JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 30, 787-817 (2014)

DOI:10.1688/JISE.2014.30.3.15

Method for Extracting Valuable Common Structures from Heterogeneous Rooted and Labeled Tree Data*

JURYON PAIK[1], JUNGHYUN NAM[2], UNG MO KIM[1] AND DONGHO WON[1,+]

[1]*Department of Computer Engineering*
Sungkyunkwan University
Suwon, 440-746 Korea

[2]*Department of Computer Engineering*
Konkuk University
Chungju, 380-701 Korea

The most commonly adopted approach to find valuable information from tree data is to extract frequently occurring subtree patterns. 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) the computationally high cost of the candidate maintenance, (2) the repetitious input dataset scans, and (3) the high memory dependency. These problems stem from the fact that most of the algorithms are based on the well-known apriori algorithm and have used anti-monotone property for candidate generation and frequency counting in them. To solve the problems, we apply the pattern-growth approach instead of the apriori approach, and choose to extract maximal frequent subtree patterns rather than frequent subtree patterns. We present several new definitions and evaluate the effectiveness of the proposed algorithm.

Keywords: tree mining, subtree pattern, maximal frequent subtree, pattern-growth approach, label projection

1. INTRODUCTION

1.1 Motivation

Tree is popular in many research fields thanks to its sufficient expressive power to describe data. In database area [1-7] XML documents are rooted labeled trees where the vertices represent elements or attributes and the edges represent element-subelement and attribute-value relationships. Labels are usually assigned to all vertices of trees via labeling functions. The structure of XML allows the applications to interpret and process information on the Web easily. Moreover, being platform-neutral about the contents published on the Web, it simplifies the integration of existing applications and representation of data in various human readable formats. Currently, it becomes the lingua franca for modeling of a wide variety of data sources as XML documents and produces an enormous amount of information. With the ever-increasing amount of tree data, the researchers in database communities have been required to consider analyzing complex

Received February 7, 2013; revised May 19 & July 13, 2013; accepted August 19, 2013.
Communicated by Vincent S. Tseng.
* This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2010-0020210).
+ Corresponding author.

tree data, such for mining, classification, clustering, indexing and so on.

In web mining, access trees are used to represent accessing patterns of each individual user [8-10]. It improves the quality of web services delivered to users because such patterns provide the web administrators with meaningful information about user access behaviors or usage patterns.

In molecular biology and evolution, a phylogenetic tree is used to show the inferred evolutionary relationships among various biological species [11, 12]. One of the methods is that scientists model the evolutionary history of a set of species that have a common ancestor using rooted unordered labeled trees. Finding cousin pairs, which represent evolutionary relationships between species, in these trees allows much easier understanding of the evolutionary history for those species.

From the above examples, we can see that trees in real applications are often labeled and rooted. Even they model data in different research fields, the practical parts from them are commonly occurred parts having frequent and intentional knowledge, best known as frequent patterns. In case of tree data, patterns mean also trees usually called subtrees. In order to use intentional information from tree data, frequent patterns, frequent subtrees, have to be discovered.

However, the problem of finding subtrees is not easy task to do in spite of its applicability to a variety of subject areas. Because of the hierarchical structure of trees, the traditional information finding methods which are typically applied to the data of flat structure cannot be directly used. Even though trees can be flattened out into a set, this may result in loss of significant structural information. It is not a trivial work to discover useful and common information from a collection of trees, which is the aim of this paper.

1.2 Focus

The first step toward mining information from trees is to find the subtrees frequently occurring in the trees. Frequent subtrees provide useful knowledge in main cases such as gaining general information of data sources, mining of association rules, classification as well as clustering, and helping standard database indexing [13]. However, as observed in Chi *et al.*'s paper [14], due to combinatorial explosion, the number of frequent subtrees usually grows exponentially with the size of a tree, especially when the trees stored in the database are strongly correlated each other. What we can suppose from the exponential growth of the trees follows.

First, the end-users will be overwhelmed by the huge number of frequent subtrees presented to them and, therefore, have difficulty in gaining insights from the frequent subtrees. Second, the mining algorithms may not be easily solved because of the exponentially growing numbers of frequent subtrees. The algorithms presented by Wang and Liu [15] and Xiao *et al.* [16] tried to alleviate the first problem by only finding so called "maximal frequent subtrees" and presenting them to end-users.

A maximal frequent subtree is also a frequent subtree. However, it has a particular condition that is none of its proper supertrees are frequently occurred in a given dataset. Maximal frequent subtrees hold all of frequent subtrees inside according to its definition. Therefore, the number of maximal frequent subtrees is much smaller than that of frequent subtrees. However, finding maximal frequent subtrees is still not fully developed and needs to be further researched. Handling the maximal frequent subtrees is an interesting challenge, though, and represents the core of this paper.

1.3 Contributions

From the perspective of data mining algorithm designs, we consider the challenging problem how efficiently we can reduce the number of subtrees generated to obtain maximal frequent subtrees. The proposed solution relies on such a bijection between the labeled rooted tree and sequence, and leverages its properties to realize an algorithm that is inherently list-based not tree-based. The benefit of the proposed algorithm is that it not only gets rid of the process for infrequent tree pruning, but also totally eliminates the problem of subtree generation, which significantly improves the performance of the whole mining process.

The main contributions of this paper are as follows: (1) It is introduced a concept of label-projected database and described how it is related to a tree database; (2) The proposed algorithm requires only a single time scan of original tree database, which significantly reduces the time cost for the database; (3) For the first time, modified structures of linked-lists are applied to store tree data, which are basic units of a label-projected database. They represent each label of trees in a newly way; (4) A conceptually simple, yet computationally efficient algorithm is fully and thoroughly described, which aim is to discover only maximal frequent subtrees from a rooted and labeled database; (5) The proposed algorithm directly extracts its targets without any subtree generations.

2. RELATED WORKS

The rising of tree data and the need for mining them have sparked a lot of interest in finding frequent trees in forests [17-22]. Wang and Liu [17] considered mining of paths in ordered trees by basing the apriori strategy [23]. It was a start of a journey towards being able to research trees vigorously and a real developmental milestone for them.

One of the best-known algorithms, TreeMiner, developed by Zaki [21] for mining frequent ordered embedded subtrees used the depth-first traversal idea. Also, for efficient subtree counting and manipulation he adopted a novel string representation of tree, so named scope-list. During joining trees, other than the naïve apriori property it took the advantage of a useful property of the string encoding for rooted ordered trees. TreeMiner represented trees in vertical format and uses scoping to prune the search space and efficiently mine for frequent subtrees. A limitation of this method, however, is that it uses pointer-based dynamic data structures and spends a lot of memory.

FREQT algorithm proposed by Asai *et al*. in [18] used an extension only approach to find all frequent induced subtrees from labeled ordered trees. In a preprocessing phase, all frequent labels in the database are determined. Then an enumeration tree is built to enumerate all frequent subtrees. To build an enumeration tree, for each (k+1)-subtree P_{k+1}, one has to identify a unique parent P_k (of size k). In FREQT this problem is solved by removing only the rightmost vertex.

The famous two algorithms, however, had been developed basing on the apriori property, which can suffer from two high computational costs: generating a huge number of candidate sets and scanning the database repeatedly for the frequency counting of candidate sets. That may degrade the mining performance dramatically.

To solve those problems, FP-growth methods [24] have been developed, which

adopt mostly a divide-and-conquer strategy. FP-growth based algorithms avoid candidate sets generation. Instead, they construct a concise in-memory data structure that preserves all necessary information, recursively partitions an original database into several conditional databases, and searches for local frequent subtrees to assemble larger global frequent subtrees. Each conditional database is associated with the data structure and frequent subtrees. Well-known FP-growth based methods are *l*-patterns by Chopper and FST-Forest by PathJoin.

Wang *et al*. [20] had proposed two algorithms, Chopper and XSpanner, to mine frequent embedded subtrees. Chopper recasts subtree mining into sequence mining and uses PrefixSpan [25] to compute the set of frequent sequences, *l*-patterns. These frequent sequences correspond to candidate subtrees that are evaluated against the database and those subtrees that are infrequent are pruned away. The major cost of Chopper comes from two parts. The first part is for mining the *l*-patterns: the pruning of unpromising *l*-patterns improves the performance of the sequential pattern mining. The second part is for checking every one of trees against the *l*-patterns and candidate tree patterns. Such unpromising *l*-patterns may bring unnecessary overhead to the mining. This observation motivates the design of XSpanner, in which mining sequential patterns and extracting frequent subtree patterns are integrated. It recursively mines projected databases. A potential problem is, however, that the recursive projection may lead to a lot of pointer chasing and poor cache behavior.

Xiao *et al.* presented the algorithm PathJoin [16]. Their methodology was based on FP-growth, however, their target results were not frequent subtrees but maximal frequent subtrees. Due to the previously mentioned problems caused by mining frequent subtrees, they mined the special frequent subtrees which none of their proper supertrees are frequent. PathJoin used uses a new compact data structure, FST-Forest, to store compressed trees representing the trees in the database. However, the algorithm applied post-processing techniques that pruned away non-maximal frequent subtrees after discovering all frequent subtrees. Therefore, the problem of the exponential number of frequent subtrees still remains.

Another algorithm described by Chi *et al*. [26] attempted to directly find closed and maximal frequent subtrees only. The algorithm used several pruning and heuristic techniques to reduce the search space that does not correspond to closed and maximal frequent subtrees and to improve the computational efficiency on the generation of closed and maximal frequent subtrees. However, it bases on enumeration trees, which is one of the branches of apriori techniques.

In this paper, we suggest a new algorithm. Its underline approach is taken from FP-growth method and its target result is set to maximal frequent subtrees. The former is for not performing the candidate subtrees generation and the latter is for alleviating the huge amount of frequent subtrees.

3. PRELIMINARIES

3.1 General Tree Definitions

In this subsection we briefly recap tree related well-known definitions first based on the papers [13, 21].

The fundamental tree property for representing data to achieve in the paper is ***rooted labeled trees***. A rooted tree is directed acyclic graph satisfying: (1) there is a special vertex, called a root, which has no entering edges, (2) every other vertex has exactly one entering edge, and (3) there is a unique path from the root to each vertex. A tree is a labeled tree if there is a labeling function that assigns a label to each vertex of a tree. The labels in a tree could be unique, or duplicated labels are allowed for different vertices.

The rooted labeled tree is denoted as $T = (r, N, E, \mathcal{L})$, where $N = \{v_1, v_2, \ldots, v_i\}$ is a set of vertices, $E = \{(u, v) \mid u, v \in N\}$ is a set of edges, $r \in N$ is the root, and $\mathcal{L}$ is a labeling function which maps each vertex of T to one of labels in a set $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_j\}$. For brevity we call the rooted labeled tree simply as a tree in the remains of the paper.

A size of a tree T is defined as a number of vertices the tree has. In the paper, a tree with size k is denoted by a k-tree. A path p in a tree is a sequence of edges, $p = \langle (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n) \rangle$, where $v_n \in N$ $(1 \le n \le i)$, shortly $p = \langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$. There is a unique path from the root to each vertex in a tree. If $u, v \in N$ and there is a path from u to v, then u is called an ancestor of v and v a descendant of u. If u is an immediate ancestor, then u is called the parent of v, and v the child of u. Each vertex v has just one parent while a vertex u can have one or more children. There are sibling vertices which share a same parent vertex. If for each parent vertex its children are uniquely identified, left to right, we call a tree is ordered. Otherwise, a tree is unordered.

According to the above definitions, trees have unique features that make them easy to represent other data types. However, those features also cause a tricky problem in retrieving information from trees. That is because of the hierarchy of trees, and the suggested solution of discovering information from them is a means of subtree inclusion [123]. The general tree inclusion problem given a pattern tree S and a target tree T is to find the subtrees of T that are instances of S. We give three types of subtree inclusion which are widely applied in the tree mining along with their characteristics.

The first one is an exact subtree; given a tree $T = (r, N, E, \mathcal{L})$, we say that an ordered tree $S = (r', N_S, E_S, \mathcal{L}')$ is included as an exact subtree of T, if and only if (1) $N_S \subseteq N$, (2) $E_S \subseteq E$, (3) for any vertex $v \in N$, if $v \in N_S$ then all descendants of v must be in N_S, (4) for all edges $(u, v) \in E_S$ the parent-child relation between vertices u and v is preserved in T identically with the one in S, (5) the label of any vertex $v \in N_S$, $\mathcal{L}'(v) = \mathcal{L}(v)$, and (6) the left to right ordering between the siblings in S must be preserved in T. Exact subtree problems have been researched comprehensively and are solvable in linear time, however, the definition is too restricted and there is no room for further improvement. A more relaxed definition is required.

The second one is an induced subtree; given a tree $T = (r, N, E, \mathcal{L})$, we say that an ordered tree $S = (r', N_S, E_S, \mathcal{L}')$ is included as an induced subtree of T, if and only if (1) $N_S \subseteq N$, (2) $E_S \subseteq E$, (3) for all edges $(u, v) \in E_S$ the parent-child relation between vertices u and v is preserved in T identically with the one in S, (4) the label of any vertex $v \in N_S$, $\mathcal{L}'(v) = \mathcal{L}(v)$, and (5) the left to right ordering among siblings in S should be a subordering of the corresponding vertices in T. An induced subtree S of T can be acquired by repeatedly pruning leaf vertices or the root if it has only one child in T. However, it still relies on the parent-child relation causing the restriction.

The third one is an embedded subtree; given a tree $T = (r, N, E, \mathcal{L})$, we say that an unordered or ordered tree $S = (r', N_S, E_S, \mathcal{L}')$ is included as an embedded subtree of T, if and only if (1) $N_S \subseteq N$, (2) for all edges $(u, v) \in E_S$, u is an ancestor of v in T, (3) the la-

bel of any vertex $v \in N_S$, $\mathcal{L}'(v) = \mathcal{L}(v)$, (4) if S is ordered, then for $v_1, v_2 \in N_S$ the order of v_1 must precede that of v_2 in S if and only if the order of v_1 precedes that of v_2 in T. The definition of embedded subtree extends those of exact and induced subtrees. The tree T must preserve ancestor-descendant relation but not necessarily parent-child relation for vertices in S.

3.2 Frequent Subtree vs. Maximal Frequent Subtree

Often occurred information means some data patterns frequently used by various users or applications. And, the primary goal of mining tree dataset is to provide such often occurred tree patterns. However, it is not straightforward as mentioned in the previous. We briefly describe the fundamental concepts of tree pattern occurrences.

We let $D = \{T_1, T_2, \ldots, T_i\}$ be a set of trees and $|D|$ be a number of trees in D, where $0 \le i \le |D|$. Given a tree S, the frequency of S with respect to D, $freq_D(S)$, is defined as $\sum_{Ti \in D} freq_{Ti}(S)$, 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 a fraction of trees in D that have S as a subtree, $Sup_D(S) = freq_D(S)/|D|$. 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 specified number is called minimum support and written σ. The problem of mining frequent subtrees is to uncover all pattern trees S which satisfies the condition $Sup_D(S) \ge \sigma$, σ-frequent subtrees. As stated in the previous, however, the combinatorial time for subtree generation becomes an inherent bottleneck of frequent subtrees mining and it causes finding all of them becomes harder.

The alternative way is required, and that is maximal frequent subtrees. It has to satisfy the following conditions when some minimum support σ is given: (1) $Sup_D(S) \ge \sigma$. (2) there exists no any other σ-frequent subtree S' in D such that S is a subtree of S'. A maximal frequent subtree is also frequent subtrees, however, none of its proper supertrees are frequent. Despite the fewer total numbers than that of frequent subtrees, maximal frequent subtrees do not lose frequent patterns since they subsume all of them.

4. SEAMSON ALGORITHM

4.1 Mining Goal

As we recall, the goal of this paper is to discover some special frequent subtrees, named maximal frequent subtrees, from a database of rooted labeled trees. The simple depiction of expected result from the suggested algorithm is on Fig. 1. We illustrate simply two user's access trees to Amazon web sites. In the trees, vertices correspond to web pages users visited and edges do to the links between the pages. Actually, each vertex label is taken by a URL, but we just use key words of it for the simplicity. As the result, the obtained maximal frequent subtree has five vertices. It turns out that this subtree is a part of a web site for HarryPotter store and a part of a web site for the books of Database. From this mining result, we can infer that both visitors have similar interests in browsing Amazon web sites. A service provider (in this example, we can say Amazon itself) can grasp the point of customer's web trace at a glance, due to the maximal frequent subtree.

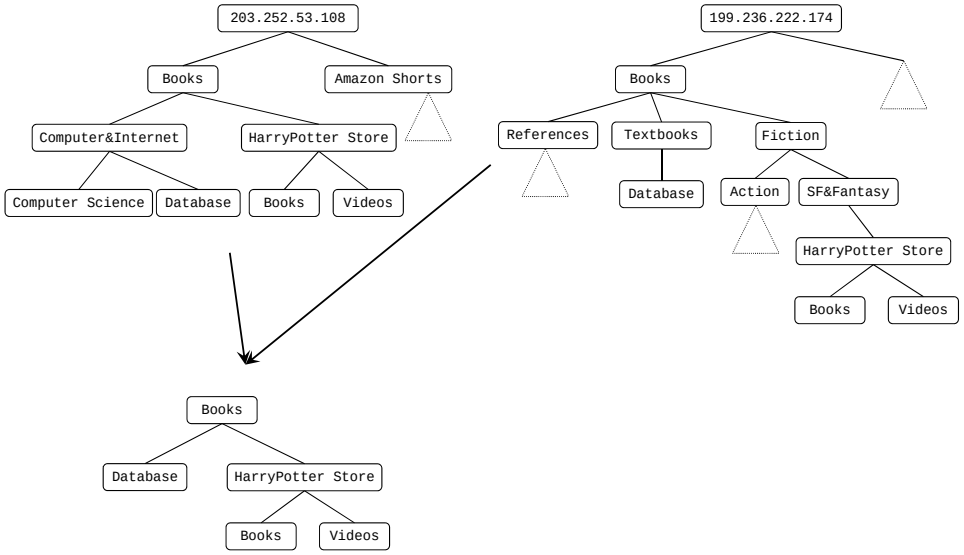


Fig. 1. Uncovered maximal frequent subtree from two user's access trees.

One of experiments presented by Chi *et al*. in [28] shows the benefits from using maximal frequent subtrees in mining of web access trees. Their experiments were run on the log files at UCLA Data Mining Laboratory (http://dml.cs.ucla.edu). From the log files 2,793 user access trees were generated that touched a total of 310 web pages, where vertices correspond to those pages and edges correspond to the links between them. As the vertex labels URLs were taken. For minimum support $\sigma = 1\%$, their algorithm mined *one maximal frequent subtree* having 18 vertices along with *16,507 frequent subtrees*. And it provides that the maximal frequent subtree is a part of a web site for the ESP^{2}Net project. From the mining result, one factor can be derived that many visitors to UCLA DM Lab are interested in details about the ESP^{2}Net project.

Handling efficiently such maximal frequent subtrees represents the core of this paper. For the ultimate goal we propose a fresh tree mining algorithm which adopts the approach of pattern-growth methods. SEAMSON standing for **S**calable and **E**fficient **A**lgorithm for **M**aximal frequent **S**ubtrees extracti**ON** which fundamental ideal had been published in [29, 30]. Its goal is same as that of the algorithm EXiT-B [27] previously published; however, its approach is totally different.

SEAMSON is developed to fix the problem of EXiT-B that is not guaranteeing that all frequent vertices have right relations with other frequent vertices when they build maximal frequent subtrees, to adopt a FP-tree based pattern-growth approach that makes the steps of subtree generations unnecessary, to extend the target dataset which is just for XML in EXiT-B but now it covers all tree structured data, and to extract maximal frequent subtrees without any candidate subtree generation that guarantees right relations between frequent vertices.

From the next subsection, we describe the detailed explanation of SEAMSON thoroughly and clearly, along with considering both the accuracy of the results and the efficiencies of performance and space aspects.

4.2 Label Projection

Trees are usually stored in a database D according to their related documents and each document is treated as a transaction. That is we say document-driven layout in this paper. In such layout, the whole trees are scanned every time whenever frequency is computed for each label and, thus, it requires $O(|D||T||L|)$ time complexity to get the frequencies of the whole labels, where $|D|$ is a total number of trees, $|T|$ is a maximum number of vertices of a tree, and $|L|$ is a number of distinct labels. It is not a serious problem when $|D|$ and $|T|$ are reasonably small. It may hinder the computation, however, if both values are large, and actually in the real world, two factors are large.

What if the database has been organized in a label-driven layout? During the scan of trees, all vertices with the same label are grouped together. The vertices from the same tree form a member of the group and the number of members actually determines the frequency of a given label; the maximum number of members is a number of trees in D. We name this new method as *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' frequencies 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 vertex indexes which are projected by ℓ construct a single linked list named as label-list. The label-list for a given label ℓ is denoted ℓ-list.

The head of label-list points to the first object in a body just like the ordinary head of linked list. Along with the indication, the head of a label-list gives information on which vertex indexes have been mapped to a projected label.

The body of a label-list immediately follows its corresponding head. The main concerns of the body are to evaluate how many trees have a projected label and to keep parents indexes of the vertices in a head. The structure of a body is a sequence of *members*. Each member is an object with one key field, one pointer field indicating to the next member, and one satellite data field. As a key a tree index is used. Because only the trees that assign a current projected label to their vertices are eligible to create members, the total number of members in a body indicates the number of trees using a projected label to their vertices. The satellite data field is for parent vertex indexes of the vertex in a head. Note that the parent index becomes 0 if a vertex index in a head has no parent. To mark it, we set 0 for such vertex's parent index.

Assume T_1, T_2, and T_3 in D are the trees whose at least one vertex is labeled by ℓ. The vertices having ℓ in each tree are: $n_a \in T_1$, n_b, $n_c \in T_2$, and n_d[1] $\in T_3$. First, tree indexes are placed in key fields and parent indexes of the vertices are stored in satellite data fields. The ℓ-list is $\langle(p_1, T_1, \rightarrow), (p_2\ p_3, T_2, \rightarrow), (p_4, T_3, \varepsilon)\rangle$, where p_i is a parent vertex index, $\rightarrow$ means a pointer to a next member, and ε means an empty pointer. The size of ℓ-list, $|\ell$-list|, is 3 with respect to the number of members.

During a database scan, members are generated and inserted into bodies of label-lists. The newly inserted member is added to the end of a body and the pointer field of its previous member points to this new member. The complete structure of a label-list is depicted on Fig. 2. Let assume a label ℓ be a current projecting label. As shown in the picture, m trees constitute the ℓ-list.

[1] n_a, n_b, n_c, n_d denote vertex indexes.

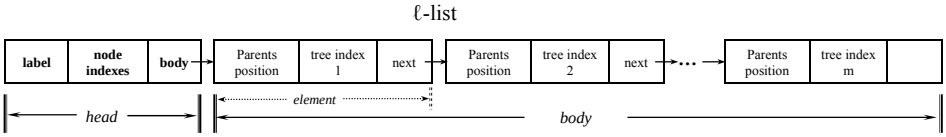


Fig. 2. General structure of a label-list.

Definition 2 [L-dictionary] According to a hashed value of a projected label ℓ, the generated ℓ-list is stored and arranged into an in-memory data structure. Whenever a label is given, its corresponding label-list is searched and retrieved from that structure to provide appropriate information. If a label has no matching label-list, it is newly projected label and thus its label-list is inserted into the structure. Since the structure works just like an ordinary dictionary, it is named *L*-dictionary and notated by $\mathbf{L}_{dic}$.

For a given tree, labels are projected by depth-first traversal order. During scan of a first tree, say T_1, only labels ℓ_k, such as $\mathcal{L}(v_i) = \ell_k \in L$ where $v_i \in N_{T_1}$ $(1 \le i \le |T_1|)$ and $1 \le k \le |L|$, establish their label-lists. Each ℓ_k-list constitutes fundamental units of a label projected database, $\mathbf{L}_{dic}$. After reading the first tree T_1, there are k label-lists whose sizes are all 1. This is because just one member for T_1 is contained in bodies of each label-list. While T_2 is being scanned, its labels are first searched through $\mathbf{L}_{dic}$ to find matching label-lists. If not, a label-list is newly made.

When any label ℓ_k is given to $\mathbf{L}_{dic}$, its hash value is computed and it is searched whether ℓ_k-list already reside in $\mathbf{L}_{dic}$. If so, a new member for T_2 is created and is appended to an end of existing members of ℓ_k-list's body. The size of ℓ_k-list is increased by 1. On the contrary, if null is returned, ℓ_k-list has to be formed and inserted to $\mathbf{L}_{dic}$. In this case, the label ℓ_k is used in none of vertices of T_1, that is $(\ell_k \in L) \wedge (\ell_k \cap L_{T_1} = \phi)$, where L_{T_1} is the labels used in T_1 $(L_{T_1} \subset L)$. The stated procedure is repeated until every tree is scanned and saved in $\mathbf{L}_{dic}$.

Lemma 1: Let $|\mathbf{L}_{dic}|$ be a number of label-lists inside $\mathbf{L}_{dic}$, size of $\mathbf{L}_{dic}$. The range of its size is always $0 < |\mathbf{L}_{dic}| \le |L|$.

***Proof*:** We prove informally by presenting the intuition. Let assume a set $L = \{\ell_1, \ell_2\}$. Since trees in *D* use only the labels in *L*, the label projection is performed with ℓ_1 and ℓ_2. Because there is no tree having no vertex label, it is impossible for both ℓ_1-list and ℓ_2-list to have zero member. At least one label should be assigned to vertices of the trees. If both labels are used in at least one tree, two label-lists are built. For the given *L*, $|\mathbf{L}_{dic}|$ is 1 or 2 always. ❑

Fig. 3 presents a label projected database $\mathbf{L}_{dic}$ constructed from tree database $D = \{T_1, T_2, T_3\}$. Each label $\ell \in L = \{A, B, C, D, E, F, G\}$ is projected to generate their label-lists. The number of members which can be added maximally in a body of a label-list is 3, because it depends on the total number of trees. Thus, the expected size of any label-list is between 1 and 3. Then, the label-lists are stored in $\mathbf{L}_{dic}$ according to the order of their hashed values, $\mathcal{H}(\ell)$; we assume the function $\mathcal{H}(\ell)$ takes a label as input and produces label's hash value as output.

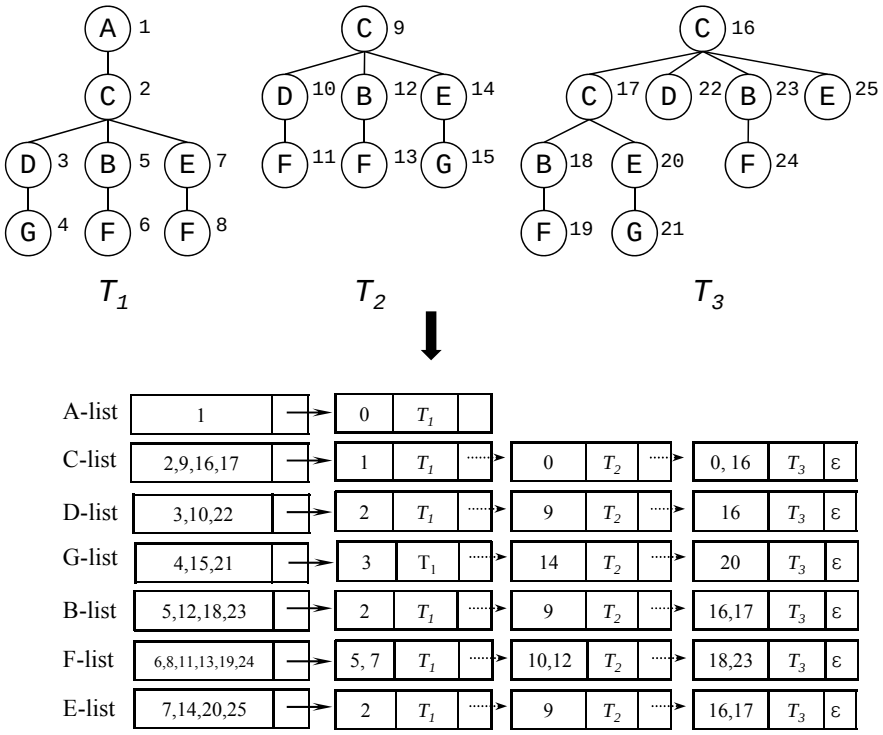


Fig. 3. $\mathbf{L}_{dic}$ from labeled trees.

4.3 Frequent 1-Subtrees and Candidate 2-Subtrees: F_1 and C_2

In traditional approaches for mining frequent subtrees, the first work is to discover frequent single vertex trees from the single vertex trees, because the entire set of frequent subtrees are generated by systematically growing the frequent 1-trees according to the apriori property [14, 31-33].

When D and S are given, all labels of S should be frequent in order S to be a frequent subtrees with respect to D. More technically speaking, if S is a k-tree and each one of k vertices are mapped by distinct k labels, the individual k 1-trees should be frequent subtrees of D in order S to be frequent. It means that k labels should be frequent labels. Through the label projection, every label is projected over D, produces its label-list, and builds $\mathbf{L}_{dic}$. At current status, label-lists in $\mathbf{L}_{dic}$ are analogous to 1-trees in D.

Definition 3 [frequent label] When σ is given with respect to D, a label ℓ is said to be a frequent label if ℓ-list is in $\mathbf{L}_{dic}$ and its $|\ell\text{-list}|$ is greater than or equal to $\sigma \times |D|$[2]. Otherwise, it is infrequent label.

Technically, being a frequent label means that a label ℓ is mapped to vertices in more than or equal to δ trees. Based on the apriori property those labels hinder a tree to grow[3] as a frequent subtree of D, because the property can be also applied to trees. Since growing trees starts from 1-trees, only frequent 1-trees contribute to the further progress.

[2] Threshold notated by δ.

[3] The meaning of 'grow a tree' is to build a supertree whose size is one more bigger than that of a current tree.

The label-lists, therefore, are pruned away from $\mathbf{L}_{dic}$ if and only if they have infrequent labels. Being pruned from $\mathbf{L}_{dic}$, those label-lists construct some special table, which replaces traditional steps for generating candidate subtrees. The detailed explanation for the table will be given later.

According to Definition 3, the label projected database $\mathbf{L}_{dic}$ is now tailored to the labels qualified for only frequent 1-subtrees in D. Since the current values of $\mathbf{L}_{dic}$ have been changed, we denote it simply $\mathbf{L}_{dic}^{\mathrm{f}}$, where f means *filtration*. Fig. 4 shows $\mathbf{L}_{dic}^{\mathrm{f}}$ produced from the $\mathbf{L}_{dic}$ on Fig. 2. We assume the given σ is ⅔. Only A-list is filtered out from $\mathbf{L}_{dic}$.

Definition 4 [frequent label-list] An label-list is said to be a frequent label-list iff it satisfies the following conditions: For a given label-list ℓ-list (1) $|\ell$ -list$| \geq \delta$. (2) For each member of it, when we let p be any parent index, $\mathcal{L}(p)$ is projected over D and its corresponding $\mathcal{L}(p)$-list exists in $\mathbf{L}_{dic}^{\mathrm{f}}$. (3) $|\mathcal{L}(p)$-list$| \geq \delta$, that is $\mathcal{L}(p)$-list $\in \mathbf{L}_{dic}^{\mathrm{f}}$.

The first condition indicates that a label-list must be made from the frequent label. The second and third ones focus on the body of a label-list, especially the parent indexes in members. The essentiality of these considerations stems from the structure of the label-list. Unlike the structures in conventional methods, a single label-list is actually embracing several trees implicitly which have two vertices, 2-trees. This is caused by the parent indexes in each member.

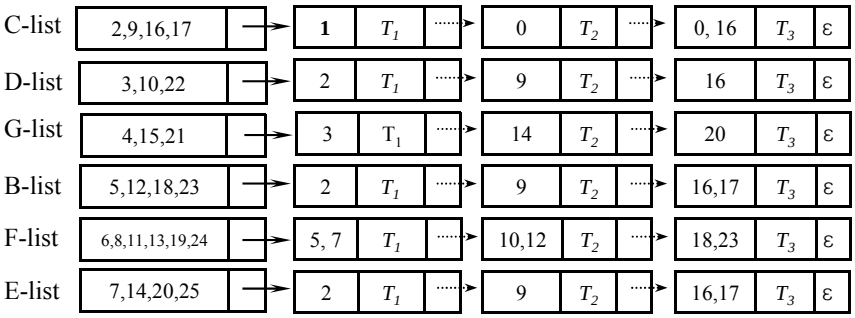


Fig. 4. Label-lists with frequent labels in $\mathbf{L}_{dic}^{\mathrm{f}}$.

If n vertex indexes exist in a head, the total number of parent indexes in a body is also n. This implies a current label-list holds the information of n 2-subtrees, and all leaf vertices of the subtrees are labeled by a projected label of the label-list. Here is the big difference; in most conventional approaches, every 1-tree has to be joined each other in order to acquire 2-subtrees. Then, frequent 2-subtrees are discriminated from the obtained trees by δ. However, in contrast to those methods, every 2-subtrees are directly acquired from the label-lists in $\mathbf{L}_{dic}^{\mathrm{f}}$. More conveniently, the 2-subtrees whose leaf vertices are mapped by infrequent labels are never generated.

However, it is not guaranteed that all parent indexes in $\mathbf{L}_{dic}^{\mathrm{f}}$ are mapped by frequent labels, because $\mathbf{L}_{dic}^{\mathrm{f}}$ is a result of removing the label-lists from $\mathbf{L}_{dic}$ not satisfying Definition 4. Referring Fig. 4 again, C-list is a good example to explain how Definition 4 works. The projected label C is frequent. For the conditions (2) and (3), the parent index

of the first member is given to a labeling function $\mathcal{L}$ as input, and then, $\mathcal{L}(1)$ computes the output A. A-list is in $\mathbf{L}_{dic}$, but the list is not in $\mathbf{L}_{dic}^{f}$. This means the 2-subtree in T_1 whose leaf vertex's label is C and its parent vertex's label is A is not frequent.

Starting from 2-subtrees the definition of embedded subtree has to be considered. It causes the evaluation of ancestor indexes' labels and their label-lists because embedded frequent 2-subtree is made by replacing an infrequent parent vertex index with its frequent ancestor's index. Instead of generating and evaluating candidate 2-subtrees, label-lists in $\mathbf{L}_{dic}^{f}$ can provide frequent 2-subtrees by simply making frequent label-lists.

4.4 Frequent 2-Subtrees: F_2

The label-lists in $\mathbf{L}_{dic}^{f}$ are divided into two types: one is for the label-lists which are already frequent label-lists and the other is for the label-lists which have only frequent labels. The former requires no further jobs; however, the latter does not. Some or all of parent indexes are expected to be replaced with their ancestors' indexes. Let assume a given label-list is ℓ-list, it consists of m members ($m \geq \delta$), and a parent index is p which is the only parent index in the first member. Note that p's ith ancestor is notated by p^i (p^0 is p itself).

Definition 5 [closest frequent ancestor] The index p has to be replaced by its frequent ancestor index if $\mathcal{L}(p)$-list $\notin \mathbf{L}_{dic}^{f}$. Starting from p^1 toward the root index (in the paper, 0 indicates the root), assume $r = p^d$. It is checked whether $\mathcal{L}(p^i)$ ($1 \leq i \leq d$) is frequent label or not by Definition 3. The above process is iterated until the ancestor p^i whose $\mathcal{L}(p^i)$ satisfies Definition 3 is firstly encountered. The index of p^i is stored in the first member instead of p. No more iteration is performed as soon as the ancestor is found. This peculiar ancestor vertex index is named by closest frequent ancestor according to its role and importance, and is denoted by Λ_p.

Fig. 5 clearly illustrates what the closest frequent ancestor is, how it is unveiled, and how an embedded frequent subtree is made. For the sake of convenience, we assume $D = \{T_1, T_2\}$ and $\sigma = 1$. $\mathbf{L}_{dic}^{f}$ has been constructed according to δ. In order to produce frequent label-lists, four label-lists in $\mathbf{L}_{dic}^{f}$ are checked if there is any parent index which does not have a frequent label. All label-lists, except F-list, are frequent label-lists. The index 12 in F-list is required by replacing Λ_{12}. Its label is easily obtained by $\mathcal{L}(12)$ = A, which list is pruned from $\mathbf{L}_{dic}^{f}$. The easiest way for finding Λ_{12} is to use A-list since it has the information of the parent index of 12 within its body. The index 11 is mapped to the label C and C has frequent C-list. The index 11 is the firstly met ancestor whose label is frequent, thus, Λ_{12} = 11 has been prevailed and the process is stopped. The index 12 is now replaced by 11 and F-list becomes the frequent label-list.

In the example, we should take a close look at the role of A-list. It was pruned from $\mathbf{L}_{dic}$, but it is used importantly to discover Λ_{12}. The pruned label-lists are mainly used to explore closest frequent ancestors because they are the only way to access ancestor indexes of the vertices whose labels are infrequent.

Time complexity to get Λ_p can bound $O(h)$ at worst case, where h is the height of a tree including p, because discovery of Λ_p requires actually iterative backtrack of a path, from p toward the root. The structure of a label-list, however, does not support the systematic procedures for iterative backtracking search since there is no mechanism to track

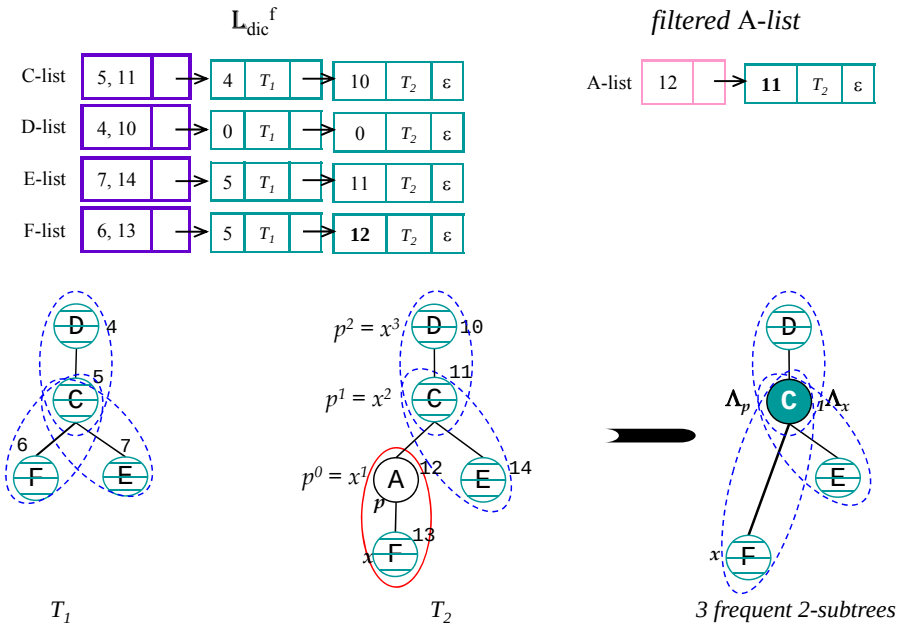


Fig. 5. Discovery of 3 frequent 2-subtrees: one is an embedded subtree uncovered by Λ_{12} and the others are exact subtrees.

several label-lists. Hence, the filtered label-lists construct a table. This table is a hash table table to provide constant-time $O(1)$ search on average regardless of the number of label-lists in the table, and is populated whenever the label-lists are pruned from $\mathbf{L}_{dic}$. The table is eliminated from the memory after every label-list in $\mathbf{L}_{dic}^{f}$ satisfies Definition 4. Since its role is analogous to that of candidate subtrees, we name it *candidate hash table* notated $\mathbf{T}_C$. The detailed explanation of how the table is constructed and Λ_p is discovered via the table is given in section 4.5.

Lemma 2: It is infeasible that an identical label-list is included in both $\mathbf{L}_{dic}^{f}$ and $\mathbf{T}_C$.

***Proof*:** Assume a label-list, say ℓ-list, exists in both $\mathbf{L}_{dic}^{f}$ and $\mathbf{T}_C$. This situation directly causes conflict. It is at least proved from that ℓ-list has frequent label if it is in $\mathbf{L}_{dic}^{f}$. Hence, it is never excluded from $\mathbf{L}_{dic}^{f}$, which means there is no chance for the list to be in $\mathbf{T}_C$. On the contrary, once ℓ-list is in $\mathbf{T}_C$, it indicates that label ℓ does not satisfy the given σ. Definitely, the label-list has to be pruned from $\mathbf{L}_{dic}$. Therefore, any label-list generated from a label in L is contained in either $\mathbf{L}_{dic}^{f}$ or $\mathbf{T}_C$. ❑

4.5 Candidate Hash Table

The primary operation of $\mathbf{T}_C$ is lookup. It works by transforming the key k using a hash function $\mathcal{H}_C$ to compute the index which indicates the desired location where the value should be. The index is computed by $\mathcal{H}_C(k)$. Under reasonable assumption, the expected time to search for a value in a hash table is $O(1)$.

According to the table, it is determined whether a given index has a frequent label or not. If a label of the index has been found in the table, the label is infrequent label. Afterwards, the process is followed to traverse the closest frequent ancestor of the index. Such mechanism is provided through three main entries of the hash table; key, value, and index which are easily obtainable from label-lists, and two functions; a labeling function $\mathcal{L}$ and a hash function $\mathcal{H}_C$.

As soon as a label-list is pruned from $\mathbf{L}_{dic}$, the three entries are decided from the lists at first to form a skeletal structure of $\mathbf{T}_C$. The label itself is eligible to become the key, but there is one obstacle that needs to be resolved before it is used as a key; labels are obtained by only vertex indexes with $\mathcal{L}$. What is given to $\mathbf{T}_C$ is the parent vertex index not parent vertex's label. If labels are defined as the keys, an additional work is required to get labels of vertex indexes through $\mathcal{L}$. This work only consumes $O(1)$.

After a key has been determined, it is straightforward to select values for them. It is an ancestor vertex index that gives the necessary information to perform backtracks, which is provided by the body part of label-lists. In order to traverse, each key is associated with the body of their label-lists by the hash function $\mathcal{H}_C$.

The table structure and preprocessing step are depicted on Fig. 6 along with its overall workflow. As stated in the previous subsection, label-lists in $\mathbf{L}_{dic}{}^{f}$ are examined and parent indexes in their bodies are switched by their closes frequent ancestors to satisfy all the conditions in Definition 4.

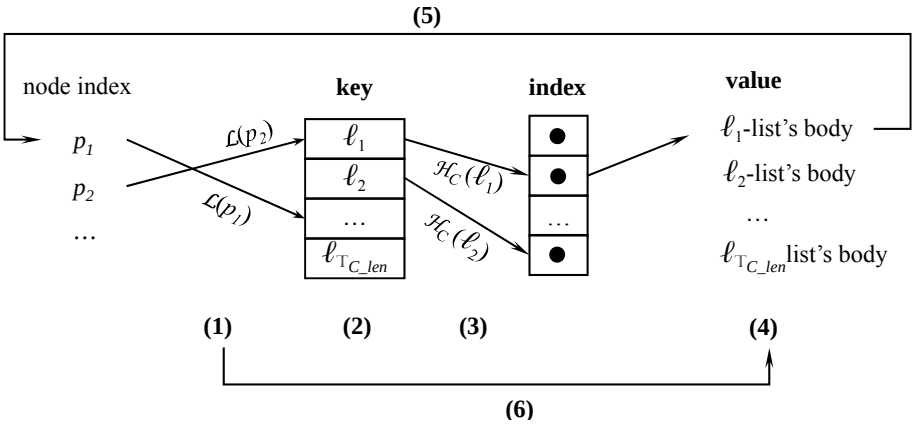


Fig. 6. Structure of $\mathbf{T}_C$ and its overall workflow.

Let ℓ_1-list $\in \mathbf{L}_{dic}{}^{f}$ be a currently tested label-list and $|\ell_1\text{-list}|$ be m. Each member of its ℓ_1-list.b[4] is required to undertake the following: Let any parent index in members be p. (1) Each p of a member is associated to its label by $\mathcal{L}(p)$. (2) The obtained $\mathcal{L}(p)$ is given to $\mathbf{T}_C$. If $\mathcal{L}(p)$ is not found among keys, p has a frequent label. Thus, p becomes a proper closest frequent ancestor, and the process is terminated. (3) In case of $\mathcal{L}(p)$ is infrequent, the index p is computed by $\mathcal{H}_C(\mathcal{L}(p))$. (4) According to a result, the value $\mathcal{L}(p)$-list.b is returned. (5) As backtracking, (1) to (4) is done to every p^1 in the value. (6) The p^1 whose label is found as the key of $\mathbf{T}_C$ iteratively performs (3) through (5) until Λ_{p^1} is found.

[4] From now on, ℓ-list.b means the body of a ℓ-list.

For all m members the steps (1) to (6) are performed. What if a proper Λ_p is not discovered until the end of backtracks? In such a case, none of p's ancestors including p itself have frequent labels. That means the vertex index whose label is ℓ_1 has no frequent ancestors. Hence, Λ_p is set by 0 to indicate it is the root position'.

Fig. 7 shows how $\mathbf{L}_{dic}^{\mathrm{f}}$ on Fig. 4 qualifies all of its label-lists to be frequent by using $\mathbf{T}_{\mathrm{C}}$. The parent index, p=1, in the first member of C-list.b is labeled by A, but, A-list was not in $\mathbf{L}_{dic}^{\mathrm{f}}$ because of |A-list| < 2. Instead, it is in $\mathbf{T}_{\mathrm{C}}$.

Under support of $\mathbf{T}_{\mathrm{C}}$, the index 1 traverses Λ_1: (1) The label of 1 is determined through $\mathcal{L}(1)$ = A. (2) The label A is given to $\mathbf{T}_{\mathrm{C}}$ to ascertain whether it is one of the keys of $\mathbf{T}_{\mathrm{C}}$. (3) Unfortunately, the returned value is A-list.b not `null`, which means the projected label A is infrequent. Thus, the index 1 should be switched by its Λ_1. (4) The returned value provides the method for backtracking, therefore, every vertex index inside it corresponds to parents of index 1 (grandparents of index 2). (5) The value has only one member, whose index 0 is given to $\mathbf{T}_{\mathrm{C}}$ in order to decide if its projected label is frequent or not. (6) $\mathcal{L}(0)$ cannot be found among keys of $\mathbf{T}_{\mathrm{C}}$ because it is always in $\mathbf{L}_{dic}^{\mathrm{f}}$ (the reason will be given below). Therefore, 0 is the closest frequent ancestor of 1. (7) The index 1 of the first member of C-list.b is changed by 0.

Because there is no tree not to have the root vertex, the index 0 is always frequent actually. However, it is not reflected in the $\mathbf{L}_{dic}^{\mathrm{f}}$ shown on Fig. 7. When we retransform frequent label-lists to obtain maximal frequent subtrees, 0 index and its corresponding label have to be required. Therefore, we artificially assign a special label $\top$ to 0 index, and insert its label-list, $\top$-list, into $\mathbf{L}_{dic}^{\mathrm{f}}$. Unlike other label-lists, the $\top$-list has no any other members, because its role is just to provide the root label. We call the $\top$-list as *dummy label-list*.

Referring Fig. 7 again, the first member of F-list.b has 5 and 7 indexes. Respectively their corresponding labels are B and E, which are different. On the other hand, the third member of F-list.b has 18 and 23 indexes and their corresponding labels are same as B. In the former it is appropriate for both parent indexes to be stored, because differently labeled parent indexes can have the children same labeled. However, the latter is different. Reminding the definition of frequency count, it does not matter how many times a same subtree is occurred in one tree. The frequency is always 1. The duplicated information is removed. The remained parent index is a *representative index* of the label.

After adding a dummy label-list into $\mathbf{L}_{dic}^{\mathrm{f}}$ and organizing each member with representatives, the finally gained $\mathbf{L}_{dic}^{\mathrm{f}}$ is shown on Fig. 8. The bold numbers are the representatives of members.

4.6 Label-List Extension

Now $\mathbf{L}_{dic}^{\mathrm{f}}$ assures that all of vertex indexes in it are labeled by frequent labels. Then, what about paths when the vertices form edges? 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 vertex labels by a and b, e = (a, b). If e wants to be a frequently occurred edge, both labels a and b must be frequent labels. Hence, if a path wants to be a frequently appeared path, all edges of the path should be frequently occurred. When let a path be $\mathbf{P}$, it is composed of a finite number of edges; $\mathbf{P} = e_1e_2\ldots e_m$. Also the path $\mathbf{P}$ can be expressed with labels as shown in the following equations:

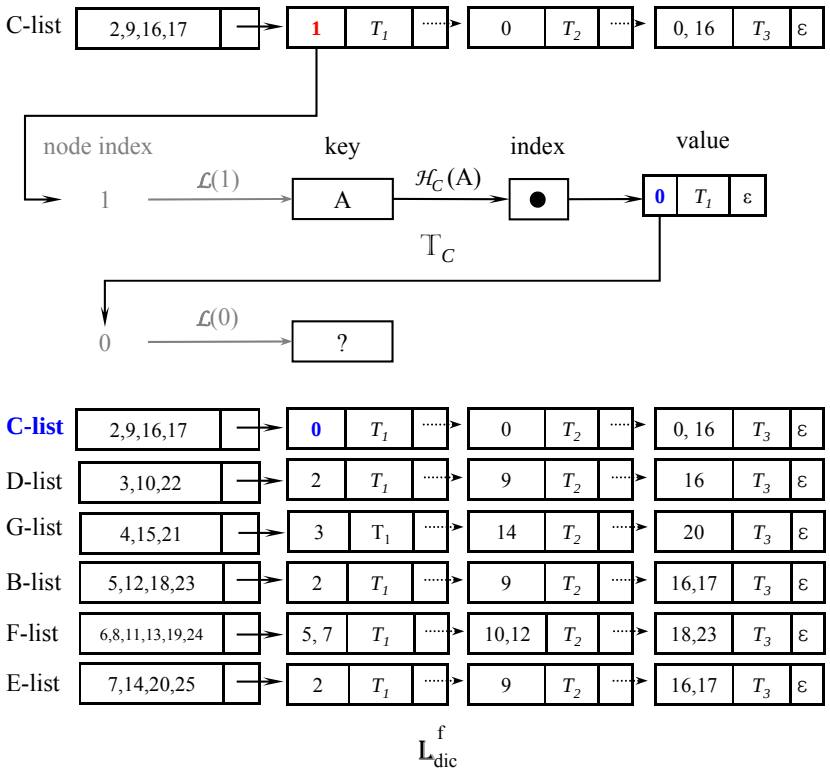


Fig. 7. Discovery and replacement of Λ_1.

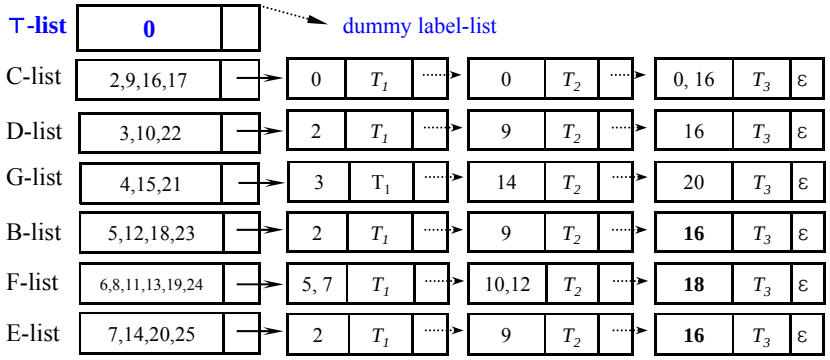


Fig. 8. $\mathbf{L}_{dic}^{\text{f}}$ with a dummy ⊤-list and representatives.

$$\begin{aligned}\mathbf{P} &= e_1e_2\ldots e_m = (v_1, v_2)(v_2, v_3)\ldots(v_m, v_{m+1})\\ &= (\mathcal{L}(v_1), \mathcal{L}(v_2))\ldots(\mathcal{L}(v_m), \mathcal{L}(v_{m+1}))\\ &= \mathcal{L}(v_1)\cdot\mathcal{L}(v_2)\ldots\mathcal{L}(v_m)\cdot\mathcal{L}(v_{m+1})\end{aligned} \tag{1}$$

According to Eq. (1), a path $\mathbb{P}$ is finally represented with a sequence of labels, which means all of the labels on the path should be frequent labels in order for $\mathbb{P}$ to be frequent. The bottom line is that the frequency of a path depends on each individual edge of it, and the frequency of an edge depends on its two composing vertices' labels.

In order to verify edge frequencies veiled inside of the label-lists in $\mathbf{L}_{dic}^{\mathrm{f}}$, simply parent indexes stored in body parts are used with the vertex indexes within the head parts. For revealing frequent edges, the required work before any effort is done is setting so called symbolic vertices based on the label-lists. What symbolic vertices are the vertices that become outlines of maximal frequent subtrees.

Let assume a current reading label-list is ℓ-list. A symbolic vertex s labeled by ℓ, s_ℓ, is firstly generated. Afterwards, its body is traversed to reach parent indexes, and according to their labels, their corresponding symbolic vertices are also created if not exist. The explicit edges between s_ℓ and parent symbolic vertices are made. Since any parent symbolic vertex's label is the label of a label-list in $\mathbf{L}_{dic}^{\mathrm{f}}$, forming edges actually relies on extending labels of the label-lists. Hidden paths are revealed along with the systematic relations between symbolic vertices.

Definition 6 [label-list extension] For ℓ-list, p is one of parent indexes in one of its members. A symbolic vertex s_ℓ which label is ℓ is set first, and then the second symbolic vertex whose label is $\mathcal{L}(p)$, $s_{\mathcal{L}(p)}$ is set. Those two symbolic vertices are joined together in order to uncover an edge. Since it seems that the vertex created from $\mathcal{L}(p)$-list is extending the range of its scope with the vertex from ℓ-list, the extension is named as label-list extension operation, abbreviated ℓ^2e. The operation ℓ^2e is denoted $s_{\mathcal{L}(p)} \rightarrow \ell$, which means $\mathcal{L}(p) \rightarrow \ell$, where '$\rightarrow$' indicates the direction of extending, parent to child. For every label-list in $\mathbf{L}_{dic}^{\mathrm{f}}$ the extension is committed.

The Eq. (1) can be rewritten with the label-list extension as the follows:

$$\begin{aligned}\mathbb{P} &= e_1e_2\ldots e_m = \mathcal{L}(v_1) \rightarrow \mathcal{L}(v_2) \rightarrow\ldots\rightarrow \mathcal{L}(v_m) \rightarrow \mathcal{L}(v_{m+1}) \\ &= s_{\mathcal{L}(v_1)} \rightarrow s_{\mathcal{L}(v_2)} \rightarrow\ldots\rightarrow s_{\mathcal{L}(v_m)} \rightarrow s_{\mathcal{L}(v_{m+1})}\end{aligned} \tag{2}$$

4.7 Potential Maximal Pattern Tree

While performing ℓ^2e operations, the required number of scanning $\mathbf{L}_{dic}^{\mathrm{f}}$ is two. During the first scan, only head parts of label-lists are scanned and each label is read to set their symbolic vertices. A created symbolic vertex has three fields; `prm` for pointing a primary parent vertex, `sub` for indicating a subordinate parent vertex, and `cnt` for counting edge's frequency, respectively. The detailed role and functionality of the fields will be described later in respect of the second scan.

The initially made symbolic vertices are identified as *seeds* because a potential maximal frequent tree is brought forth base on and growing them. Along with the settlement of seeds, a table is constructed with seeds and labels of the label-lists in $\mathbf{L}_{dic}^{\mathrm{f}}$. It facilitates the process of extracting and traversing the tree being made. The key of the table has labels from the label-lists as its values. Another column is for pointers which indicate the locations of seeds within the tree; we name these pointers as seed-links. Whenever a label is given to the table, a position of its seed is retrieved via its proper seed-link. Since labels in the table head for the locations of their seeds in the tree and

present them, we name the table as *label header table*, denoted $\mathbf{T}_{\mathrm{L}}$.

The storage size for saving $\mathbf{T}_{\mathrm{L}}$ is appropriate; assuming a size of a single table record x and $|\mathbf{L}_{dic}{}^{\mathrm{f}}| = \mathrm{M}$, the total necessary space is fixed at xM because the length of a table depends on the number of labels in $\mathbf{L}_{dic}{}^{\mathrm{f}}$. In the worst case, $\mathbf{T}_{\mathrm{L}}$ has a storage complexity of $\Theta(\mathrm{x}|L|)$ time if labels of a label set L are all frequent. However, it is unusual that every label of L is frequent. This means the actually necessary space to store $\mathbf{T}_{\mathrm{L}}$ is much smaller than that of the worst case, because of $\mathrm{M} \ll |L|$. It is mainly used at the second scan along with the operation $\ell^2 e$.

After completing the first scanning process, total seven seeds are generated for $\mathbf{L}_{dic}{}^{\mathrm{f}}$ on Fig. 8 and as well as a table $\mathbf{T}_{\mathrm{L}}$ has been created with seven rows. Each row contains a label of a seed and a seed-link to the seed. Currently, none of seeds have specified values in their three fields because the values of fields are set during $\ell^2 e$.

$\mathbf{L}_{dic}{}^{\mathrm{f}}$ is scanned for the second time to run $\ell^2 e$. Body parts are analyzed. When a current reading label-list is ℓ-list, $\ell^2 e$ is started by parent indexes in each member through the following: (1) Let a seed of the list be s_ℓ which is pointed by the label ℓ. (2) The body of ℓ-list is read. Assume a current member is m and it has total i parent indexes, where each parent index is notated by p_{m_i}. (3) A label of p_{m_i} is obtained by $\mathcal{L}(p_{m_i})$, and guarantees that the row of $\mathcal{L}(p_{m_i})$'s seed, $s_{\mathcal{L}(p_{m_i})}$, appears in $\mathbf{T}_{\mathrm{L}}$. (4) Look up $\mathbf{T}_{\mathrm{L}}$ to obtain the location of $s_{\mathcal{L}(p_{m_i})}$. (5) The seed $s_{\mathcal{L}(p_{m_i})}$ is a firstly determined symbolic parent vertex of s_ℓ, if s_ℓ.`prm` is null. The field `prm` is for a first symbolic vertex set as a given seed's parent. The returned value, address of $s_{\mathcal{L}(p_{m_i})}$, is stored in s_ℓ.`prm`. According to the address, $\mathcal{L}(p_{m_i}) \rightarrow \ell$ is run and $s_{\mathcal{L}(p_{m_i})}$.`cnt` is incremented by 1. Instead of incrementing a child seed's `cnt`, we increment a parent seed's `cnt`. Because $\ell^2 e$ is processed in a bottom-up way, a parent seed is the end point of an edge.

What if s_ℓ.`prm` is already preoccupied by other seed's location, say an address of $s_{\mathcal{L}(q)}$? There are three cases depending on whether $s_{\mathcal{L}(p_{m_i})} = s_{\mathcal{L}(q)}$ or not. According to the case, the step (5) is replaced with one of the followings:

(5-1) If $s_{\mathcal{L}(p_{m_i})} \neq s_{\mathcal{L}(q)}$ and s_ℓ's `sub` = `null`, s_ℓ has more than one parent whose labels are different as $\mathcal{L}(p_{m_i})$ and $\mathcal{L}(q)$. To avoid graph structure, a concept of subordinate parent has been proposed; If a current seed's `prm` is already set as the address of $s_{\mathcal{L}(q)}$, a new symbolic vertex is made, its address is set in s_ℓ.`sub`, and its `prm` value is the address of $s_{\mathcal{L}(p_{m_i})}$. By this way, there are one primary parent at top and several subordinate parents following it. We denote it $s^j{}_\ell$ where j is the order of sub-parents. For instance, $s^1{}_\ell$ is the first sub-parent of s_ℓ, where it is pointed by s_ℓ.`sub` and $s^1{}_\ell$.`cnt` is incremented.

(5-2) Another case is that $s_{\mathcal{L}(p_{m_i})} = s_{\mathcal{L}(q)}$, where the `cnt` value of $s_{\mathcal{L}(q)}$ is increased by 1.

(5-3) The third case is that $(s_{\mathcal{L}(p_{m_i})} \neq s_{\mathcal{L}(q)}) \wedge (s_\ell$.`sub` $\neq$ `null`$)$. In this case, $s_{\mathcal{L}(p_{m_i})}$ has to be compared with `prm` fields of s_ℓ's sub-parent vertices if s_ℓ's `sub` is not null. If the value of any sub-parent's `prm` is same as the address of $s_{\mathcal{L}(p_{m_i})}$, `cnt` of the corresponding symbolic vertex is incremented by 1.

(6) Repeat from step (2) for all parent indexes p_{m_i}, and iterate all steps for all label-lists in $\mathbf{L}_{dic}{}^{\mathrm{f}}$.

After completing the whole process, a single tree which root is the seed of $\top$-list is derived. Because the tree includes maximal frequent subtrees, we name it potential maximal symbolic tree abbreviated PMST.

Fig. 9 illustrates $\mathbf{T}_L$ and PMST. Each seed is associated with its corresponding label in $\mathbf{T}_L$ via seed-links (marked as seed @), shown as dotted lines with small arrowheads. Edges made by operating $\ell^2 e$ and connecting seeds or symbolic vertices are depicted by solid lines in the figure. Solid lines are divided into two types according to their usefulness; blurred lines and bold lines. The reason is the frequency of edges was not considered in PMST. To take for it, `cnt` fields are read. If s.`cnt` is less than a given threshold, s and its entering edge are not frequent. The blurred lines are those edges pruned from PMST. On the contrary, the bold ones are frequent edges remained in PMST.

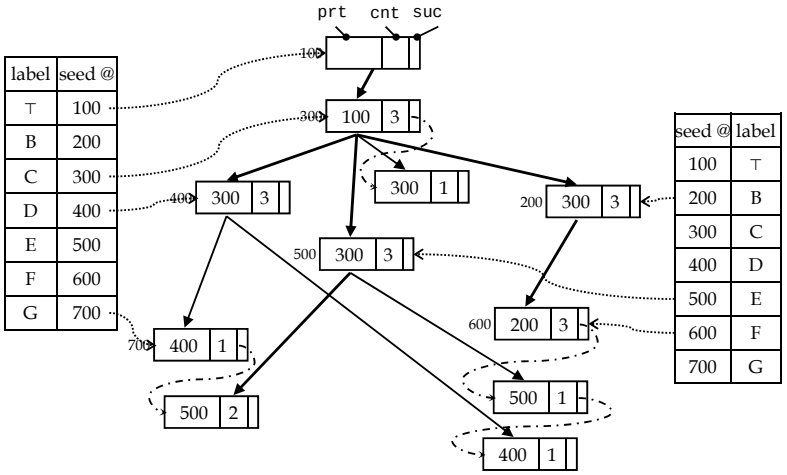


Fig. 9. Derived PMST from $\mathbf{L}_{dic}^{\mathrm{f}}$, it holds potential maximal frequent subtrees.

In the figure, some seeds associated with the labels C, F, G are chained by several sub-parents. Those are drawn with dash-dotted curves.

Before deriving a PMST tree, we can infer the number of maximal frequent subtrees from label-lists in a final $\mathbf{L}_{dic}^{\mathrm{f}}$ according to the following equation:

$$\#\text{ of maximal frequent subtrees} = \#\text{ of label-lists in which a count of members having the index 0 is more than or equal to } \delta \tag{3}$$

Let ℓ_1-list, ℓ_2-list, ℓ_3-list be arbitrary frequent label-lists in $\mathbf{L}_{dic}^{\mathrm{f}}$ and their sizes are $|\ell_1\text{-list}| = |\ell_2\text{-list}| = 2$, $|\ell_3\text{-list}| = 4$, when a given δ is 2. We assume each member of them has only one parent index, for the sake of simplicity. The members of ℓ_1-list and ℓ_2-list have a parent vertex index 0 which means the root; ℓ_1-list is denoted $\langle(0, T_1, \rightarrow),(0, T_2, \varepsilon)\rangle$ and ℓ_2-list $\langle(0, T_1, \rightarrow),(0, T_2, \varepsilon)\rangle$, *which implies there maybe two maximal frequent subtrees*. Let the parent indexes in members of ℓ_3-list be p_1, p_2, p_3, and p_4. Then, we can consider the following three cases:

- **Case 1:** $\mathcal{L}(p_1) = \mathcal{L}(p_2) = \ell_1$ and $\mathcal{L}(p_3) = \mathcal{L}(p_4) = \ell_2$: Two vertices labeled by ℓ_1 and ℓ_2 are direct children of the root, because both edge frequencies satisfy 2. The vertex labeled by ℓ_3 becomes a sub-parent, because both edge frequencies of different parents also

meet 2. Since ℓ_3-list has no members satisfying the value of the equation, just *two* maximal frequent subtrees can be derived, one is (ℓ_1, {ℓ_1, ℓ_3}, {(ℓ_1, ℓ_3)}, $\mathcal{L}$)[5] and the other (ℓ_2, {ℓ_2, ℓ_3}, {(ℓ_2, ℓ_3)}, $\mathcal{L}$).

- **Case 2:** $\mathcal{L}(p_1) = \mathcal{L}(p_2) = \mathcal{L}(p_3) = \ell_1$ and $\mathcal{L}(p_4) = \ell_2$: The edge ($\ell_1$, ℓ_3) satisfies the threshold as 3, but the edge (ℓ_2, ℓ_3) is not. Like case 1, ℓ_3-list has no members satisfying the value of the equation. Therefore, the number of maximal frequent subtrees are still *two*, (ℓ_1, {ℓ_1, ℓ_3}, {(ℓ_1, ℓ_3)}, $\mathcal{L}$) and (ℓ_2, {ℓ_2}, {ϕ}, $\mathcal{L}$).
- **Case 3:** $\mathcal{L}(p_1) = \mathcal{L}(p_2) = 0$ and $\mathcal{L}(p_3) = \mathcal{L}(p_4) = \ell_1$ or ℓ_2: ℓ_3-list has two members in which they have the index 0 and it satisfies the condition. The index 0 has total three labels, ℓ_1, ℓ_2, and ℓ_3. By $\mathcal{L}(p_3) = \mathcal{L}(p_4) = \ell_1$ or ℓ_2, ℓ_1 or ℓ_2 is connected by ℓ_3. There can be total *three* maximal frequent subtrees, ($\ell_1||\ell_2$, {$\ell_1||\ell_2$}, {ϕ}, $\mathcal{L}$), ($\ell_1||\ell_2$, {$\ell_1||\ell_2$, ℓ_3}, {($\ell_1||\ell_2$, ℓ_3)}, $\mathcal{L}$) and (ℓ_3, {ℓ_3}, {ϕ}, $\mathcal{L}$).

Theorem 1 (Correctness) Label-list extension returns all the possible maximal frequent subtrees in the given tree database.

***Proof*:** According to Definitions 4, 5 and Lemma 2, all the vertices marked in $\mathbf{L}_{dic}^{f}$ are guaranteed to be assigned by only frequent labels. The label-list extension procedure produces every possible frequent path, and finally considers only maximal frequent paths by extending each path with $(L - \mathbf{T}_C) \cup \{\top\}$ of the current label's $\mathbf{L}_{dic}^{f}$. During $\ell^2 e$, an edge e_i is uncovered by $s_{\mathcal{L}(v_i)} \rightarrow s_{\mathcal{L}(v_{i+1})}$ $(1 \le i \le m)$. If $\mathcal{L}(v_i)$ is $\top$, then, the child symbolic vertex $s_{\mathcal{L}(v_{i+1})}$ is the real root of some tree, because $s_{\mathcal{L}(v_{i+1})}$ is one of the highest ancestors' symbolic vertices which all descendants have frequent labels according to Definition 5. The $s_{\mathcal{L}(v_{i+1})}$ rooted tree, $Ts_{\mathcal{L}(v_{i+1})}$, have its all possible frequent paths including maximal frequent paths. Let assume any discovered path of $Ts_{\mathcal{L}(v_{i+1})}$, $\mathbb{P} = s_{\mathcal{L}(v_{i+1})} \rightarrow s_{\mathcal{L}(v_{i+2})} \rightarrow \ldots \rightarrow s_{\mathcal{L}(v_{i+m-1})} \rightarrow s_{\mathcal{L}(v_m)}$. According to the values of symbolic vertices' `cnt`s, there are two cases:

If $\forall j \mid s_{\mathcal{L}(v_j)}.\texttt{cnt} \ge \delta\,(i{+}1 \le j \le i{+}m{-}1)$, then the whole path $\mathbb{P}$ is a maximal frequent path else $\exists j \mid s_{\mathcal{L}(v_j)}.\texttt{cnt} < \delta$, then only subpath $\mathbb{P}' = s_{\mathcal{L}(v_{i+1})} \rightarrow s_{\mathcal{L}(v_{i+2})} \rightarrow \ldots \rightarrow s_{\mathcal{L}(v_{j-1})}$ becomes a maximal frequent path. A subpath $\mathbb{P}'' = s_{\mathcal{L}(vj)} \rightarrow s_{\mathcal{L}(v_{j+1})} \rightarrow \ldots \rightarrow s_{\mathcal{L}(v_{i+m-1})} \rightarrow s_{\mathcal{L}(v_m)}$ is a dangling frequent path. Later such dangling paths are pruned paths in PMST. In the end, because the vertex $s_{\mathcal{L}(v_{i+1})}$ is extended to its all maximal frequent paths, the tree $Ts_{\mathcal{L}(v_{i+1})}$ carries a possible maximal frequent subtree. Therefore, the number of possible maximal frequent subtrees equal to the value of $s_{\top}.\texttt{cnt}$.

4.8 Algorithm Analysis

The main idea of SEAMSON includes the following three points: $\mathbf{L}_{dic}$ from label projection, $\mathbf{L}_{dic}^{f}$ with $\mathbf{T}_C$, and derivation of PMST. Fig. 10 shows the pseudo-code of SEAMSON. There is only a single database scan. The label projected database, $\mathbf{L}_{dic}$, is constructed by the first key method Transform-Trees-to-Lists. Its first parameter r_T is the root of a tree T, and the second parameter $\mathbf{L}_{dic}$ is for storing a label projected database. The next work is for identifying frequent labels among projected labels. As stated in lines (4)-(7), each label-list in $\mathbf{L}_{dic}$ is checked whether its number of members is less than the given δ. If so, the corresponding label-list is put into the table $\mathbf{T}_C$ and deleted from $\mathbf{L}_{dic}$. The running time is $O(|L|)$ because it depends on the number of label-lists.

[5] $T = (r, N, E, \mathcal{L})$.

```
Algorithm SEAMSON
input: D, ℒ, σ                                  output: MaxForest
(1)δ ← D × σ
(2)for each tree T ∈ D
(3)   Transform-Trees-to-Lists(r_T, L_dic)
(4)for each label-list ℓ-list ∈ L_dic
(5)   if |ℓ-list| < δ
(6)      T_C ← T_C + ℓ-list
(7)      L_dic^f ← L_dic – ℓ-list
(8)L_dic^f ← Compute-Frequent-Lists(L_dic^f, T_C)
(9)for each ℓ-list ∈ L_dic^f
(10)  for i = 0 to |ℓ-list|-1
(11)     for each parent index p ∈ ℓ-list[i].pids set a representative
(12)L_dic^f ← L_dic^f + ⊤-list
(13)for each ℓ-list ∈ L_dic^f
(14)   insert ℓ into T_L.label
(15)   create a seed s_ℓ for PMST
(16)   write the address of s_ℓ to T_L[ℓ].seed@
(17)for each ℓ-list ∈ L_dic^f
(18)   Derive-Temp-MaxTree(ℓ-list, T_L, PMST)
(19)for each node s ∈ PMST (except the root s_r)
(20)   if s.cnt < δ then PMST ← PMST – s
(21)MaxForest ← PMST - s_r
```

Fig. 10. Pseudo-code of SEAMON algorithm.

By through the method Compute-Frequent-Lists, $\mathbf{L}_{dic}$ contains only frequent label-lists. For each label-list a few empirical works, inserting a dummy label-list and setting representatives, are done as stated from line (9) to (12). In the code, ℓ-list[i] means the (i+1)th member of ℓ-list and ℓ-list[i].pid means parent indexes of ℓ-list's (i+1)th member. At line (12), the dummy label-list is inserted manually.

Before invoking the third method, the table $\mathbf{T}_L$ and an initial PMST are constructed for deriving full PMST. This work requires simple insertion and creating operations, thus, the running time is a constant time. From (9) to (16) the required running time is $O(|L|(1 + |D|) = O(|L| + |L||D|)$. By invoking Derive-Temp-MaxTree at line (18), the final goal of SEAMSON is established. We now describe the each key method in more detail.

Transform-Trees-to-Lists Its pseudo-code is listed on Fig. 11. When the method is invoked, the first work is to search a label of a passed vertex v whether it exists in $\mathbf{L}_{dic}$ or not. If so, the vertex index v is inserted to the label-list's head. According to existence or nonexistence of v's tree index in members of the label-list's body, its parent indexes are inserted to an existing member or a new member, respectively. The lines (1) to (6) shows the code of it, where $\mathcal{L}(v)$-list[i].tid means a tree index of the (i+1)th member of $\mathcal{L}(v)$-list and $\mathtt{tid}_v$ is the tree index having the vertex index v.

Lines (7) and (8) are for the case when v's label is not found in labels of $\mathbf{L}_{dic}$. As shown in the code, simply a new label-list for $\mathcal{L}(v)$ is generated and added to $\mathbf{L}_{dic}$. Lines (9)-(10) check whether a v has children and, when any child is found, the method is recursively applied to each child vertex. Considering the running time of this method, for searching work at line (1) it is proportional to a length of $\mathbf{L}_{dic}$. Fortunately, the worst-case

```
Method 1 Transform-Trees-to-Lists
input: vertex v, L_dic                                  output: L_dic
(1)if ℒ(v) ∉ L_dic.label
(2)   L_dic ← +ℒ(v)-list
(3)else
(4)   ℒ(v)-list.h ← +v
(5)   for i = 0 to |ℒ(v)-list|-1
(6)      if ℒ(v)-list.tid == tid_v then ℒ(v)-list[i].pid ← v^1
(7)   if there is no member having the same tree id with tid_v
(8)      ℒ(v)-list.b ← + new member for tid_v
(9)while v's children are all being scanned
(10)  Transform-Trees-to-Lists(child u, L_dic)
```

Fig. 11. Pseudo-code of transformation for trees in order to convert labels-lists.

is $O(|L|)$, because it depends on the number of labels L. The time for insertion is $O(1)$. The loop on lines (5)-(6) takes time $O(|D|)$ because a size of a corresponding label-list's body can be a total number of trees in D at the worst-case. Following the depth-first manner, the running time of Transform-Trees-to-Lists is $O(|N|(|L|+|D|))$.

Compute-Frequent-Lists As shown in the codes stated in Fig. 12, the main work is to search closest frequent ancestors for the parent vertex indexes having infrequent labels. Due to the recursive function closest-frequent-ancestors, every parent index finds its closest ancestor having frequent label. In the code, $\mathbf{T}_C$.key is the keys of the table $\mathbf{T}_C$, and $\mathbf{T}_C[\mathcal{L}(p)]$.val implies that the returned value from $\mathbf{T}_C$, actually $\mathcal{L}(p)$-list.b.

```
Method 2 Compute-Frequent-Lists
input: old L_dic^f, T_C                                 output: new L_dic^f
(1)for each label-list ℓ-list ∈ old L_dic^f
(2)   for i = 0 to |ℓ-list|-1
(3)     if ∃p|(p ∈ ℓ-list[i].pid ∧ ℒ(p) ∈ T_C.key)
(4)       cfa ← closest-frequent-ancestor(ℓ-list[i].tid, p, T_C[ℒ(p)].val)
(5)     else
(6)       cfa ← p
(7)     p ← cfa
closest-frequent-ancestor(t, p, v)
(8)for i = 0 to |v.b|-1
(9)   if v[i].tid == t
(10)    if ∃a|(a ∈ v[i].pid ∧ ℒ(a) ∈ T_C.key)
(11)      cfa ← closest-frequent-ancestor(v[i].tid, a, T_C[ℒ(a)].val)
(12)    else
(13)      cfa ← a
(14)    return cfa
(15)return cfa
```

Fig. 12. Pseudo-code of computing frequent label-lists for enhanced $\mathbf{L}_{dic}^{f}$.

The worst time to process Compute-Frequent-Lists method is $O(|L||D|I_{cfa})$, where I_{cfa} indicates the cost of time to compute closes frequent ancestors. The function closest-frequent-ancestor itself requires $O(1)$ time to replace an infrequent vertex with a frequent

vertex. Because it contains a recursive call to itself, the operations actually run in time at most proportional to the height of an original tree and thus take $O(\lg|N|)$ time, therefore, I_{cfa} is replaced by $O(\lg|N|)$. The cost of time to run the method Compute-Frequent-Lists is $O(|L||D|\lg|N|)$.

Derive-Temp-MaxTree When any label-list, ℓ-list, is read as the first action of Derive-Temp-MaxTree, each parent index in its body checks and determines its seed's location based on a comparison with s_ℓ.prm or s_ℓ.sub to make a parent-child relation with s_ℓ for PMST. Also, the occurrence count is accumulated in a proper seed or just symbolic vertex. The work for determination is listed from lines (3) to (10) on Fig. 13.

The required running time depends on only the number of members of a label-list for the first for loop; at the worst case, some label-list can have the number of members as many as a total number of trees. The time complexity of Derive-Temp-MaxTree method, therefore, is $O(|D|)$.

```
Method 3 Derive-Temp-MaxTree
input: a single ℓ-list, T_L, initial PMST       output: full PMST
(1)for i = 0 to |ℓ-list|-1
(2)   for each p ∈ ℓ-list[i].pid
(3)     if s_ℓ.prm == null              ▷ s_ℓ is known by T_L[ℓ].seed@
(4)       s_ℓ.prm ← T_L[L(p)].seed@
(5)       s_L(p).cnt ← +1
(6)     else if (s_ℓ.prm == T_L[ℓ].seed@) or (∃j|s_ℓ^j.sub == T_L[L(p)].seed@)
(7)       s_L(p).cnt ← +1 or s_ℓ^j.cnt ← +1
(8)     else
(9)       create s_ℓ^(j+1)
(10)      s_ℓ^(j+1).cnt ← +1
```

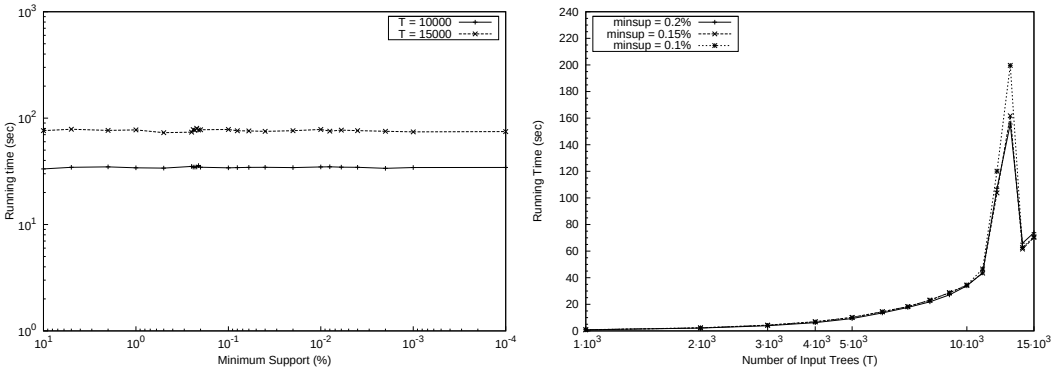
Fig. 13. Pseudo-code of computing frequent label-lists for enhanced $\mathbf{L}_{dic}^{f}$.

5. EXPERIMENTAL EVALUATION

This section provides extensive experiments to evaluate the performance of the algorithm SEAMSON using both synthetic datasets and a real application dataset; the former is generated by a tree generation program inspired by Zaki [21] and Termier [34]. It constructs a set of trees D based on some parameters; $\mathcal{T}$ the number of trees in D, L: the set of labels, f: the maximum branching factor of a vertex, d: the maximum depth of a tree, ρ: the random probability of one vertex in the tree to generate children or not, η: the average number of vertices in each tree in D. We allow multiple vertices in a tree to have the same label. We used the following default values for the parameters; $\mathcal{T}$ = 10,000, L = 100, f = 5, d = 5.

The latter is CSLOGS[6], which is also described in [21]. This real world dataset consists of the web access trees collected over one month at the CS department of the Rensselaer Polytechnic Institute. CSLOGS contains 59,691 trees corresponding to user browsing subtrees of the CS department website, and 13,209 unique vertex labels corresponding to the URLs of the web pages.

[6] http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software



(a) Minimum support vs. Time. (b) Number of trees vs. Time.

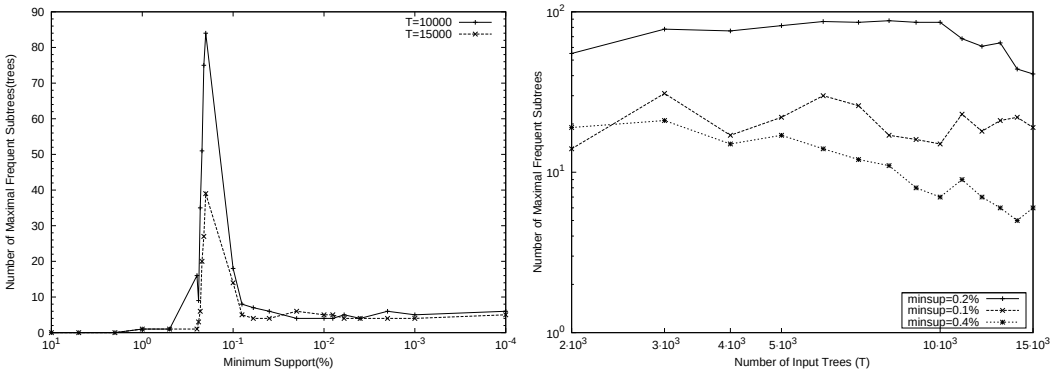
Fig. 14. Scalability of the algorithm.

5.1 Performance Evaluation

In the first experiment, we want to evaluate the scalability of our algorithm with varying minimum support and the number of trees $\mathcal{T}$, while other parameters are fixed as: $L = 100$, $f = 5$, $d = 5$, $\rho = 20\%$, $\eta = 13.8$ and 20.5 (when $\mathcal{T} = 10{,}000$ and 15,000, respectively). Fig. 14 (a) shows the result, where the minimum support is set from 10% to 0.0001%. Both X- and Y-axis are drawn on a logarithmic scale for the convenience of observation. We can find that the running time increases when the number of trees $\mathcal{T}$ increases, however, both running times are rarely affected by the decrease of the minimum support. This is because SEAMSON relies on the number of labels not the number of vertices. Thus, it is very efficient for datasets with varying and growing tree sizes.

In Fig. 14 (b) the parameter $\mathcal{T}$ varies from 1,000 to 15,000 with $\eta = 20$. We evaluated three different minimum supports, 0.2%, 0.15%, and 0.1%. The corresponding graphs reveal considerable similarity which slowly increases until $\mathcal{T} = 11{,}000$ and suddenly goes up between $\mathcal{T} = 11{,}000$ and $\mathcal{T} = 13{,}000$. Afterwards, the graphs are started to rapidly deteriorate. Our understanding of this phenomenon is that the size of $\mathbf{L}_{dic}^{\mathrm{f}}$ and its label-lists are maximized when the number of input trees reaches at 12,000 and 13,000 under 100 distinct vertex labels.

Fig. 15 presents the number of maximal frequent subtrees under different minimum supports and the number of input trees. Firstly on Fig. 15 (a) it seems that in both $\mathcal{T} = 10{,}000$ and $\mathcal{T} = 15{,}000$ the number of maximal frequent subtrees slowly grows before the rapid rise. Between $\sigma = 0.3\%$ and 0.2%, the two datasets generated with $L = 100$ produce the biggest number of maximal frequent subtrees. Afterwards, the number drops off and keeps in steady state. This feature stems from the limited number of labels and its random distribution for datasets generation. As another evaluation of the number of maximal frequent subtrees, we gradually increased the parameter $\mathcal{T}$ from 2,000 to 15,000 under three different minimum supports as presented on Fig. 15 (b). The number of maximal frequent subtrees slowly decreases while $\mathcal{T}$ increases under any minimum support. Because of the limited number of labels $L = 100$ and their random assignments to vertices, the number of maximal frequent subtrees is getting less instead the size of those trees are getting bigger.



(a) Maximal frequent subtrees vs. σ. (b) Maximal frequent subtrees vs. $\mathcal{T}$.

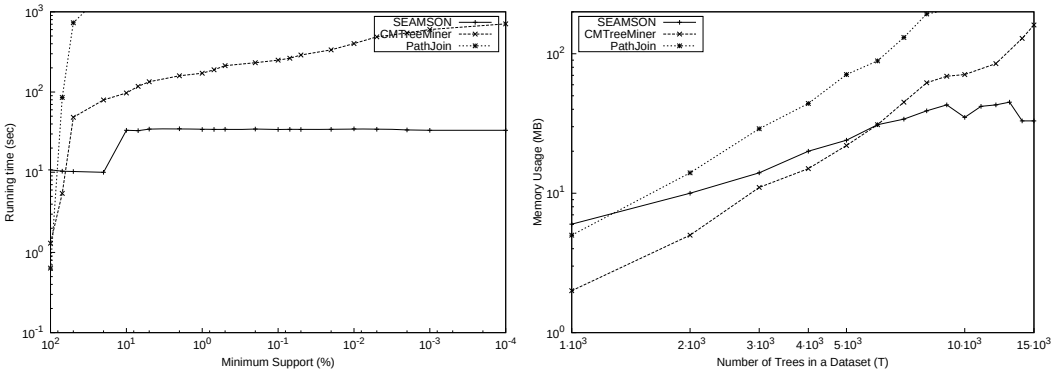
Fig. 15. Maximal frequent subtrees under different minimun supports and number of trees.

5.2 Performance Comparison

In this subsection, we evaluate the performance of SEAMSON in comparison with PathJoin [16] which mines maximal frequent subtrees by adopting the idea of the FP-tree, and CMTreeMiner [26] which is proposed to mine closed and maximal frequent subtrees with tree enumerations. Because PathJoin uses the paths from roots to leaves to help subtree mining, it does not allow any siblings in a tree to have the same labels. Therefore, we changed the synthetic generator in the previous section 5.1 to generate a dataset that meets the requirement. The parameters for the dataset are: $\mathcal{T}$ = 12,000, L = 100, f = 5, d = 5, η = 20%. We compare the above three algorithms in the aspects of time consumption and memory usage.

Fig. 16 (a) compares running time of SEAMSON with those of CMTreeMiner and PathJoin under different minimum supports from 100% to 0.0001%. PathJoin shows a dramatic increase of time consumption as σ 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. After obtaining all frequent subtrees, PathJoin eliminates those that are not maximal. Thus, the number of frequent subtrees is getting bigger as σ is getting smaller, and this is the reason why PathJoin requires the serious waste 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. Compared to those algorithms, SEAMSON shows a different trend of time consumption which is already presented and explained in section 5.1. In spite of decreasing minimum support, it is rarely affected by minimum support. This means that the running time of SEAMSON has been fairly stable condition over any minimum support.

The second comparison is for memory usage during each algorithm's execution. As the result in Fig. 16 (b), the slope of SEAMSON is lower than those of two. At the starting point SEAMSON uses more memory than other algorithms, however, its memory usage slowly increases, then slightly decreases, and increases again trivially. From $\mathcal{T}$ = 10,000 the memory consumption of SEAMSON is stabilized. On the contrary, the mem-



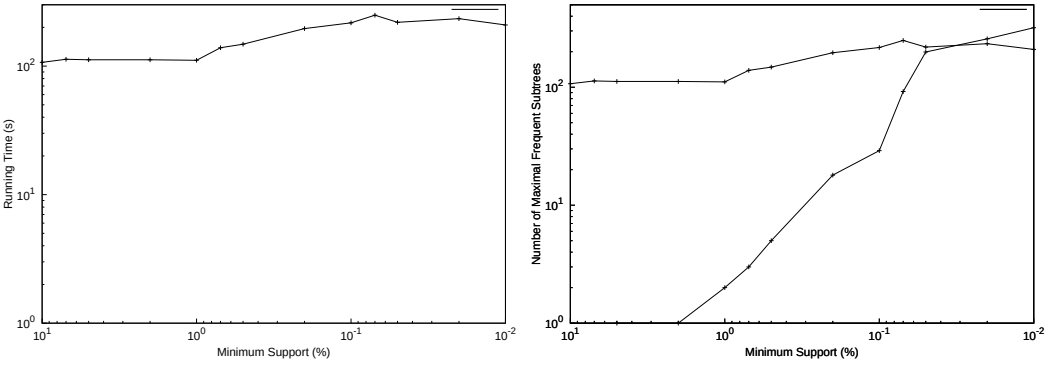
(a) Running time. (b) Memory usage.

Fig. 16. Comparison for three algorithms.

ory usages of CMTreeMiner and PathJoin are linearly increased during the growth of number of trees. PathJoin more consumes the memory than SEAMSON from $\mathcal{T}$ = 2000 and CMTreeMiner exceeds SEAMSON from $\mathcal{T}$ = 6000. The memory usage keeps growing during the algorithms' execution except SEAMSON.

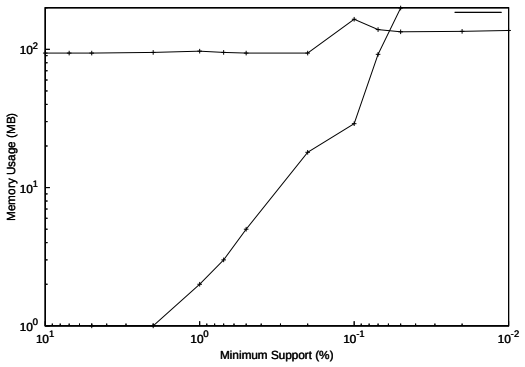
5.3 Real-World Dataset

With CSLOGS dataset, we evaluate SEAMSON running time, number of maximal frequent subtrees, and memory usage. Fig. 17 shows respectively. In the first graph, it can be observed the similar trend presented in Fig. 14 (a), only it is a little bit increased around from σ = 1%. However, afterwards it still shows the stability even though the minimum support is decreased. The following experiment is for how many maximal frequent subtrees can be extracted from CSLOGS. As presented in Fig. 17 (b) the number dramatically increases two times: from 29 to 92 when the minimum support decreases from 0.1% to 0.07% and from 92 to 199 when the support does from 0.07% to 0.05%. The number of maximal frequent subtrees slightly increases after σ = 0.05%, but the slope is gentle.



(a) Minimum support vs. Time. (b) Maximal frequent subtrees vs. Minimum support.

Fig. 17. Stability and reliability test for real world dataset.



(c) Memory usage.

Fig. 17. (Cont'd) Stability and reliability test for real world dataset.

In addition, to check the memory usage for the real world dataset, we perform the experiment and show the result in Fig. 17 (c). It can be noticed that memory usage is rarely affected by the minimum support. The dramatic increase is happened one time when the support drops from 0.2% to 0.1%. In our opinion, it would stem from the fact that the rapid increase of the maximal frequent subtrees presented in Fig. 17 (b).

With CSLOGS, the algorithm SEAMSON shows its stability and reliability both in running time and memory usage even though minimum support decreases.

6. CONCLUSION

In this paper, we were interested in the problem of finding maximal frequent subtrees. To deal with it, first we pointed out inefficiency of some previously published frequent subtree mining algorithms. Second, we proposed a new tree mining algorithm which incorporates tree structures into list structures and thus, it allows us to analyze the intricate trees more thoroughly. Lastly, we performed extensive evaluations to show the benefits of our algorithm for maximal frequent subtrees mining.

The beneficial effect of our method was 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. We plan to extend our work in the following directions. First, some trees in real applications are unrooted, i.e., they are free trees. Extending our algorithm to mining maximal frequent free trees is another challenge. Second, frequent itemset mining has been extended to sequential pattern mining [35] and episode mining [36]. Many databases of labeled trees also have time-stamps for each transaction tree. Mining maximal sequential patterns and episode trees from such databases is one of our future research topics.

REFERENCES

1. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML," in *Pro-*

ceedings of ACM SIGMOD International Conference on Management of Data, 2001, pp. 413-424.
2. R. K. Wong, "The extended XQL for querying and updating large XML databases," in *Proceedings of ACM Symposium on Document Engineering*, 2001, pp. 95-104.
3. M. Arenas, W. Fan, and L. Libkin, "On verifying consistency of XML specifications," in *Proceeding of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database System*, 2002, pp. 259-270.
4. A. Azagury, M. Factor, Y. Maarek, and B. Mandler, "A novel navigation paradigm for XML repositories," *Journal of the American Society for Information Science and Technology*, Vol. 53, 2002, pp. 515-525.
5. R. Praveen and M. Bongki, "Prix: indexing and querying XML using prüfer sequences," in *Proceedings of International Conference on Data Engineering*, 2004, pp. 288-299.
6. Z. Vagena, M. M. Moro, and V. J. Tsotras, "Supporting branched versions on XML documents," in *Proceedings of International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications*, 2004, pp. 137-144.
7. L. I. Rusu, W. Rahayu, and D. Taniar, "Mining changes from versions of dynamic XML documents," in *Proceedings of International Conference on Knowledge Discovery from XML Documents*, LNCS 3915, 2006, pp. 3-12.
8. R. Agrawal, S. Rajagopalan, R. Srikant, and Y. Xu, "Mining newsgroups using networks arising from social behavior," in *Proceedings of International World Wide Web Conference*, 2003, pp. 529-535.
9. R. Cooley, B. Mobasher, and J. Srivastava, "Web mining: information and pattern discovery on the world wide web," in *Proceedings of IEEE International Conference on Tools with Artificial Intelligence*, 1997, pp. 558-567.
10. X. Wang, A. Abraham, and K. A. Smith, "Intelligent web traffic mining and analysis," *Journal of Network and Computer Applications*, Vol. 28, 2005, pp. 147-165.
11. D. Shasha, J. T. L. Wang, and S. Zhang, "Unordered tree mining with applications to phylogeny," in *Proceedings of International Conference on Data Engineering*, 2004, pp. 708-719.
12. S. Zhang and J. T. L. Wang, "Mining frequent agreement subtrees in phylogenetic databases," in *Proceedings of SIAM International Conference on Data Mining*, 2006, pp. 222-233.
13. Y. Chi, S. Nijssen, R. R. Muntz, and J. N. Kok, "Frequent subtree mining – an overview," *Fundamenta Informaticae*, Vol. 66, 2004, pp. 161-198.
14. Y. Chi, Y. Yang, and R. R. Muntz, "HybridTreeMiner: an efficient algorithm for mining frequent rooted trees and free trees using canonical forms," in *Proceedings of International Conference on Scientific and Statistical Database Management*, 2004, pp. 11-20.
15. K. Wang and H. Liu, "Discovering typical structures of documents: a road map approach," in *Proceedings of Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1998, pp. 146-154.
16. Y. Xiao, J.-F. Yao, Z. Li, and M. H. Dunham, "Efficient data mining for maximal frequent subtrees," in *Proceedings of IEEE International Conference on Data Mining*, 2003, pp. 379-386.

17. K. Wang and H. Liu, "Schema discovery for semistructured data," in *Proceedings of International Conference on Knowledge Discovery and Data Mining*, 1997, pp. 271-274.
18. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa, "Efficient substructure discovery from large semi-strucutured data," in *Proceedings of SIAM International Conference on Data Mining*, 2002, pp. 158-174.
19. A. Termier, M.-C. Rousset, and M. Sebag, "DRYADE: A new approach for discovering closed frequent trees in heterogeneous tree databases," in *Proceedings of IEEE International Conference on Data Mining*, 2004, pp. 543-546.
20. C. Wang, M. Hong, H. Pei, H. Zhou, W. Wang, and B. Shi, "Efficient pattern-growth methods for frequent tree pattern mining," in *Proceedings of Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, LNCS 3056, 2004, pp. 441-451.
21. M. J. Zaki, "Efficiently mining frequent trees in a forest: algorithms and applications," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, 2005, pp. 1021-1035.
22. H. Tan, T. S. Dillon, F. Hadzic, E. Chang, and L. Feng, "IMB3-Miner: mining induced/embedded subtrees by constraining the level of embedding," in *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, LNAI 3918, 2006, pp. 450-461.
23. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of International Conference on Very Large Databases*, 1994, pp. 487-499.
24. J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: a frequent-pattern tree approach," *Data Mining and Knowledge Discovery*, Vol. 8, 2004, pp. 53-87.
25. J. Pei, J. Han, B. Mortazavi-Asl, and H. Pinto, "PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth," in *Proceedings of IEEE International Conference on Data Engineering*, 2001, pp. 215-224.
26. Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz, "Mining closed and maximal frequent subtrees from databases of labeled rooted trees," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, 2005, pp. 190-202.
27. J. Paik, J. Nam, D. Won, and U. M. Kim, "Fast extraction of maximal frequent subtrees using bits representation," *Journal of Information Science and Engineering*, Vol. 25, 2009, pp. 435-464.
28. Y. Chi, Y. Yang, and R. R. Muntz, "Canonical forms for labeled trees and their applications in frequent subtree mining," *Knowledge and Information Systems*, Vol. 8, 2005, pp. 203-234.
29. J. Paik, J. Nam, J. Hwang, and U. M. Kim, "Mining maximal frequent subtrees with lists-based pattern-growth method," in *Proceedings of Asia-Pacific Web Conference*, LNCS 4976, 2008, pp. 93-98.
30. J. Paik, J. Nam, H. Y. Youn, and U. M. Kim, "Discovery of useful patterns from tree-structured documents with label-projected database," in *Proceedings of International Conference on Autonomic and Trusted Computing*, LNCS 5060, 2008, pp. 264-278.
31. A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining fre-

quent substructures from graph data," in *Proceedings of European Conference on Principles of Data Mining and Knowledge Discovery*, LNAI 1910, 2000, pp. 13-23.
32. M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Proceedings of IEEE International Conference on Data Mining*, 2001, pp. 313-320.
33. R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993, pp. 207-216.
34. A. Termier, "Extraction of frequent trees in a heterogeneous corpus of semi-structured data: application to xml documents mining," Ph.D. Thesis, Department of Laboratoire de Recherche en Informatique, U.M.R. CNRS 8623, Paris South University, 2004.
35. R. Srikant and R. Agrawal, "Mining sequential patterns: generalizations and performance improvements," in *Proceedings of International Conference on Extending Database Technology*, LNCS 1057, 1996, pp. 3-17.
36. H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequence," *Data Mining and Knowledge Discovery*, Vol. 1, 1997, pp. 259-289.

Juryon Paik (白珠蓮) received the B.E. degree in Information Engineering from Sungkyunkwan University, Korea, in 1997. She received her M.E. and Ph.D. degrees in Computer Engineering from Sungkyunkwan University in 2005 and 2008, respectively. Currently, she is a Research Professor at the Department of Computer Engineering, Sungkyunkwan University. Her research interests include XML mining, semantic mining, and web search engines.

Junghyun Nam (南正鉉) received the B.E. degree in Information Engineering from Sungkyunkwan University, Korea, in 1997. He received his M.S. degree in Computer Science from University of Louisiana, Lafayette, in 2002, and the Ph.D. degree in Computer Engineering from Sungkyunkwan University, Korea, in 2006. He is now an Associate Professor in Konkuk University, Korea. His research interests include information security and retrieval.

Ung Mo Kim (金應模) received the B.E. degree in Mathematics from Sungkyunkwan University, Korea, in 1981 and the M.S. degree in Computer Science from Old Dominion University, USA, in 1986. His Ph.D. degree was received in Computer Science from Northwestern University, USA, in 1990. Currently he is a Professor of School of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include data mining, database security, data warehousing, and GIS.

Dongho Won (元東豪) received his B.E., M.E., and Ph.D. degrees from Sungkyunkwan University in 1976, 1978, and 1988, respectively. After working at ETRI (Electronics and Telecommunications Research Institute) from 1978 to 1980, he joined Sungkyunkwan University in 1982, where he is currently a Professor of School of Information and Communication Engineering. In the year 2002, he served as the President of KIISC (Korea Institute of Information Security and Cryptology). He was the Program Committee Chairman of the 8th International Conference on Information Security and Cryptology (ICISC 2005). His research interests are on cryptology and information security.