라벨 트리 데이터의 빈번하게 발생하는 정보 추출☆

Frequently Occurred Information Extraction from a Collection of Labeled Trees

백 주 련* 남 정 현** 안 성 준*** 김 응 모****
Juryon Paik Junghyun Nam Sung-Joon Ahn Ung Mo Kim

요 약

트리 데이터로부터 유용한 정보들을 추출하는 가장 일반적인 방식은 빈번하게 자주 발생하는 서브트리 패턴들을 얻는 것이다. XML 마이닝, 웹 사용 마이닝, 바이오인포매틱스, 네트워크 멀티캐스트 라우팅 등 빈번 트리 패턴 마이닝은 여러 다양한 영역에서 광범위하게 이용되고 있기 때문에, 해당 패턴들을 추출하기 위한 많은 알고리즘들이 제안되어 왔다. 하지만, 현재까지 제안된 대부분의 트리 마이닝 알고리즘들은 여러 가지 심각한 문제점들을 내포하고 있는데 이는 특히 대량의 트리 데이터 집합을 대상으로 했을 때는 더 심각해 진다. 주요하게 발생하는 문제점들로는, (1) 계층적 트리 구조의 데이터 모델링, (2) 후보군 유지를 위한 고비용 계산, (3) 반복적인 입력 데이터 집합 스캔, (4) 높은 메모리 의존성이 대표적이다. 이런 문제점들을 발생하게 하는 주요 원인은, 대부분의 기존 알고리즘들이 apriori 방식에 근거하고 있다는 점과 후보군 생성과 빈발 횟수 집계에 anti-monotone 원리를 적용한다는 점에 기인한다. 언급한 문제들을 해결하기 위해, 본 저자들은 apriori 방식 대신 pattern-growth 방식을 기반으로 하며, 빈번 서브트리 추출 대신 최대 빈번 서브트리 추출을 목적으로 한다. 이를 통해 제안된 방법은, 빈번하지 않은 서브트리들을 제거하는 과정 자체를 배제할 뿐만 아니라, 후보군 트리들을 생성하는 과정 또한 전혀 수행하지 않음으로써 전체 마이닝 과정을 상당히 개선한다.

ABSTRACT

The most commonly adopted approach to find valuable information from tree data is to extract frequently occurring subtree patterns from them. Because mining frequent tree patterns has a wide range of applications such as xml mining, web usage mining, bioinformatics, and network multicast routing, many algorithms have been recently proposed to find the patterns. However, existing tree mining algorithms suffer from several serious pitfalls in finding frequent tree patterns from massive tree datasets. Some of the major problems are due to (1) modeling data as hierarchical tree structure, (2) the computationally high cost of the candidate maintenance, (3) the repetitious input dataset scans, and (4) the high memory dependency. These problems stem from that most of these algorithms are based on the well-known apriori algorithm and have used anti-monotone property for candidate generation and frequency counting in their algorithms. To solve the problems, we base a pattern-growth approach rather than the apriori approach, and choose to extract maximal frequent subtree patterns instead of frequent subtree patterns. The proposed method not only gets rid of the process for infrequent subtrees pruning, but also totally eliminates the problem of generating candidate subtrees. Hence, it significantly improves the whole mining process.

☞ KeyWords : Tree mining, Maximal frequent subtree, Embedded tree, Pattern-growth method, 트리 마이닝, 최대 빈번서브트리, 임베디드 트리, 패턴-성장 방식

1. INTRODUCTION

1.1 MOTIVATION

One of the most general approaches for modeling

* 정 회 원 : 성균관대학교 정보통신공학부 연구교수
wise96@ece.skku.ac.kr
** 정 회 원 : 건국대학교 컴퓨터응용과학부 조교수
jhnam@kku.ac.kr
*** 정 회 원 : 성균관대학교 정보통신공학부 부교수
finger@skku.edu
**** 정 회 원 : 성균관대학교 정보통신공학부 교수
umkim@ece.skku.ac.kr

[2009/03/06 투고 - 2009/03/13 심사 - 2009/04/28 심사완료]
☆ 이 논문은 2009년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. 2009-0075771)

complex structured data is to prescribe the data with tree structure. In the database area [1, 2], XML documents are rooted trees where the nodes represent elements or attributes and the edges represent element-subelement and attribute-value relationships. In Web traffic mining, access trees are used to represent the access patterns of different users [3]. In the analysis of molecular evolution, an evolutionary tree is used to describe the evolution history of certain species [4]. In computer networking, multicast trees are used for packet routing [5].

With the ever-increasing amount of available tree data, the ability to extract valuable information from them becomes increasingly important and desirable. However, the problem of finding information on tree data has not been extensively studied, in spite of its applicability to a variety of problems. The first step toward finding information from trees is to mine the subtrees frequently occurring in the trees. Frequent subtrees in a database of trees provide useful knowledge in many cases such as gaining general information of data sources, mining of association rules, classification as well as clustering, and helping standard database indexing [6]. However, the discovery of frequent subtrees appearing in a large-scaled tree dataset is not an easy task. As observed in Chi et al's paper [7], due to combinatorial explosion, the number of frequent subtrees usually grows exponentially with the size (number of nodes) of the tree and, therefore, mining all frequent subtrees becomes infeasible.

A more practical and scalable alternative is thus required, which is the discovery of maximal frequent subtrees. A maximal frequent subtree is a frequent subtree for which none of its proper supertrees are frequent, and the number of them is much smaller than that of frequent subtrees. However, mining maximal frequent subtrees is still in the immature stage and needs to be further researched, compared to the substantial achievements in mining frequent subtrees. Most existing researches on maximal frequent subtrees are inherently complex and cause some computational problems.

1.2 RELATED WORK

The most popular approaches to find useful information from trees are either apriori-based[8] or frequent-pattern-growth(FP)-based[9]. The algorithms based on the former extract frequent subtrees by the well known anti-monotone property: *every non-empty subtree of a frequent tree is also frequent*, for candidate-generate-and-test. Since it provides significant reduction of the size of candidate sets and leads to good performance gain, various techniques have been applied to improve their efficiency [10, 11, 12, 13]. They are efficient and scalable when short patterns are usually extracted from sparse datasets. What if datasets are dense and there are a lot of long patterns? That may degrade mining performance dramatically because a large number of candidates need to be generated and tested.

To solve the problems, FP-growth method is extended to mine tree patterns, which avoids the generation of candidates in support of the construction of concise in-memory data structures that preserve all necessary information, recursively partition an original database into several conditional databases and search for local frequent subtrees to assemble larger global frequent subtrees. However, it is not trivial work for trees because of two major obstacles: one is to *test efficiently* whether a pattern is a subtree of a given tree in a dataset. The other is to determine a good *tree growing* strategy and avoid tree redundancy. The algorithm XSpanner [14] has been recently presented to generate frequent patterns without explicit candidate generation, however, its recursive

projections of a dataset may cause a lot of pointer cashing and bad cache behavior.

The goal of the above mentioned algorithms is to discover *all frequent subtrees* from a database of trees. However, as observed in Chi et al's papers [13], the number of frequent subtrees usually grows exponentially with the tree size, therefore, mining all frequent subtrees becomes infeasible for a large number of trees. The algorithms presented by Xiao et al. [15] and Chi et al. [16] attempt to alleviate the huge amount of frequent subtrees by finding and presenting to end-users only the maximal frequent subtrees. The former uses a new compact data structure, FST-Forest, to store compressed trees, representing the trees in a database. Nevertheless, the algorithm uses post-processing techniques that prune away non-maximal frequent subtrees after discovering all the frequent subtrees. Therefore, the problem of the exponential number of frequent subtrees still remains. The latter directly aims at closed and maximal frequent subtrees only. However, it bases on the enumeration trees, which is one of branches of apriori techniques. Therefore, this algorithm may have the potential problem if a dataset is dense and there are a lot of long patterns.

Handling the maximal frequent subtrees is an interesting challenge, though, and represents the core of this paper.

2. PROBLEM DEFINITIONS

General tree concepts A rooted tree is directed acyclic graph satisfying (1) there is a special node called the "root" that has no entering edges, (2) every other node has exactly one entering edge, and (3) there is a unique path from the root to each node. A tree is a labeled tree if there exists a labeling function that assigns a label to each node of a tree. Let $T = (r, N, E, \mathcal{L})$ be a rooted labeled tree, where $r \in N$ is the root node, N is a set of nodes, E is a set of edges, and $\mathcal{L}$ is a labeling function which maps each node of T to one of labels in a finite set $L = \{l_1, l_2 \ldots l_i\}$; for any node $v \in N$, $\mathcal{L}(v)$ assigns the label of v. For brevity, in the remaining of this paper, unless otherwise specified, we call a rooted labeled tree as simply a tree.

Embedded Subtree Given a tree $T = (r, N, E, \mathcal{L})$, we say that a tree $S = (r', N_S, E_S, \mathcal{L}')$ is included as an *embedded subtree* of T, denoted $S \lesssim T$, iff (1) $N_S \in N$, (2) for all edges $(u, v) \in E_S$ such that u is the parent of v, u is an ancestor of v in T, (3) the label of any node $v \in N_S$, $\mathcal{L}'(v) = \mathcal{L}(v)$. The tree T must preserve ancestor relation but not necessarily parent relation for nodes in S.

Support and frequent subtree The primary goal of mining some set of data is to provide information often occurred in a dataset. However, it is not straightforward in the case for trees unlike the case for traditional item data.

Let $D = \{T_1, T_2, \cdots, T_i\}$ be a set of trees and $|D|$ be the number of trees in D, where $0 < i \leq |D|$. Given D and a tree S, the frequency of S with respect to D, $freq_D(S)$, is defined as $\Sigma_{Ti \in D}\, freq_{Ti}(S)$ where $freq_{Ti}(S)$ is 1 if S is a subtree of T_i and 0 otherwise. The *support* of S with respect to D, $sup_D(S)$, is the fraction of the trees in D that have S as a subtree. That is, $sup_D(S) = freq_D(S) / |D|$. A subtree is called *frequent* if its support is greater than or equal to a minimum value of support specified by users or applications. This user-specified minimum value is often called the *minimum support* (*minsup* or σ). The problem of mining frequent subtrees is defined as to uncover all pattern trees S, such that $sup_D(S) = \Sigma_{Ti \in D}\, freq_{Ti}(S) / |D| \geq minsup$. However, the discovery of frequent subtrees appearing in a large set of trees is not easy task to

do. The combinatorial time for subtree generation becomes an inherent bottleneck of frequent subtree extraction and it causes that finding all frequent subtrees is impossible.

Given some minimum support σ, a subtree *S* is called *maximal frequent* with respect to D iff it satisfies the following conditions: (1) the support of *S* is not less than σ, i.e., $sup_D(S) \geq \sigma$. (2) there exists no any other σ-frequent subtree *S'* with regard to D such that *S* is a subtree of *S'*.

There are fewer maximal frequent subtrees compared to the number of frequent subtrees. In addition, by uncovering only maximal frequent subtrees, we do not lose other frequent information by the fact that the set of maximal ones subsumes all frequent subtrees.

3. THE PROPOSED ALGORITHM

In this section, we introduce a new pattern-growth algorithm SEAMSON (Scalable and Efficient Algorithm for Maximal frequent Subtrees extractiON) based on its interesting definitions and important features. The initial version of SEAMSON was presented in [17, 18].

3.1 LABEL PROJECTION

Finding frequently occurred subtrees is virtually to discover the subtrees whose nodes are labeled by frequently appeared labels in a given tree database D and, therefore, scanning database time to find out how many times each label has been used in D is one of the time consuming part in mining trees. Trees are usually stored in D according to their relating documents and each document is treated as a transaction. That is document-driven layout. In such layout, the whole trees are scanned every time whenever frequency is computed for each label and, thus, it requires $O(|D||T_{avg}||L|)$ time complexity to get the frequencies of whole labels, where |D| is a total number of trees, $|T_{avg}|$ is an average number of nodes of a tree, and |*L*| is a number of distinct labels. It is not serious problem when the number of |D| and $|T_{avg}|$ are reasonably small. It may hinder the computation, however, if both values are large, and actually in real world, two factors are large.

What if the database has been organized in a label-driven layout? A label itself plays the key role which is usually performed by tree or transaction indexes. All trees in D are reorganized according to labels. During scan of the trees in D, all nodes with the same label are grouped together. The nodes composed of the same tree form a member of the group and the number of members actually determines the frequency of the given label; the maximum number of members is a number of trees in D, which is called *label-projection*. After all labels are projected, the document-driven layout is changed into label-driven layout in which the time complexity to check labels' frequency requires at most O(|L||D|). If hash-based search is used, the complexity is reduced up to O(|D|).

Definition 1 (label list) *Let l be a label in L. During pre-ordered scanning trees, tree indexes and node indexes which are projected by l construct a single linked list. It is called a label list and the label list for a given label l is denoted l-list.*

The structure of a label list is similar to that of a linked list in that it has a head and a body. The only difference is the formation and functionality of the head. The head of a label list points the first object in a body just like the ordinary head of a linked list. Along with the indication, the head of a label list gives information on which node indexes have been

mapped to a projected label. The body of a label list follows immediately its corresponding head. The main concerns of a body are to evaluate how many trees have the key in its paired head and to find parents positions of the nodes in the head. The former is for dealing with frequencies of each label, while the latter is for handling the hierarchical information of the label. To achieve such intentions, the structure of a body is a sequence of *members* which is arranged in a linear order. Each member is an object with one key field, one link field pointing to the next member, and one satellite data field.

As a key, a tree index number is used, which means that the label in a corresponding head has been assigned to the nodes of the tree. During the database scan, the member is generated and inserted into bodies of label lists. The newly inserted member is added to the end of a proper body and the pointer field of its previous member points this new member. The body of a label list provides the method we can judge in how many trees have the label of a current label list, that is the number of members in a body and we define it as *size of a label list*.

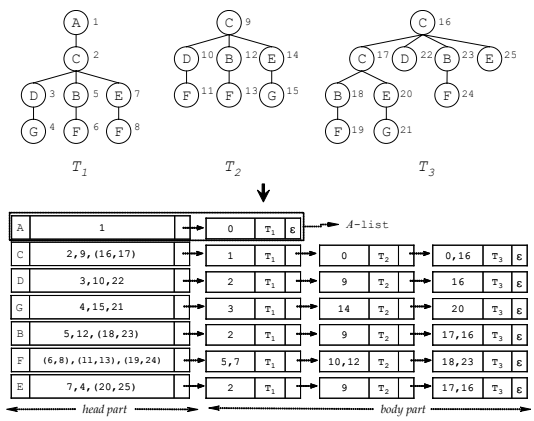
Assume T_1, T_2, and T_3 in D are the trees whose at least one node is labeled by l. The nodes having l in each tree are: $n_a \in T_1$, n_b, $n_c \in T_2$, and $n_d \in T_3$. A number of members of l-list are three because 3 trees include the label. The l-label projection, l-list, is $\langle (p_a, T_1, \rightarrow), (p_b\ p_c, T_2, \rightarrow), (p_d, T_3, \varepsilon) \rangle$, where p_a, p_b, p_c, p_d are parent node indexes, $\rightarrow$ means a pointer to a next member, and ε means an empty pointer.

Definition 2 (label dictionary) *The constructed label lists are collected and stored in the memory. Whenever a label is given, a corresponding label list is retrieved and the count of its members is returned. Due to its activity, the collection is named as label dictionary, denoted* L_{dic}.

As infrequent single-node trees are eliminated in conventional approaches, the label lists which do not confirm a given threshold δ, $\delta = \sigma * |D|$, must be pruned from L_{dic}, because the initial L_{dic} is analogous to a set of all single-node trees of D.

Figure 1 shows an depicted example of how L_{dic} and its label lists are constructed and managed from the database D. For easy distinction between nodes of different trees, we assign unique consecutive indexes in pre-order traversal. The bodies of lists decide the frequencies of corresponding labels. For instance, the label A does not satisfy the given minsup which is set $\frac{2}{3}$ because it has only 1 member in the body of A-list.

Definition 3 (frequent label list) *Given a label list for l, l-list* $\in L_{dic}$, *let |l-list| be a number of its members. l-list is said to be a frequent label list iff it satisfies the following conditions: (1)* $|l\text{-list}| \geq \delta$ *(2) for each member, every parent index p in it, the label of p, L(p), has been projected and has* $\mathcal{L}(p)$*-list. (3)* $|\mathcal{L}(p)\text{-list}| \geq \delta$.



(Figure 1) Original tree database and its label dictionary L_{dic}

A label list is said to be *projected from a frequent label* iff the first condition is satisfied. Since our approach is inspired by the pattern growth algorithms, the label lists whose projected labels do not confirm the first condition cannot be further grown. Even a label list is projected from a frequent label, it is still not clear if the list is a frequent label list or not, because of the structural uniqueness of a label list which is one of the key factors in allowing SEAMSON not to generate any subtrees.

The parent indexes in members should have frequent labels in order the subtrees with size 2 to be frequent, and this is checked by the second and third condition of Definition 3. However, it rarely happens that parent index also has frequent label. In that case, such a parent index whose label is not frequent can be switched to either one of its ancestors which label is frequent, or the root, because SEAMSON cares embedded subtrees. Some means to manage parent indexes in members is required to check the frequency of and to switch them, which makes label lists be frequent.

3.2 CANDIDATE HASH TABLE

Before acquiring frequent label lists, the method to make them must be provided, which is the distinctive feature of SEAMSON and make it differentiate from previous approaches. To this end, the label lists whose projected labels do not satisfy the condition (1) of the definition 3 are excluded from L_{dic} (now, the current L_{dic} is denoted L_{dic}^{f}) and form a special hash table named *Candidate Hash Table*, T_C, whose purpose is to determine if a given node index has a frequent label or not. If its label is found in the table, it means that the label is infrequent because its label list is not in L_{dic}^{f}. The parent indexes of members in L_{dic}^{f} are verified and switched to fulfill the last two conditions of the definition. T_C is composed of node indexes, labels of the nodes, and label lists, where keys are labels and their values which are label lists are returned by a hash function H(*label*). Note that the input of the table is node indexes which are mapped to their proper labels by $\mathcal{L}(n)$.

Theorem 1 *It is infeasible that an identical label list is included in both L_{dic}^{f} and T_C.*

Proof. Assume a label list, say *l*-list, exists in both L_{dic}^{f} and T_C. This situation directly creates conflict; It is reasonable *l*-list has frequent label if it is in L_{dic}^{f}. Hence, it is never excluded from L_{dic}^{f}, which means there is no chance for the list to be in T_C. On the contrary, once *l*-list is in T_C, it indicates the label *l* didn't satisfy a given σ. The label list had to be definitely pruned from L_{dic}. Therefore, any label list generated from a label in *L* is contained in either L_{dic}^{f} or T_C.

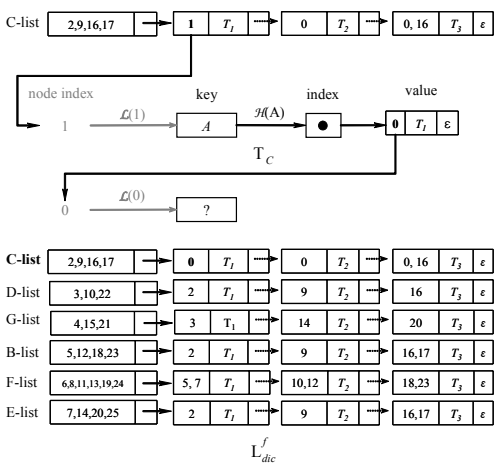
It is easily determined by T_C if the label of a given parent index has projected as one of label lists in L_{dic}^{f}, then, what about the switching in case the label is in T_C. As the required method we define the following.

Definition 4 (closest frequent ancestor) *Given an arbitrary label list in L_{dic}^{f}, any parent index p in its members is required to traverse its ancestors if $\mathcal{L}(p)$ is in T_C. During the move toward the root, the ancestor index of p is searched what is the first ancestor whose label is not in T_C. We call this ancestor closest frequent ancestor of p, denoted Λ_p.*

Let l_1-list ($\in L_{dic}^{f}$) is a current label list and $|l_1\text{-list}|$ be *m*. Each member of its l_1-list.b is required to undertake the following. Let any parent node index in members be *p*. Note that *p*'s n^{th} ancestor is notated by p^n (p^0 is *p* itself). (1) Each *p* of a

member is associated to its label by L(p). (2) The obtained $\mathcal{L}(p)$ is given to T_C. If $\mathcal{L}(p)$ is not found between keys, p has frequent label. Thus, p becomes the proper Λ_p, and the process is terminated. Otherwise, p's label is infrequent. (3) To uncover the desired location of $\mathcal{L}(p)$, the index is computed by H($\mathcal{L}(p)$). (4) According to H($\mathcal{L}(p)$), the value $\mathcal{L}(p)$-list is returned. (5) Since the value is $\mathcal{L}(p)$-list.b, it consists of several members. The node indexes in the members correspond to those of p^1. (6) As backtracking, (1) to (4) is done against every p^1 in the value. (7) The p^1 whose label is found as the key of T_C iteratively performs (3) through (6) until $\Lambda_p{}^1$ is found. For all m members the steps (1) to (7) are performed to fill themselves with only closest frequent ancestors. What if a proper Λ_p is not acquired until end of backtracks? In such case, none of p's ancestors including p itself have frequent labels. That means the node index whose label is l_1 has no parent node, and therefore, it becomes the root. Hence, Λ_p is set by 0 to indicate that 'it is the root position'.

The figure 2 shows T_C constructed simultaneously with $L_{dic}{}^f$ (Here, $L_{dic}{}^f$ is the same as L_{dic} on Figure 1, except that it does not contain A-list and C-list has been modified by the closest frequent ancestor.



(Figure 2) Discovery and replacement of Λ_1

3.3 LABEL LIST EXTENSION

The finally obtained $L_{dic}{}^f$ contains all frequent label lists, which implicitly indicates the projected labels are all frequently occurred and every node index is mapped by one of them. Then, what about paths between nodes if the nodes are explicitly linked? In such case, the paths could possibly be frequent, however, that is not guaranteed because of the fact that a path is a sequence of edges. Let an edge e connect exactly two distinct nodes v_1, v_2 labeled by a and b, which is notated $e = (v_1, v_2) = (\mathcal{L}(v_1), \mathcal{L}(v_2)) = (a, b)$. If e wants to be a frequently occurred edge, the labels a and b must be frequent labels. Hence, if a path wants to be a frequently appeared path, all edges forming the path should be frequently occurred. When let a path be p, it is composed of a finite number of edges; $p = e_1\ e_2 \cdots e_m$. The path can also be expressed with labels because a path is a sequence of labels as shown on the above equations. Therefore, all the labels should be frequent in order the path to be frequent. However, it is not guaranteed when edges between nodes are explicitly unveiled from $L_{dic}{}^f$, because only nodes and labels were considered to build the initial L_{dic}. During the read of label lists, edges are formed by joining a symbolic node whose label is the projected label and symbolic nodes of parent indexes' labels in its members. Unveiling edges totally relies on every frequent label lists, because the symbolic nodes of parent indexes' labels have also their frequent label lists. The hidden paths between label lists are discovered by extending the node of a current label with the nodes of other label lists.

Definition 5 (label list extension) *Given $L_{dic}{}^f$, let a current label list be l-list and p be one of parent indexes in its members. For l-list, firstly a symbolic*

node whose label is l is set and the node is prepared to join with its parent. The second symbolic node is set from the parent index p. Its label is easily obtained by L(p) and the corresponding L(p)-list is in L_{dic}^{f} *due to the definition 3. Consequently, the node labeled by l is joined to the node labeled by L(p). We call this operation label list extension and denote* $l \leftarrow \mathcal{L}(p)$ *where '←' indicates the direction of parent to child.*

Note that the extension is performed with labels not the node indexes. The node index is just used to get label or its corresponding label list. A symbolic node is created whenever a label requires it. The fundamental method is actually to extend label. The label list extension is committed to every label list in L_{dic}^{f}. After completing the work of extension, the labels in head parts are connected each other via nodes. The structure of the result is a tree whose root is labeled by /. This tree contains all of potentially maximal frequent subtrees and thus is named Potentially Maximal Pattern tree (PMP-tree in short). The detailed is followed.

Two times of L_{dic}^{f} scans are required. First, only head parts of label lists are scanned to determine how many symbolic nodes are needed. This number depends on $|L_{dic}^{f}|$ since head part exists per label. The settled number of nodes are created and related by labels in head parts. During the first scan, the nodes are fixed and they are never changed because label list extension is performed on the basis of those nodes and PMP-tree is also derived from them. These nodes are called *seeds* of PMP-tree. Each seed contains three fields; *cnt* field for edge frequency, and two pointer fields: *prt* and *suc* for parent node and successor node, respectively. The detailed roles of fields will be explained in the paragraph for second scanning.

Along with the determination of seeds, one table is constructed with labels and seeds. This table facilitates the traversal of PMP-tree by providing location information of seeds. Once a label is given to the table, its corresponding seed's position in PMP-tree is retrieved. Since the table searches nodes based on labels, we name it *Label Header Table*, T_L, and it is built simultaneously with seeds. T_L is composed of two columns: label and seed# (location of seed).

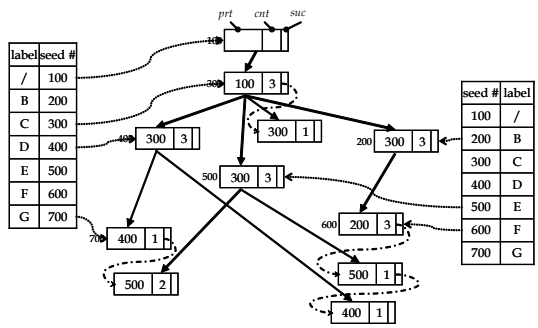
The storage size required for saving the table is followed. Assuming a size of a single table record x and $|L_{dic}^{f}| = M$, the total required space is fixed at xM because seeds are created from the labels of all heads in L_{dic}^{f}. It has a storage complexity of $\Theta(x|L|)$ time in the worst case, however, it is unusual that every unique label for D is frequent. Therefore, the actually necessary space to store T_L is much smaller than the worst case, because of $M \ll |L|$.

After finishing the first scan, total seven seeds are generated from the L_{dic}^{f} on Figure 2 and they are related their labels in T_L via the seed# column. The current seeds have empty parents, empty successors, and 0 counters. Since not yet performed the label list extension, no relation is expected between seeds. To process label list extension over the seeds, L_{dic}^{f} is scanned for the second time and last time. During this scanning period, the body parts in turn are analyzed, which have been skipped at the first scan. The parent indexes of each member are spliced with the existing seeds by following steps: (1) Let a seed of a current reading label list, l-list, be s_l associated with a label l in T_L. (2) For a parent index p in a member of body of l-list, obtain a label of p by $\mathcal{L}(p)$. (3) According to Definition 3, the label $\mathcal{L}(p)$ forms a record of T_L and is associated with one of seeds. (4) Look up the table T_L to get the seed# which corresponds to $\mathcal{L}(p)$; let the corresponding

seed be $s_{L(p)}$. (5) If s_l has empty prt which means it is not yet linked with any parent node, the seed $s_{L(p)}$ is the firstly appeared its parent. Hence, the seed# of $s_{L(p)}$ is stored in the prt of s_l; this action directly perform $s_l \leftarrow s_{L(p)}$. Also, the cnt of $s_{L(p)}$ is incremented by 1 because an edge between s_l and $s_{L(p)}$ has been formed and it occurred. The value of cnt implies frequency of an edge which is formed between a current seed and a parent seed. Instead of incrementing the current seed's cnt, we choose to increment the parent seed's cnt because the parent seed is end point of an edge. Because making edge is completed by the parent seed, the information of edge frequency has to be kept in parent, which means "child seed and I occurred together as many as my cnt times". Thus, we increment the edge frequency in parent seed.

It is trivial work to link s_l to $s_{L(p)}$ when prt of s_l is empty. However, *what if prt of s_l is already preoccupied by another seed#, say $s_{L(q)}$?* There are two cases depending on whether $s_{L(p)} = s_{L(q)}$ or not, and the step (5) in the previous paragraph is replaced by the following: (5-1) If $s_{L(p)} \leftarrow s_{L(q)}$, the fact has to be taken into account that the seed s_l has more than one parents with different labels. To cope with such situation, we add a successor at the end of the seed s_l, or at the last successor if s_l already has successors. Successor has exactly same structure as that of seed. Without using successors one child can have several parents and it causes the graph structure which requires NP-complete subtree decomposition. In case of that s_l has successors, the values in all prts of s_l and its successors have to be compared with the seed# of $s_{L(q)}$. If not found the same value, $s_{L(q)}$ is another parent seed of s_l, thus, a new successor for s_l is attached at the end of successors and it is linked with $s_{L(q)}$. (5-2) In case of that $s_{L(p)}$ is equal to the preoccupied prt value $s_{L(q)}$ or any one of prt values in s_l's successors, cnt of s_l or corresponding successor is incremented by 1. Because the edge of between s_l and $s_{L(p)}$ has been already made, the only work is to increase the frequency; the already existing edge corresponds either to $s_l \leftarrow s_{L(q)}$ or to s_l.suc $\leftarrow s_{L(p)}$ (6) The procedures from step (2) to (5-2) is repeated until all parent indexes in L_{dic}^{f} are considered.

Figure 3 illustrates both the table T_L and the derived PMP-tree from Figure 2. The currently obtained PMP-tree is composed of many edges which link between seeds or successors. Connecting nodes in a tree implies the extension of tree. The frequency of nodes is considered in PMP-tree, however, the frequency of edges is not guaranteed. Therefore, frequency of an edge has to be considered to complete the PMP-tree. This is done by marking the counts of edges in the field cnt of each seed or successor. If s.cnt is less than a given threshold, s and its entering edge are not frequent. Thus, both are spliced out from a current PMP-tree. On the figure, only the edges with bold lines are remained edges in PMP-tree.



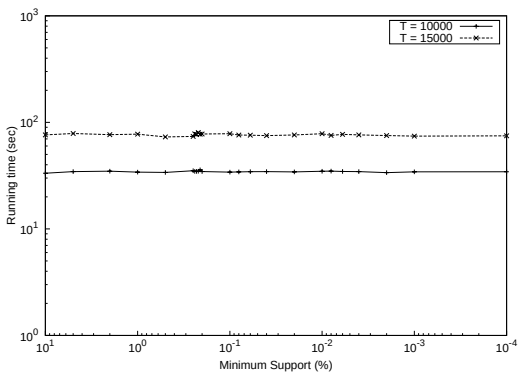
(Figure 3) Final PMP-tree derived from L_{dic}^{f}

4. EXPERIMENTS

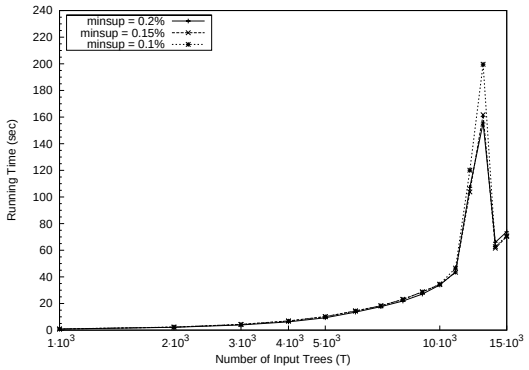
We performed some experiments to evaluate the performance of SEAMSON algorithm using synthetic datasets. All experiments were done on a 2.2GHz

AMD Athlon 64 3500+ PC with 1GB main memory, running Windows XP operating system. All algorithms were implemented in Java. The synthetic datasets are generated by the tree generation program whose underlying ideas are inspired by Termier [19] and Zaki [12]. The generator constructs a set of trees, D, based on some parameters supplied by a user, T: the number of trees in D, *L*: the set of labels, *f*: the maximum branching factor of a node, *d*: the maximum depth of a tree, ρ: the random probability of one node in the tree to generate children or not, η: the average number of nodes in each tree in D. We used the following default values for the parameters: T = 10,000, *L* = 100, *f* = 5, and *d* = 5.

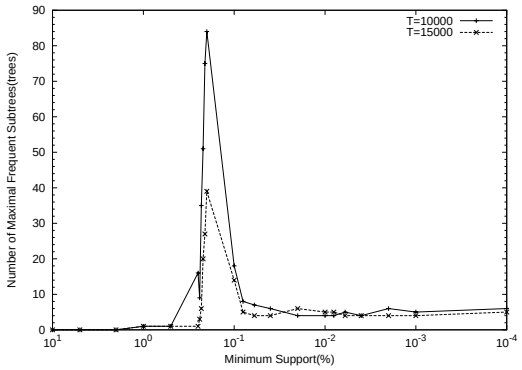
In the following experiments, we first evaluate the scalability of our algorithm with varying minimum support as well as the number of trees T, while other parameters are fixed as: *L* = 100, *f* = 5, *d* = 5, ρ = 20%, η = 13.8. Second, we present the number maximal frequent subtrees under different *minsup*. As the last experiment, the memory usage of SEAMSON is evaluated while *minsup* is 0.2% and 0.1%, where its characterized processes: constructing L_{dic}, finding closest frequent ancestors, and deriving PMP-tree, are especially evaluated for their memory consumption. In the figures of experiments, both X- and Y-axis are drawn on a logarithmic scale for the convenience of observation.



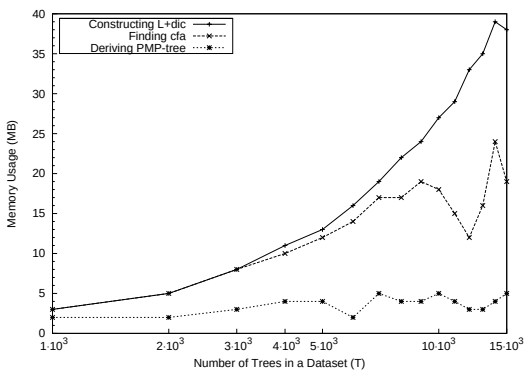
(a) support vs. time



(b) input trees vs. time



(c) number of maximal frequent subtrees



(d) memory usage per each process

(Figure 4) Scalability and memory consumption of SEAMSON

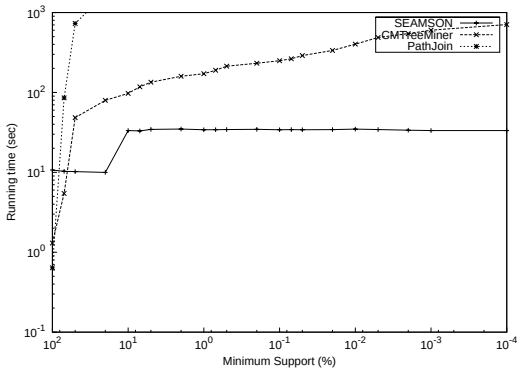
From Figure 4(a), we can find that the running time increases when T increases, however, both running times are rarely affected by the decrease of the minimum support. With *minsup* becoming

smaller, there is no big difference in execution time for both datasets. This is because SEAMSON relies on the number of labels not the number of nodes. Thus, it is very efficient for datasets with varying and growing tree sizes. Then, Figure 4(b) shows the scalability with size of dataset — the number of input trees. The parameter T varies from 1,000 to 15,000 with ρ = 20. We evaluate three different *minsup*, 0.2%, 0.15%, and 0.1%. The corresponding graphs show considerable similarity which slowly increases until T = 11,000 and suddenly goes up between T = 11,000 and T = 13,000. Afterwards, the graphs are started to rapidly deteriorate. Our understanding of this phenomenon is that the sizes of L_{dic}^{f} and its label lists are maximized with 100 distinct node labels when the number of input trees reaches at 12,000 and 13,000.

In Figure 4(c) two graphs have analogous patterns in a number of maximal frequent subtrees; the number of maximal frequent subtrees slowly grows before the rapid rise where lies between *minsup* = 0.3 and *minsup* = 0.2. Afterwards, the number drops off and keeps in steady state. The characteristics stems from the limited number of labels and its random distribution for datasets generation.

Figure 4(d) shows the trends of memory usage of the three processes. In the memory usage, the process for constructing L_{dic} consumes the most amount of memory along with the growing number of trees. On the contrary, the process for PMP-tree consumes almost stabilized amount of memory during the experiments ranged from T = 10,000 to T = 15,000. Because the former is responsible for scanning an original tree dataset and converting them into label projected dataset, the required memory size is getting larger along with the increase of trees in D. The memory usage of the process for finding closest frequent ancestors also increases when T grows; however, its slope is much smooth and gentle compared to that of the first process. It requires less memory because it manages the data in the already generated structure, L_{dic}. At the end, the process for deriving PMP-tree consumes less than 5MB during the experiments. The goal of this process is to derive maximal frequent subtrees from the L_{dic}^{f}.

The following Figure 5 evaluates the running time of SEAMSON in comparison with PathJoin[15] and CMTreeMiner[7]. The parameters for the dataset are: T = 12,000, L = 100, f = 5, and d = 5, η = 20, and minsups from 100% to 0.0001%. PathJoin shows the dramatic increase of time consumption when the minsup decreases. Although it is fast for the minimum values around 100%, it becomes obvious that PathJoin suffers from severe growth of computation time from less than 70% while the other two do not. It stems from the post-processing pruning which is the way PathJoin mines maximal frequent subtrees. After obtaining all frequent subtrees, PathJoin eliminates those that are not maximal. Thus, the number of frequent subtrees is getting bigger as the minsup is getting smaller, and this is the reason why PathJoin requires the worst time consumption compared with SEAMSON and CMTreeMiner. The algorithm CMTreeMiner shows better running time than PathJoin. However, it still gradually increases along with the decrease of the minimum value and requires more time consumption than SEAMSON. This is because of the large amount of candidates generated by apriori-based techniques. Compared to the other two, SEAMSON shows a different trend of time consumption. In spite of decreasing minimum support, it is rarely affected by it. This means that the running time of SEAMSON has fairly stable condition over any minimum support.



(Figure 5) Running time comparison of the three algorithms

5. CONCLUSION

We presented a new concept of label-projection and simple lists and labels based approach to extract maximal frequent subtrees from a database of trees. Unlike the traditional approaches, the proposed method did not perform any subtree generation. To this end, we devised both a special database L_{dic} which adopted the concept of label-projection, and its basic unit, label list, which preserved all necessary information to discover maximal frequent subtrees.

The beneficial effect of our method is that it not only got rid of the process for infrequent tree pruning, but also eliminated totally the problem of candidate subtrees generation. Hence, we significantly improved the whole mining process. To the best of our knowledge, the proposed method in this paper is the first algorithm that directly discovers maximal frequent subtrees without any subtree generation.

References

[1] R. Praveen, M. Bongki, "Prix: Indexing and Querying XML Using Prüfer Sequences," *Proc. of IEEE Int'l Conf. on Data Mining*, pp. 288-299, 2004.

[2] L. I. Rusu, W. Rahayu, T. Taniar, "Mining Changes from Versions of Dynamic XML Documents," *Proc. of Int'l Conf. on Knowledge Discovery from XML Documents, LNCS vol. 3915*, pp. 3-12, 2006.

[3] S. L. T. Adali, M. Magdon-Ismail, "Optimal Link Bombs are Uncordinated," *Proc. of the 1st Workshop on Adversarial Information Retrieval on the Web*, pp. 487-499, 1994.

[4] S. Zhang, J. T. L. Wang, "Mining Frequent Agreement Subtrees in Phylogenetic Databases," *Proc. of the 6th SIAM Int'l Conf. on Data Mining*, pp. 222-233, 2006.

[5] J. Cui, J. Kim, D. Maggiorini, K. Boussetta, M. Gerla, "Aggregated Multicast—A Comparative Study," *Cluster Computing*, 8(1), pp. 15-26, 2005.

[6] Y. Chi, S. Nijssen, R. R. Maggiorini, J. N. Kok, "Frequent Subtree Mining—An Overvire," *Fundamental Informaticae*, 66(1-2), pp. 161-198, 2005.

[7] Y. Chi, Y. Xia, Y. Yang, R. R. Muntz, "Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees," *Proc. of the 16th Int'l Conf. on Scientific and Statistical Database Management*, pp. 11-20, 2004.

[8] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th Int'l Conf. on Very Large Databases*, pp. 487-499, 1994.

[9] J. Han, J. Pei, Y. Yin, "Mining Frequent Pattern without Candidate Generation," *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pp. 1-12, 2000.

[10] K. Wang, H. Liu, "Schema Discovery for Semistructured Data," *Proc. of the 3rd Int'l*

Conf. on Knowledge Discovery and Data Mining, pp. 271-274, 1997.

[11] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, S. Arikawa, "Efficient Substructure Discovery from Large Semi-structured Data," *Proc. of the 2nd SIAM Int'l Conf. on Data Mining*, pp. 158-174, 2002.

[12] M. J. Zaki, "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications," *IEEE Transactions on Knowledge and Data Engineering*, 17(8), pp. 1021-1035, 2005.

[13] Y. Chi, Y. Yang, R. R. Muntz, "Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining," *Knowledge and Information Systems*, 8(2), pp. 203-234, 2005.

[14] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, B. Shi, "Efficient Pattern-growth Methods for Frequent Tree Pattern Mining," *Proc. of the 8th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, LNAI vol.3056*, pp. 441-451, 2004.

[15] Y. Xiao, J.-F. Yao, Z. Li, M. H. Dunham, "Efficient Data Mining for Maximal Frequent Subtrees," *Proc. of IEEE Int'l Conf. on Data Mining*, pp. 379-386, 2003.

[16] Y. Chi, Y. Xia, Y. Yang, R. R. Muntz, "Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees," *IEEE Transactions on Knowledge and Data Engineering*, 17(3), pp. 190-202, 2005.

[17] J. Paik, U. M. Kim, "A Simple yet Efficient Approach for Maximal Frequent Subtrees Extraction from a Collection of XML Documents," *Proc. of the 7th Int'l Conf. on Web Information Systems Engineering*, pp. 94-103, 2006.

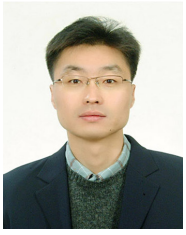
[18] J. Paik, J. Lee, J. Nam, U. M. Kim, "Mining Maximally Common Substructures from XML Trees with Lists-based Pattern-growth Method," *Proc. of IEEE Int'l Conf. on Computational Intelligence and Security*, pp. 209-213, 2007.

[19] A. Termier, M.-C. Rousset, M. Sebag, "Treefinder: A First Step towards XML Data Mining," *Proc. of IEEE Int'l Conf. on Data Mining*, pp. 450-457, 2002.

◐ 저 자 소 개 ◑

백 주 련

1997년 성균관대학교 정보공학과 졸업(학사)
2005년 성균관대학교 대학원 컴퓨터공학과 졸업(석사)
2008년 성균관대학교 대학원 컴퓨터공학과 졸업(박사)
2008 ~현재 성균관대학교 정보통신공학부 연구교수
관심분야 : 트리 마이닝, 지식정보검색, 유사성 검색.
E-mail : wise96@ece.skku.ac.kr



남 정 현

1997년 성균관대학교 정보공학과 졸업(학사)
2002년 Univ. of Louisiana at Lafayette 컴퓨터과학과 졸업(석사)
2006년 성균관대학교 대학원 컴퓨터공학과 졸업(박사)
2007 ~현재 건국대학교 컴퓨터응용과학부 조교수
관심분야 : 암호학, 컴퓨터보안.
E-mail : jhnam@kku.ac.kr

안 성 준

1985년 서울대학교 기계설계학과 졸업(학사)
1987년 한국과학기술원 생산공학과 졸업(석사)
2004년 Univ. Stuttgart 기계공학과 졸업(박사)
1985~1990년 금성사 가전연구소 주임연구원
1990~2004년 Fraunhofer IPA(독) 연구원
2005 ~현재 성균관대학교 정보통신공학부 부교수
관심분야 : 3D Information Processing.
E-mail : finger@skku.edu

김 응 모

1981년 성균관대학교 수학과 졸업(학사)
1986년 Old Dominion Univ. 컴퓨터과학과 졸업(석사)
1990년 Northwestern Univ. 컴퓨터과학과 졸업(박사)
현재 성균관대학교 정보통신공학부 교수
관심분야 : 데이터마이닝, 데이터웨어하우징. 데이터베이스보안
E-mail : umkim@ece.skku.ac.kr