) candidate generation, and (4) frequent counting. We focus on pruning strategies during candidate generation. Two techniques, *automorphism-based pruning* and *canonical mapping pruning* are proposed to facilitate candidate generation and they contribute a dramatic speedup to the final performance of our *ftree* mining algorithm.

### 3.1. Canonical Form

A *ftree* can be possibly represented in different ways, because it is unrooted and unordered. The *canonical form* is a unique representation of a *ftree* in the sense that two *ftrees*, $t_1$ and $t_2$, share the same canonical form if and only if $t_1$ is isomorphic to $t_2$. During frequent *ftree* mining, only *ftrees* in their canonical form need to be considered while others in non-canonical form 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 a *ftree*. Both steps of the algorithm are $O(n)$ for a $n$-*ftree* [1].

### 3.2. The Enumeration Tree

An enumeration tree, $T(V, E)$, is a data structure representing all frequent *ftrees* of a graph database. Here, $V$ is a set of vertices representing canonical frequent *ftrees* (*ftrees* in their canonical form), $E$ is a set of edges representing a subtree-supertree relationship. The root of $T$ is a virtual vertex, which represents the special 0-*ftree*. Children of the root are all frequent 1-*ftrees*. According to the pattern-growth approach, a vertex can be appended to an 1-*ftree* $t$ and all candidate 2-*ftrees* originated from $t$ are gen-

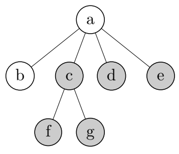


**Figure 2. The extension frontier of a ftree**

erated. This procedure continues iteratively, i.e., all ($n$+1)-*ftrees* originated from a $n$-*ftree* $t$ can be obtained from $t$ by the pattern-growth approach. By pre-order traversal of the enumeration tree, *F3TM* can discover all frequent *ftrees* in the graph database.

Figure 1 shows an enumeration tree example. Here, the root node is labeled with $\emptyset$, which has three children, three *ftrees* with label a, b, and c, provided that they are only possible labels. The *ftree* with one vertex a can grow to three 2-*ftrees*, a−a, a−b, and a−c. As shown in Figure 1, 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 *ftree*, a−b, can be grown either from *ftree*, a, by appending a vertex with label b, or from *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 frequent, 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.

## 3.3. Candidate Generation

Given a $n$-*ftree* $t$, a naive algorithm of generating candidate frequent ($n$+1)-*ftrees* is to add one vertex on each of $n$ vertices of $t$, which will generate a large number of duplicated ($n$+1)-*ftrees*. In the following, we prove that the complete set of candidate *ftrees* is ensured even if we grow vertices on the predefined positions of a *ftree*, called *extension frontier*. Below, we first define concepts of *leg*, *last-leg*, and *extension-frontier* of a *ftree*.

**Definition 1.** *Given a canonical ftree $t$, a leaf at the bottom level is called a leg. 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 canonical ftree $t$, the extension-frontier of $t$ is composed of three part: (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 2, the legs of the *ftree* $t$ are vertices f and g. The last leg is g, and the parent of the last leg is vertex c $= pl(t)$. The leaves at the second last level, whose

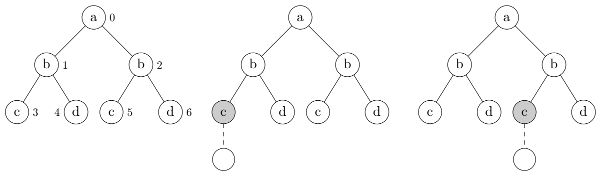


**Figure 3. Automorphism-Based Pruning**

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 canonical $n$-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 3 possibilities: (1) $pl(t)$ is a leg of ftree $t'$, (2) $pl(t) = pl(t')$ and (3) $pl(t)$ appears on the extension-frontier of $t'$, where (1) and (2) do not hold.*

Because of the space limitation, we omit the proof for Lemma 1. The detailed proof can be found in [17].

**Theorem 1.** *All candidate frequent ftrees 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 1-*ftree* is its sole root, and all 2-*ftrees* can be grown on the extension frontiers of 1-*ftrees*. Assume that all $n$-*ftrees* can be grown a vertex on the extension frontiers of their corresponding ($n$-1)-*ftrees*. Given a ($n$+1)-*ftree*, $t$, after deleting the last leg $l$ from $t$, we get a canonical $n$-*ftree*, $t'$. Based on Lemma 1, $pl(t)$ is located on the extension frontier of $t'$. So $t$ can be grown with one more vertex $l$ from the extension frontier of $t'$. Therefore, Theorem 1 holds. □

Theorem 1 represents all legal positions of the $n$-*ftree* $t'$ on which the *last leg* can be appended to achieve the new ($n$+1)-*ftree* $t$, while no *ftrees* are omitted during this frontier-extending process.

### 3.3.1 Automorphism-Based Pruning

Based on the pattern-growth approach mentioned above, we can generate the complete set of frequent *ftrees* in the enumeration tree. 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. However, when $|\mathcal{T}|$ becomes large, the cost of checking $t = t'$ for every $t' \in \mathcal{T}$ can possibly become the dominating cost of the whole mining algorithm.

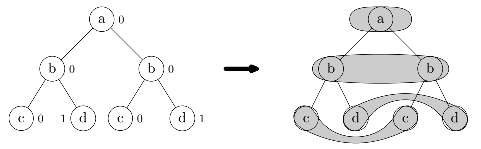


**Figure 4. Equivalence classes of a ftree**

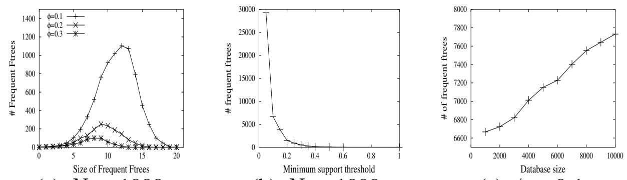
Figure 3 shows a concrete example. The leftmost *ftree* $t$ is a frequent 7-*ftree*, where vertices are identified with a unique number as *vertex id*. Since vertex 3 and vertex 5 of $t$ are both located in the extension frontier of $t$, we can grow a vertex on either of them to generate a candidate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

Base on the observation mentioned above, we need a way to make it possible to grow from either vertex 3, or vertex 5 of $t$, but not both, without checking all candidate *ftrees* in $\mathcal{T}$, since $|\mathcal{T}|$ may be large. We propose an automorphism-based pruning technique in this section which efficiently prunes redundant candidates in $\mathcal{T}$ while avoids checking if a *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 canonical ftree safisfy an equivalence relation based on automorphism, if and only if (1) $u$ and $v$ are at the same level of the ftree, (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.*

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 of the same equivalence class hold the equivalence relation in the *ftree*. All vertices in a given *ftree* can be *partitioned* into different equivalence classes. Figure 4 shows how to partition vertices in 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 an equivalence class, if vertices of the equivalence class are in the extension frontier of the *ftree*. In other words, given two vertices $u$ and $v$ in a $n$-*ftree*, and assume that they are in the extension-frontier and in the same equivalence-class. Any ($n$+1)-*ftree* that grows from $u$ is isomorphic to a ($n$+1)-*ftree* that grows from $v$.



(a) $N = 1000$ (b) $N = 1000$ (c) $\phi = 0.1$

**Figure 5. Characteristics of the real dataset**

### 3.3.2 Pruning Based on Canonical Mapping

Consider a frequent $n$-*ftree* $t$ and a graph $g_i \in \mathcal{D}$ and suppose that $t$ occurs $k$ times in $g_i$, where $k \geq 0$. The mining algorithms maintain *mappings* from $t$ 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 ($n$+1)-*ftree*.

However, there are redundant mappings representing the same occurrence of $t \in g_i$, where $g_i \in \mathcal{D}$, with regards to automorphism of a free tree. *Canonical mapping* is a unique mapping from $t$ to one of its occurrences in $g_i$. After orienting $t$ to its canonical mapping in $g_i \in \mathcal{D}$, we can select potential labels from $g_i$ for candidate generation, while other non-canonical mappings of $t$ are efficiently pruned which facilitates the whole mining process dramatically.

## 3.4. Frequency Counting

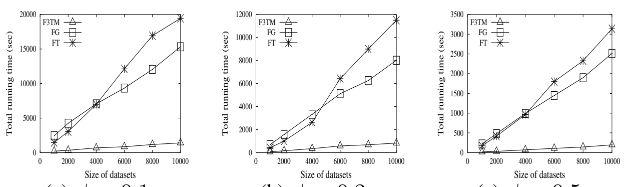
Given a candidate frequent *ftree* $t$, the frequency counting is to check whether $\sigma(t, \mathcal{D}) \geq \phi N$ (Eq. (1)). Ullmann's backtracking algorithm [15] or Mckay's Nauty algorithm [12] can be applied to tackle this NP-complete problem. We design our frequency counting procedure based on Ullmann's approach, but it combines the tree-in-graph testing and candidate generation into one process, thus saving lots of computation.

## 4. Performance Studies

In this section, we report a systematic performance study that validates the effectiveness and efficiency of our algorithm: *F3TM*. We use a real dataset in our experiments. We implemented *FT*-Algorithm based on [2] and *FG*-Algorithm based on [13] for comparison. All experiments were done on a 3.4GHz Intel Pentium IV PC with 2GB main memory, running MS Windows XP operating system. All algorithms are implemented in C++ using the MS Visual Studio compiler.

The experiments use the antiviral screen dataset from Developmental Theroapeutics Program in NCI/NIH[1]. This 2D structure dataset contains 42390 compounds retrieved from DTP's Drug Information System. There are total 63

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



(a) $\phi = 0.1$ (b) $\phi = 0.2$ (c) $\phi = 0.5$

**Figure 6. Performance comparisons**

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 have $43$ vertices and $45$ edges. The graph of maximum size has $221$ vertices and $234$ edges. In the following experiments, $n$ is denoted as the size of frequent *ftrees* represented as vertex number. $N$, the database size, and $\phi$, 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 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 5 (a). As shown in Figure 5 (a), most frequent *ftrees* have vertices varying from $n = 5$ to 15. While the number of small *ftrees* ($n < 5$) and large *ftrees* ($n > 15$) is fairly small. Second, we use the same randomly selected $1,000$ graphs, and show how the number of frequent *ftrees* decreases while the minimum threshold $\phi$ varies from $0.05$ to $1$ in Figure 5 (b). Third, we fix $\phi = 0.1$ and increase the number of graphs sampled from the 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 performance of our *F3TM* with *FT*-Algorithm and *FG*-Algorithm, and report the findings in Figure 6. In Figure 6, we increase the number of graphs, sampled from the antiviral screen dataset, from $1,000$ to $10,000$. Figure 6 (a), (b), and (c) show the performance of the 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, while *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.

## 5. Conclusion

In this paper, we investigate the issue of mining frequent free trees in a graph database. We proposed a novel algorithm *F3TM* to discover all frequent free trees in a graph database with the focus on reducing the cost for candidate generation. We proved a theorem to guarantee the completeness of frequent free trees discovered in graph databases. We also proposed two pruning techniques, automorphism-based pruning and pruning based on canonical mapping. They are applied in candidate generation to greatly facilitate the mining process. Our experimental studies show that *F3TM* outperforms the up-to-date existing algorithms by an order of magnitude and it 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).

## References

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

[2] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proceedings of ICDM03*, 2003.

[3] Y. Chi, Y. Yang, and R. R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowl. Inf. Syst.*, 8(2), 2005.

[4] J.-H. Cui, J. Kim, D. Maggiorini, K. Boussetta, and M. Gerla. Aggregated multicastła comparative study. *Cluster Computing*, 8(1), 2005.

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

[6] J. Han, X. Yan, and P. S. Yu. Mining and searching graphs and structures. In *Proceeding of ICDE06*.

[7] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Appl. Math.*, 71(1-3), 1996.

[8] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of ICDM03*, 2003.

[9] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of PKDD00*, 2000.

[10] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of ICDM01*, 2001.

[11] T.-L. Liu and D. Geiger. Approximate tree matching and shape similarity. In *ICCV99*, 1999.

[12] B. D. Mckay. Nauty user's guide. In *Technical Report, the Department of Computer Science, Australia National University*, 1990.

[13] U. Rückert and S. Kramer. Frequent free tree discovery in graph data. In *Proceedings of SAC04*, 2004.

[14] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of PODS02*, 2002.

[15] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[16] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM02*, 2002.

[17] P. Zhao and J. X. Yu. Fast frequent free tree mining in graph databases. In *Technical Report of Department of SEEM, CUHK*, 2006.