International Journal of Database Theory and Application
Vol. 2, No. 1, March, 2009

Finding Frequently Occurred Tree Patterns without Candidate Subtrees Maintenance

Juryon Paik
Dept. of Computer Eng.
Sungkyunkwan Univ.
wise96@ece.skku.ac.kr

Junghyun Nam
Dept. of Computer Science
Konkuk Univ.
jhnam@kku.ac.kr

Ung Mo Kim
Dept. of Computer Eng.
Sungkyunkwan Univ.
umkim@ece.skku.ac.kr

Abstract

The most commonly adopted approach to find valuable information from trees data is to extract frequently occurring subtree patterns from them. Because mining frequent tree patterns has a wide range of applications such as xml mining, web usage mining, bioinformatics, and network multicast routing, many algorithms have been recently proposed to find the patterns. However, existing tree mining algorithms suffer from several serious pitfalls in finding frequent tree patterns from massive tree datasets. Some of the major problems are due to (1) the computationally high cost of the candidate maintenance, (2) the repetitious input dataset scans, and (3) the high memory dependency. These problems stem from that most of these algorithms are based on the well-known apriori algorithm and have used anti-monotone property for candidate generation and frequency counting in their algorithms. To solve the problems, we base a pattern-growth approach rather than the apriori approach, and choose to extract maximal frequent subtree patterns instead of frequent subtree patterns. We would present some new theorems derived from and evaluate the effectiveness of the proposed algorithm in comparison to the previous works.

1. Introduction

1.1. Motivation

One of the most general approaches for modeling complex structured data is to prescribe the data with tree structure. In the database area [10, 13], XML documents are rooted trees where the nodes represent elements or attributes and the edges represent element-subelement and attribute-value relationships. In Web traffic mining, access trees are used to represent the access patterns of different users [2]. In the analysis of molecular evolution, an evolutionary tree (or phylogeny) is used to describe the evolution history of certain species [18]. In computer networking, multicast trees are used for packet routing [4].

With the ever-increasing amount of available tree data, the ability to extract valuable information from them becomes increasingly important and desirable. However, the problem of finding information on tree data has not been extensively studied, in spite of its applicability to a variety of problems. The first step toward finding information from trees is to mine the subtrees frequently occurring in the trees. Frequent subtrees in a database of trees provide useful knowledge in many cases such as gaining general information of data sources, mining of association rules, classification as well as clustering, and helping standard database indexing [5]. However, the discovery of frequent subtrees appearing in a large-scaled tree dataset is not an easy task. As observed in Chi et al's paper [7], due to combinatorial explosion, the number of frequent subtrees usually grows exponentially with the size (number of nodes) of the tree and, therefore, mining all frequent subtrees becomes infeasible.

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

1.2. Related Work

The most popular approaches to find useful information from trees are either apriori-based [3] or frequent-pattern-growth(FP)-based [9]. The algorithms based on the former approach extract frequent subtrees by the well known anti-monotone property: *every non-empty subtree of a frequent tree is also frequent*, for candidate-generate-and-test. Since it provides significant reduction of the size of candidate sets and leads to good performance gain, various techniques have been applied to improve their efficiency [14, 1, 17, 8]. They are efficient and scalable when short patterns are usually extracted from sparse datasets. What if the datasets are dense and there are a lot of long patterns? That may degrade the mining performance dramatically because a large number of candidates need to be generated and tested.

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

The goal of all the above mentioned algorithms is to discover all frequent subtrees from a database of trees. However, as observed in Chi et al's papers [8], the number of frequent subtrees usually grows exponentially with the tree size, therefore, minding all frequent subtrees becomes infeasible for a large number of trees. The algorithms presented by Xiao et al. [16] and Chi et al. [6] attempt to alleviate the huge amount of frequent subtrees by finding and presenting to end-users only the maximal frequent subtrees. The former uses a new compact data structure, FST-Forest, to store compressed trees, representing the trees in the database. Nevertheless, the algorithm uses post-processing techniques that prune away non-maximal frequent subtrees after discovering all the frequent subtrees. Therefore, the problem of the exponential number of frequent subtrees still remains. The latter directly aims at closed and maximal frequent subtrees only. However, it bases on the enumeration trees which are one of branches of apriori techniques. Therefore, this algorithm may have the potential problem if a dataset is dense and there are a lot of long patterns.

Handling the maximal frequent subtrees is an interesting challenge, though, and represents the core of this paper. From the perspective of the design of pattern extraction algorithm, we consider the challenging problem of efficient avoidance of subtree generations to find maximal frequent subtrees. The proposed solution relies on a label-projected database which is introduced in this paper, and it leverages the properties of the proposed approach that is inherently list-based as opposed to tree-based. Our method not only gets rid of the process for

infrequent tree pruning, but also totally eliminates the problem of generating candidate subtrees. Hence, it significantly improves the whole mining process.

2. Key Features of SEAMSON

The problems mentioned in the introduction have motivated us to take a different approach by eliminating the generation of candidate subtrees and ensuring the acquisition of valid maximal frequent subtrees. In this section, we introduce a new pattern-growth algorithm SEAMSON (Scalable and Efficient Algorithm for Maximal frequent Subtrees extractiON) based on its interesting definitions and important features. The initial version of SEAMSON was presented in [11, 12].

2.1. Label projected database

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

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

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

The structure of a label-list is similar to that of a linked list in that it has a head and a body. The only difference is formation and functionality of the head. The head of label-list points the first object in a body just like the ordinary head of linked list. Along with the indication, the head of label-list gives information on which node indexes have been mapped to a projected label. The body of label-list follows immediately its corresponding head. The main concerns of the body are to evaluate how many trees have the key in its paired head and to find parents positions of the nodes in the head. The former is for dealing with the frequencies of each label, while the latter is for handling the hierarchical information of the label. To achieve such intentions, the structure of body is a sequence of *members* which is arranged in a linear order. Each member is an object with a key field, one link field pointing to the next member, and one satellite data field.

As a key a tree index number is used, which this means that the label in a corresponding head has been assigned to the nodes in the tree. During the database scan, the member is generated and inserted into bodies of label-lists. The newly inserted member is added to the end of a proper body and the pointer field of its previous member points this new member. The body

of a label-list provides the method we can judge in how many trees have the label of a current label-list, that is the number of members in body and we define it as size of label-list.

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

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

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

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

The parent indexes in members should have frequent labels in order the subtrees with size 2 to be frequent, and this is checked by the second and third condition in Definition 3. However, it rarely happens that parent index also has frequent projected label, where the parent is switched one of its ancestor whose label is frequent or the root, because in SEAMSON the considered relation between two nodes in a tree is ancestor-descendant relation not that of parent-child. Some means to efficiently manage parent indexes in members is necessary to check the frequency of and to switch them, in order label lists to be frequent.

Figure 1 shows an depicted example of how $\mathbb{L}_{dic}$ and its label lists are constructed and managed from the database $\mathcal{D}$. For easy distinction between nodes in different trees, we assign unique consecutive indexes in pre-order traversal. The bodies of lists decide the frequencies of corresponding labels. For instance, the label A does not satisfy the given minimum value which is set ⅔ because it has only 1 member in the body of A-list.

2.2. Table for candidates consideration

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

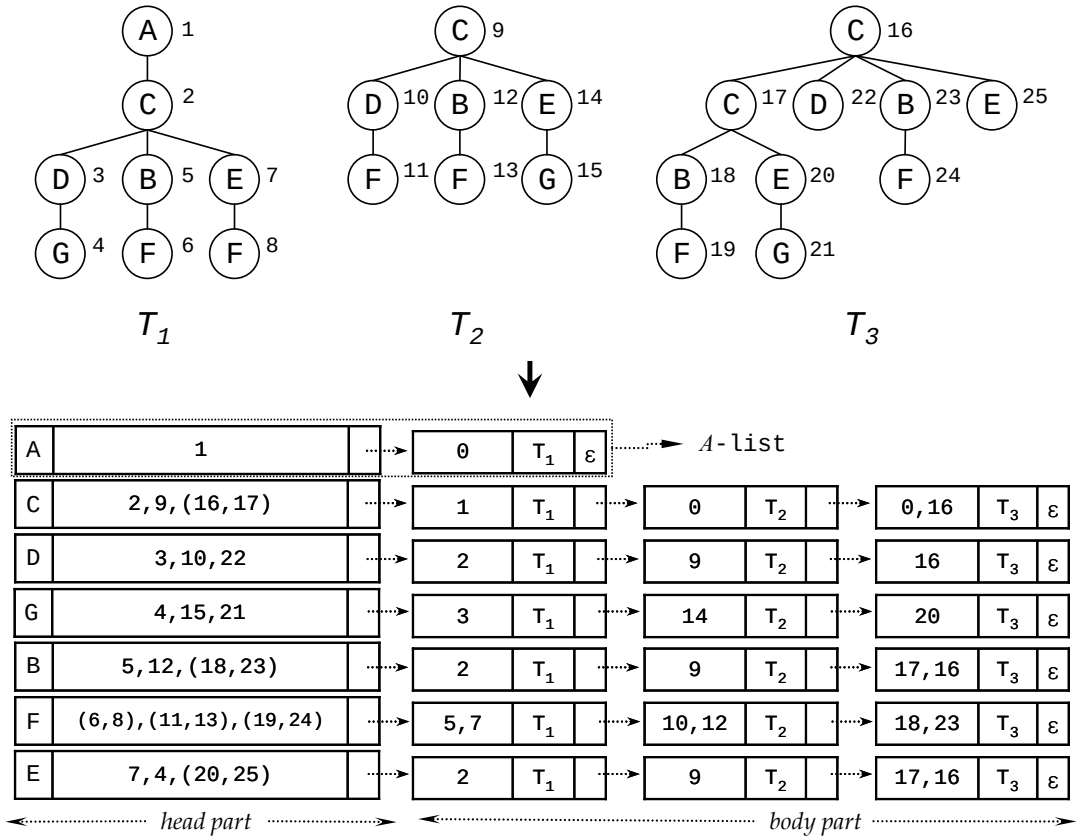


Figure 1. Original Tree Database and its label dictionary L_{dic}

2.2. Table for candidates consideration

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

It is easily determined by $\mathbb{T}_C$ if the label of a given parent index has projected as one of label lists in $\mathbb{L}_{dic}^{f}$, then, what about the switching in case the label is in $\mathbb{T}_C$? As the required method we define the following.

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

Let l_1-list($\in \mathbb{L}_{dic}^{f}$) be a currently tested label list and $|l_1\text{-list}|$ be m. Each member in its l_1-list.b is required to undertake the following. Let any parent node 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 $\mathbb{T}_C$. If $\mathcal{L}(p)$ is not found between keys, p has frequent projected label. Thus, p becomes the proper Λ_p, and the process is terminated. Otherwise, p's label is infrequent projected. (3) To uncover the desired location of $\mathcal{L}(p)$, the index is computed by $\mathcal{H}_b(\mathcal{L}(p))$.

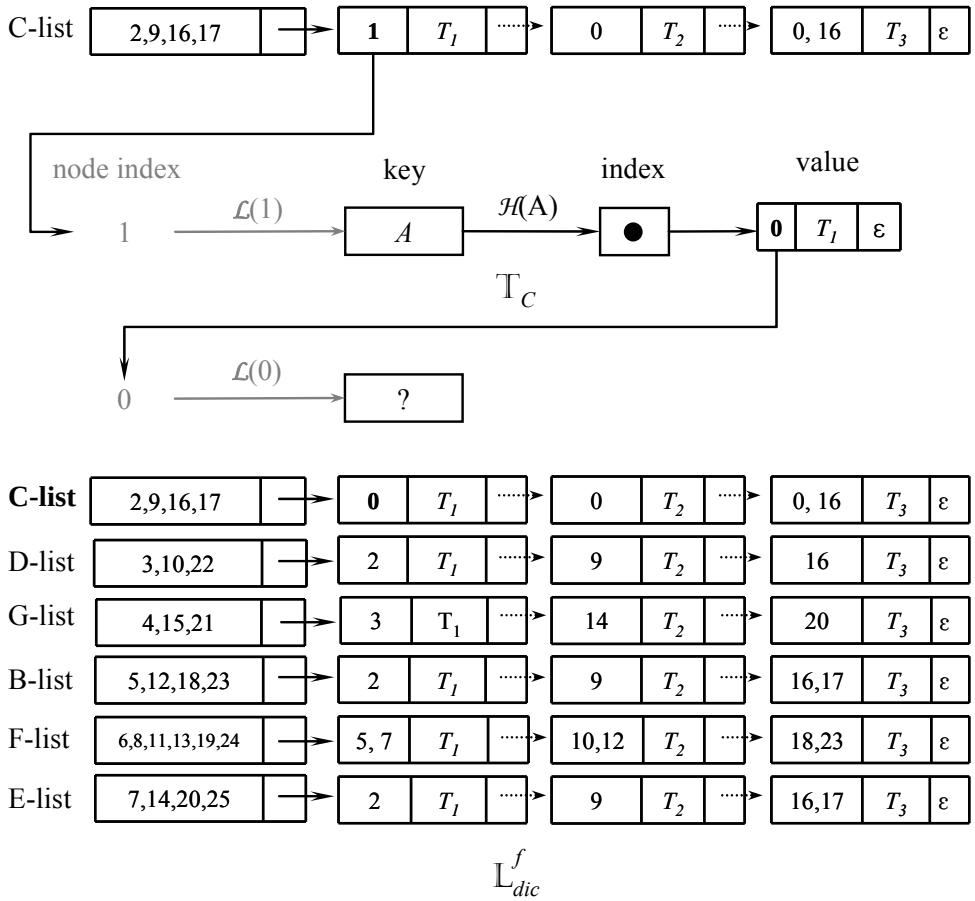


Figure 2. Discovery and replacement of Λ_1

(4) According to $\mathcal{H}_b(\mathcal{L}(p))$, the value $\mathcal{L}(p)$-list is returned. (5) Since the value is $\mathcal{L}(p)$-list.b, it consists of several members. The node indexes in the members correspond to those of p^1. (6) As backtracking, (1) to (4) are done against every p^1 in the value. (7) The p^1 whose label is found as the key of $\mathbb{T}_C$ iteratively performs (3) through (6) until Λ_{p^1} is found. For all m members the steps (1) to (7) are performed to fill themselves with only closest frequent ancestors. What if a proper Λ_p is not acquired until end of backtracks? In such case, none of p's ancestors including p itself have frequent projected labels. That means the node index

whose label is l_1 has no parent node, and therefore, it becomes the root. Hence, Λ_p is set by 0 to indicate `it is the root position'.

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

2.3. Label list extension

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

Definition 5 (lable list extension) *Given $\mathbb{L}_{dic}^{f}$, let a current label list be l-list and p be one of parent indexes in its members. For l-list, firstly a symbolic node whose label is l is set and the node is prepared to join with its parent. The second symbolic node is set from the parent index p. Its label is easily obtained by $\mathcal{L}(p)$ and the corresponding $\mathcal{L}(p)$-list is in $\mathbb{L}_{dic}^{f}$ due to the definition 3. Consequently, the node labeled by l is joined to the node labeled by $\mathcal{L}(p)$. We call this operation label list extension and denote $l \leftarrow \mathcal{L}(p)$ where `$\leftarrow$' indicates the direction of parent to child.*

After completing the work of extension, the labels in head parts are connected each other via nodes. The structure of the result is a tree whose root is labeled by /. This tree contains all of potentially maximal frequent subtrees and thus is named Potentially Maximal Pattern tree (PMP-tree in short). The detailed is followed.

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

Along with determination of seeds, one table is constructed with labels and seeds. This table facilitates the traversal of PMP-tree by providing location information of seeds. Once a label is given to the table, its corresponding seed's position in PMP-tree is retrieved. Since the table searches nodes based on labels, we name it Label Header Table, $\mathbb{T}_L$, and it is built

simultaneously with seeds. $\mathbb{T}_L$ is composed of two columns: label and seed# (location of seed). During label list extensions, $\mathbb{T}_L$ provides facilitated PMP-tree traversal as well as fast retrieval of proper seeds' locations.

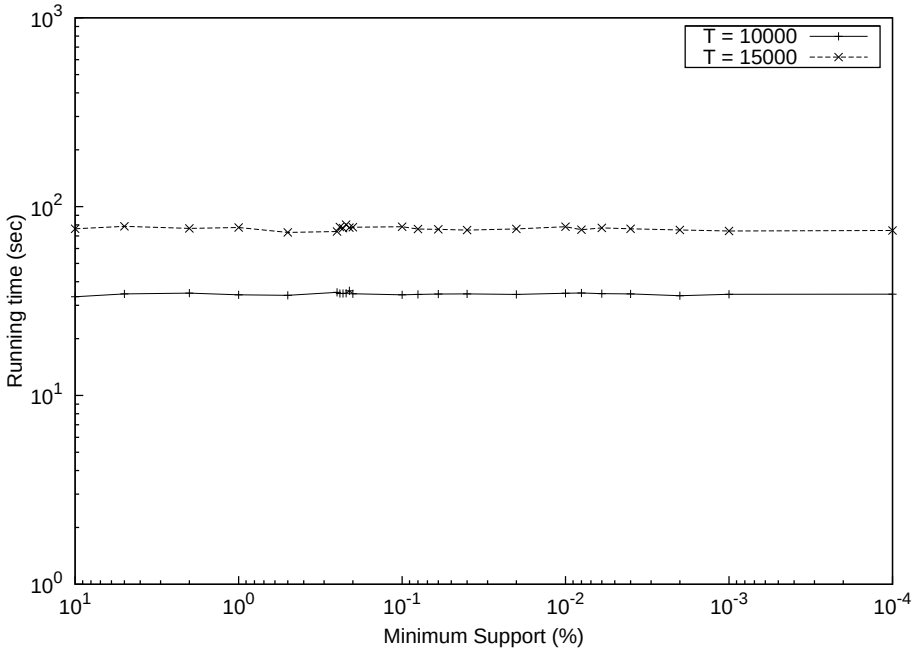
After finishing the first scan, total seven seeds are generated and they are related their labels in $\mathbb{T}_L$ via the seed\# column. The current seeds have empty parents, empty successors, and 0 counters. Since not yet performed the label list extension, no relation is expected between seeds. To process actual *l*-list extension over the seeds, $\mathbb{L}_{dic}^{f}$ is scanned for the second time and last time. During this scanning period, the body parts in turn are analyzed, which have been skipped for the first scan. The parent indexes of each element are spliced with the existing seeds by following steps: (1) Let a seed of a current reading label list, *l*-list, be s_l associated with a label *l* in $\mathbb{T}_L$. (2) For a parent index *p* in an element of *l*-list.b, obtain a label of *p* by $\mathcal{L}(p)$. (3) According to Definition 3, the label $\mathcal{L}(p)$ definitely forms a record of $\mathbb{T}_L$ and is associated with one of seeds. (4) Look up the table $\mathbb{T}_L$ to get the seed# which corresponds to $\mathcal{L}(p)$; let the corresponding seed be $s_{\mathcal{L}(p)}$. (5) If s_l has empty prt which means it is not yet linked with parent node, the seed $s_{\mathcal{L}(p)}$ is the firstly appeared its parent. Hence, the seed# of $s_{\mathcal{L}(p)}$ is stored in the prt of s_l; this action directly perform $s_l \leftarrow s_{\mathcal{L}(p)}$. Also, the cnt of $s_{\mathcal{L}(p)}$ is incremented by one because a edge between s_l and $s_{\mathcal{L}(p)}$ has been formed and it occurred.

It is trivial work to link s_l to $s_{\mathcal{L}(p)}$ when prt of s_l is empty. The extension of $s_l \leftarrow \$$ without any other work. However, what if prt of s_l is already preoccupied by another seed#, say $s_{\mathcal{L}(p)}$? There are two cases depending on whether $s_{\mathcal{L}(p)} = s_{\mathcal{L}(q)}$ or not, and the step (5) is replaced by followings: (5-1) If $s_{\mathcal{L}(p)} \neq s_{\mathcal{L}(q)}$, we add a successor at the end of the seed s_l, or at the last successor if s_l already has successors. Without using successors one child can have several parents and it causes the graph structure which requires NP-complete subtree decomposition. In case of that s_l has successors, the values in all prt of s_l and its successors have to be compared with the seed# of $s_{\mathcal{L}(q)}$. If not found the same value, $s_{\mathcal{L}(q)}$ is another parent seed of s_l, thus, a new successor for s_l is attached at the end of successors and it is linked with $s_{\mathcal{L}(q)}$. (5-2) In case of that $s_{\mathcal{L}(p)}$ is equal to the preoccupied prt value $s_{\mathcal{L}(q)}$ or any one of prt values in s_l's successors, cnt of s_l or corresponding successor is incremented by 1. (6) The procedures from step (2) to (5-2) is repeated until all parent indexes in $\mathbb{L}_{dic}^{f}$ are considered.

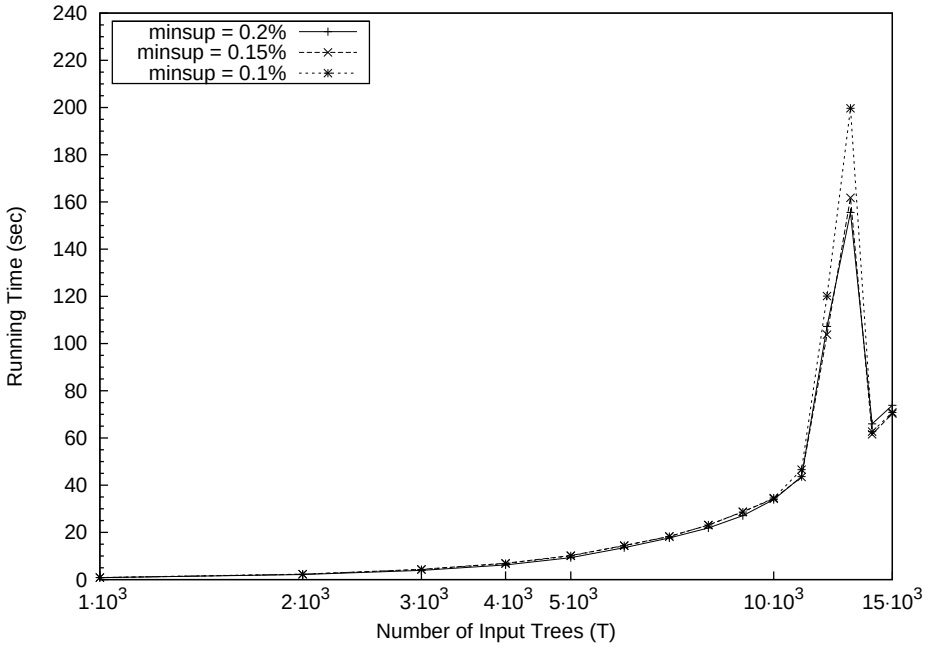
3. Experiments

The synthetic datasets generator constructs a set of trees, $\mathcal{D}$, based on some parameters supplied by the user: $\mathcal{T}$: the number of trees in $\mathcal{D}$, L : the set of labels, f : the maximum branching factor of a node, d : the maximum depth of a tree, ρ : the random probability of one node in the tree to generate children or not, η : the average number of nodes in each tree in $\mathcal{D}$. We used the following default values for the parameters: the number of trees $\mathcal{T}$ = 10,000, the number of labels L = 100, the maximal branch factor f = 5, and the maximum depth d = 5.

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



(a) support vs. time



(b) input trees vs. time

Figure 3. Scalability of SEAMSON

From Figure 3(a), 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. With minsup becoming smaller, there is no big difference in execution time for both datasets. This is because SEAMSON relies on the number of labels not the number of nodes. Thus, it is very efficient for datasets with varying and growing tree sizes. Then, Figure 3(b) shows the scalability with size of dataset - the number of input trees. The parameter $\mathcal{T}$ varies from 1,000 to 15,000 with η = 20. We evaluate three different minsup, 0.2%, 0.15%, and 0.1%. The corresponding graphs show 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 sizes of $\mathbb{L}_{dic}^{f}$ and its label lists are maximized with 100 distinct node labels when the number of input trees reaches at 12,000 and 13,000.

4. Conclusion

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

The beneficial effect of our method is that it not only got rid of the process for infrequent tree pruning, but also eliminated totally the problem of candidate subtrees generation. Hence, we significantly improved the whole mining process.

References

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

[2] S. Adali, T. Liu and M. Magdon-Ismail, *Optimal Link Bombs are Uncoordinated*, Proc. of the 1st Workshop on Adversarial Information Retrieval on the Web, pp.58-69, 2005.

[4] J. Cui, J. Kim, D. Maggiorini, K. Boussetta and M. Gerla, *Aggregated Multicast–A Comparative Study*, Cluster Computing, vol.8, no.1, pp.15-26, 2005.

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

[5] Y. Chi, S. Nijssen, R. R. Maggiorini and J. N. Kok, *Frequent Subtree Mining–An Overview*, Fundamenta Informaticae, vol.66, no.1-2, pp.161-198, 2005.

[6] 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, no.3, pp.190-202, 2005.

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

[8] 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, no.2, pp.203-234, 2005.

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

[10] R. Praveen and M. Bongki, *Prix: Indexing and Querying XML Using Pr¨ufer Sequences*, Proc. of the 20th Int'l Conf. on Data Engineering, pp.288-299, 2004.

[11] J. Paik and U. M. Kim. A simple yet efficient approach for maximal frequent subtrees extraction from a collection of xml documents. In *Proc. of 7th Int'l Conf. on Web Information Systems Engineering*, pages 94–103, 2006.

[12] J. Paik, J. Lee, J. Nam, and U. M. Kim. Mining maximally common substructures from xml trees with lists-based pattern-growth method. In *Proc. of IEEE Int'l Conf. on Computational Intelligence and Security*, pages 209–213, 2007.
[13] L. I. Rusu, W. Rahayu and T. Taniar, *Mining Changes from Versions of Dynamic XML Documents*, Proc. of Int'l Conf. on Knowledge Discovery from XML Documents, LNCS vol.3915, pp.3-12, 2006.
[14] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang and B. Shi, *Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining*, 8th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, LNAI vol.3056, pp.441-451. 2004.
[15] K. Wang and H. Liu, *Schema Discovery for Semistructured Data*, Proc. of the 3rd Int'l Conf. on Knowledge Discovery and Data Mining, pp.271-274, 1997.
[16] Y. Xiao, J.-F. Yao, Z. Li and M. H. Dunham, *Efficient Data Mining for Maximal Frequent Subtrees*, Proc. IEEE Int'l Conf. on Data Mining, pp.379-386, 2003.
[17] M. J. Zaki, *Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications*, IEEE Transactions on Knowledge and Data Engineering, vol.18, no.8, pp.1021-1035, 2005.
[18] S. Zhang and J. T. L. Wang, *Mining Frequent Agreement Subtrees in Phylogenetic Databases*, Proc. of the 6th SIAM Int'l Conf. on Data Mining, pp.222-233, 2006.

Authors

Juryon Paik
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. As of March 2008, she is a research professor at the Department of Computer Engineering, Sungkyunkwan University. Her research interests include XML mining, XML data replication, and similarity computation.

Junghyun Nam
received the B.E. degree in Information Engineering from Sungkyunkwan University, Korea, in 1997 and the M.S. degree in Computer Science from University of Louisiana, Lafayette, in 2002. His Ph.D. degree was received in Computer Engineering from Sungkyunkwan University, Korea, in 2006. Since 2007, he has been an assistant professor of the Department of Computer Science, Konkuk Univeristy, Korea. His current research interest is in the area of cryptography and computer security.

Umg Mo Kim
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 full professor of School of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include data mining, database security, data warehousing, and GIS.