Pushing Sequential Constraints into Unordered Tree Mining Algorithms

Wei Gao Zhihui Du* Sen Zhang

Abstract— **We propose to enhance existing unordered tree problems by considering two types of sequential pattern constraints, namely the scope constraint and the central pattern constraint. Furthermore, we develop a new algorithm to solve the proposed new mining problem, by considering the user specified constraints in its constituent components of pattern generation and pattern pruning algorithms. Experimental results demonstrate that our approach is capable of finding patterns containing the target constraints.** *keywords: Tree Mining, Constraints, Unordered Trees, Structured Pattern*

I. Introduction

Unordered trees find many applications where they are used to represent RNA [2], XML [3], phylogeny data [4], [5], [12], industrial parts [10], and chemistry compound data [11], etc. Therefore, mining frequent unordered trees has become an important problem in the data mining field.

The unordered trees considered by this work are rooted, labeled and unordered trees, where each of them has a root, each node has a label, and the left-to-right order among siblings is unimportant. In this work, we study new constraint specification methods, namely the scope constraint and the central pattern constraint, which allow users to express a specific set of interesting patterns in unordered tree mining tasks. The proposed constraints can be conceptually regarded as a growing and pruning framework. In other words, the user provides some central pattern cores and a topic scope, the mined result is a set of pattern trees constructed from the central pattern cores and pruned to fit the defined topic scope. Instead of mining all the pattern trees first and then filtering out uninteresting ones from the final result, our algorithm pushes these user-specified constraints into the mining process in such a way that only patterns satisfying them are kept in the intermediate steps. Since this approach leads to non-Aprori patterns. we have designed a new upward growth method in order to still generate larger subtrees the constraints from the previous level patterns meeting the constraints.

As a motivating example, we can model a set of web browsing trace data of an Internet surfer as a data tree instance, based on the hierarchical relationships among topics suggested in [1], as illustrated in Fig. 1. An Internet surfer could be interested in certain concrete topics. For example,

Wei Gao is with the School of Aerospace, Tsinghua University, Beijing 100084, P. R. China (email: gaow13@mails.tsinghua.edu.cn)

Zhihui Du is with the Department of Computer Science, Tsinghua University, Beijing 100084, P. R. China (email: duzh@tsinghua.edu.cn)

*Contact Author

Sen Zhang is with Department of Mathematics, Computer Sciences and Statistics, State University of New York, College at Oneonta, Oneonta, NY 13820, USA. (email: zhangs@oneonta.edu)

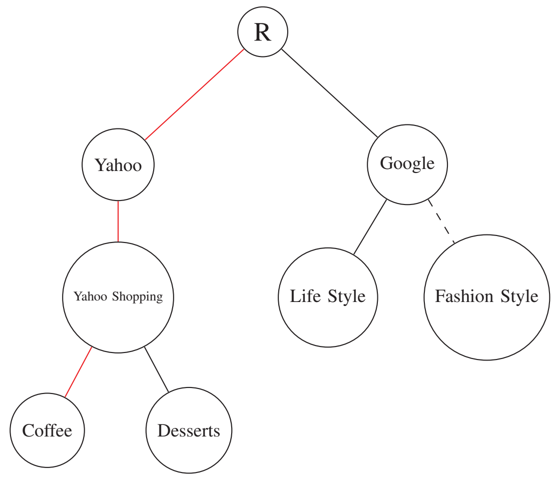


Fig. 1: A data tree usually contains both related and unrelated information from the perspective of a user. R represents an artificial root.

a coffee dealer might be especially interested in a weblog path ending in his/her products (brands of coffee), which are the most specific items according to the topic hierarchy. These patterns would serve as the central pattern cores (mark in red line). Additionally, limiting the scope of the mining can exclude irrelevant information. For instance, the coffee dealer might not be interested in the fashion styles of the consumers. Our method aims to mine patterns growing from the cores and then pruned to focus on the defined topic scope. The mining results in this example can be interpreted as the shopping preference of target consumers who are interested in $Coffee$. As the patterns growing from cores mainly come from $Yahoo$, $Desserts$ could also be interesting or relevant. The mined shopping preference also does not contain irrelevant information, like the fashion styles of the consumers, as the pattern are pruned to focus on the user-defined topic scope.

II. Notations and Preliminaries

Let $\lambda=\{l_1, l_2, \dots l_n\}$ be a finite set of labels built upon an alphabet, where elements can be alphabetically ordered. A rooted, labeled tree t is a directed acyclic graph represented by a 5-tuple, $t=(V, r, E, S, L)$, where V, r, E, S, and L, respectively, denotes the set of nodes, a distinguished root node, the set of parent-child edges, the sibling relation, and the labeling function of V. More specifically, $V=\{v_1, v_2, \dots, v_k\}$ is the set of nodes, where k denotes the size of the tree which is simply the size of the node set, i.e., $k{=}|t|{=}|V|$. Without loss of generality, we assume the elements in V are uniquely and consecutively indexed according to the preorder depth-first traversal of t. Therefore, V can also be represented by their indexes $\{0, 1, 2, \ldots, k{-}1\}$. Obviously, the index of the root of t is 0 and the index of the rightmost leaf, i.e., the last node, is $k{-}1$.

$E \in V{\times}V$ is the set of parent-child edges. The root r has no parent and a leaf node has no children. If $\{u, v\} \subset E$, then we say that u is the parent of v, and v is a child of u, denoted by $u{=}parent(v)$ and $v \in children(u)$. For every node $v \in V$, there is a unique path from the root r to v. If a node u is on the path from the root r to a node v, the node u is an ancestor of the node v, denoted by $u \in ancestors(v)$ and v is a descendant of u, denoted by $v \in descendents(u)$. S represents the left-to-right sibling ordering. If $(v_i, v_j) \in S$, then v_i is the immediate left sibling of v_j.

A rooted unordered labeled tree means that there is not a predefined ordering among each set of siblings, or this ordering is not important. L is a mapping function assigning labels to nodes $L : V \Rightarrow \lambda$. The label of a node with index i is denoted by $l(i)$, or $label(i)$. The depth of v is the number of nodes from r to v. A k-subtree (or k-pattern) is a subtree having exactly k nodes. Furtheremore, a rooted labeled tree T'(either ordered or unordered) with vertex set V' and edge set E' is an embedded subtree of another rooted labeled tree T with vertex set V, edge set E, if and only if (1) $V' \subset V$ (2) the label of the nodes of V' in T is preserved in T' (3) $(v_i, v_j) \in E$, where v'_i is the parent of v'_j in T', only if v_i is an ancestor of v_j in T. The problem of frequent tree mining can be formally stated as follows.

Frequent Tree Mining Problem. Let t and s denote a data tree and a pattern tree, respectively, we say $support(s, t){=}1$ if s is a subtree of t. Furthermore, let $TS{=}\{t_1, t_2, \ldots t_m\}$ denote a set of data trees, the support of s in the tree set TS is defined as $support(s, TS){=}\sum_{i=1}^{m} support(s, t_i)\backslash|TS|$. A subtree is frequent if its support is greater than or equal to a user-specified minimum support threshold, denoted by $minsup$. A set of all frequent subtrees of size k is denoted by FST_k. Given a set of data trees, the frequent tree mining problem aims to find all the subtrees from them with a support greater than $minsup$.

III. Flexible User-Defined Constraints Specification on Tree-Based Mining Problems

To incorporate user-defined constraints into the tree mining algorithms, two principal components are needed. The first one is a flexible constraint specification method, for the user to express a set of patterns of their interests. The second is an efficient algorithm to find all the specified patterns from the input dataset. In this section, we mainly focus on the first component. The second component is presented in section V.

Conceptually, a tree can be viewed as a "bag of sequences," as shown in Fig.2. Thus, to apply constraints to the mined subtrees, a possible method is to apply constraints on the sequences in its "bag of sequences." Based on this observation, we preserve the order of ancestor-decedent to "linearize" a tree. Consequently, multiple (sub)sequences

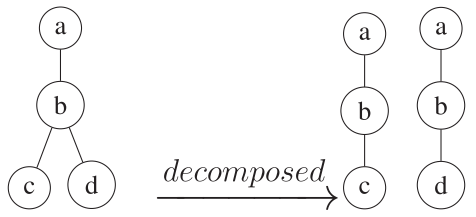


Fig. 2: A rooted labelled tree t_1 can be decomposed into its "bag of root-leaf sequence" $\{s_1, s_2\}$.

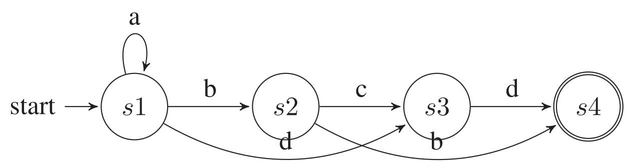


Fig. 3: A deterministic finite automaton for regular expression $a * (bb|bcd|dd)$, adapted from [1].

might be embedded in a tree, for example, sequence $<a, b>$ is an embedded sequence of the tree t_1, as shown in Fig. 2. However, we consider root-to-leaf sequences only, since any embedded sequence must be embedded in root-leaf sequences. As a result, to apply constraints on subtrees, each subtree is modeled as a "bag of root-leaf sequences," and constraints are applied to the sequences in the bag.

Based on the bag of root-leaf sequences model, many existing constraint specifications in sequence mining, like [3], [1], [8], [9], can be directly adopted to formulate constraints in tree mining problems.

IV. Problem Formulation

A. Sequence, Regular Expression Constraints and Finite Automata

For completeness, a brief review is given here on the most important principles of regular expression (RE) constraints. For a comprehensive background of RE family constraints, please refer to [1].

A RE constraint is a RE over the alphabet of tree labels using the set of RE operators, such as the disjunction (|), the Kleene closure (*) and the concatenation. A RE is equivalent to a deterministic finite automaton (DFA). Fig. 3 shows a finite automaton for the regular expression $a^*(bb|bcd|dd)$. Thus, a RE constraint specifies a language of sequential patterns that is of interest to the user. By using the concept of regular expression (RE) and finite automata, the following several constraints on sequential patterns were described in [1].

Definition 4.1 (Naive Constraints): A sequence s is said to obey the naive constraint of a RE constraint R_N, if it does not contain any labels that do not appear in R_N.

Definition 4.2 (Legal Constraints): A sequence s is said to obey the legal constraint of a RE constraint R_L, if there exists one state si of the automaton A_L for R_L, that every state transition of A_L is defined when following the sequence of transitions for the elements of s from si.

Definition 4.3 (Valid Constraints): A sequence s is said to obey the valid constraint of a RE constraint R_V, if there exists one state si of the automaton A_V for R_V, s is legal with respect to si and the final state of the transition path from si on the input sequence s is an accept state of A_V.

Remark 1: Another strengthened formulation of legal constraints on sequence s is: A sequence s is said to obey the legal constraints of a RE constraint R_L, if every state transition of A_L is defined when following the sequence of transitions for the elements of s, from the start state of R_L.

For an RE R, let S_N, S_L and S_V be the sets of sequences that obey naive, legal and valid constraints, then $S_V \subseteq S_L \subseteq S_N$. which is informally to say, the naive constraint is the weakest one, and the valid constraint is the strongest among three. Thus, a sequence set passing the valid constraints would be the most specific, that mostly suits the interest of a user. It is noted that naive and legal constraints are Apriori. If a sequence s obeys naive or legal constraints of RE R, all its sub-sequences must obey naive or legal constraints specified by R. On the contrary, the valid constraint is not Apriori, i.e. removing the last label of a valid sequence s violates the valid constraint.

B. Constraint Formulation

In this subsection, the following constraints are formulated, with their associated motivations.

Definition 4.4 (Scope Constraints): A pattern tree t is said to obey the scope constraints, if **all** the root-leaf sequences in the "bag of sequences" of t obey RE constraints specified by R_{scope}. R_{scope} is a naive or legal constraint defined in the previous subsection.

Definition 4.5 (Central Pattern Constraints): A pattern tree t is said to obey the central pattern constraints, if **there exists one of** the root-leaf sequences of t obey RE constraint specified by $R_{central}$, $R_{central}$ is a valid constraint defined in the previous subsection.

Remark 2 Other constraint specification methods on sequence mining, like [3], [8], [9] can also be used for scope constraint and central pattern constraint. The RE family constraints in [1] are selected for their general form and expressiveness.

Remark 3 The scope constraint formulated here obeys the Apriori rule, i.e., a tree t must satisfy scope constraint if t is a subtree of t', and t' satisfies scope constraint. However, the central pattern constraint does not obey Apriori rule.

The scope constraint can be regarded as a pruning process, as shown in Fig. 4. The original pattern tree with no constraints is shown in solid black line. After being applied with certain scope constraints, some of its branches might not be legal. Therefore, this tree is no longer a frequent pattern tree, despite its subtrees whose branches are all legal can still be frequent patterns. This is a part of the process of pruning uninteresting items from pattern-trees.

The central pattern constraint can be regarded as a growing process, as shown in Fig. 5. The user defines a set of "cores" by central pattern constraints. The mining result can be seen as pattern trees growing from sequences that obey the defined constraints (marked in red), i.e., the "cores" specified by the user. The user obtains unknown information (marked in black) that is correlated with core patterns capturing the user's interest.

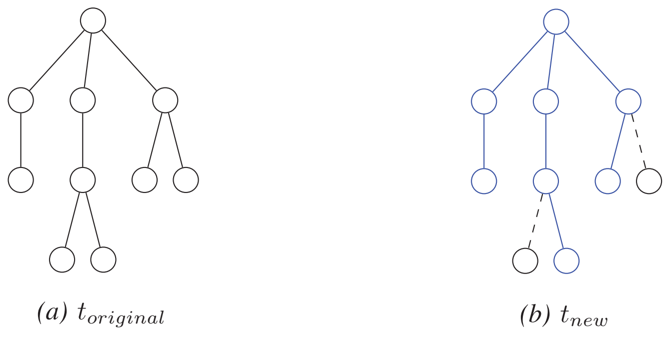

(a) $t_{original}$ (b) t_{new}

Fig. 4: Illustration of scope constraint as a pruning process. A frequent pattern tree with no constraints t_{origin} is in solid black line. When perform tree mining with scope constraints, t_{origin} might violate scope constraints. Only sub-trees of t_{origin} with pruning of all illegal branches (in black dash line) like t_{new} (in blue solid line) are preserved.

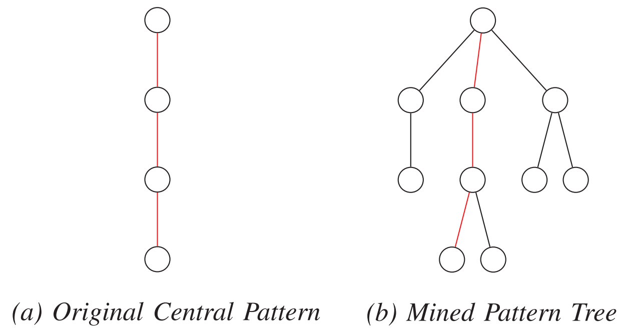

(a) Original Central Pattern (b) Mined Pattern Tree

Fig. 5: Illustration of central pattern constraint as a growing process. The mining result can be seen as pattern trees growing from sequences that serve as central sequential patterns (in red). The user obtains unknown information (in black) that is correlated with a central pattern, which can potentially be of his/her interest.

C. Mining Task Formulation

The input of the problem is a set of rooted, labeled and unordered trees as the dataset, an RE for central pattern constraint $RE_{central}$, an RE for scope constraint RE_{scope}, and a user-defined minimal support value $minsup$. The output of the mining algorithm is a set of frequent embedded subtrees that obey both the scope constraint and the central pattern constraint.

Tables I and II show a typical example of how our constraint-aware tree mining algorithm works. The input parameters are described in the Table I; and some representative subtrees are listed in Table II. The algorithm should filter out subtrees that violate constraints or whose occurrence is smaller than $minsup$. For notational simplicity, the string encoding [5] is used to represent trees. This representation is to add a special symbol (-1 in this example) to the depth-first left to right traversal of a tree, if there is a backtrack from a child node to its parent.

size	tree
Input Database	a b e -1 c b -1 -1 -1 d -1 b a c -1 a e -1 d -1 -1 -1 a b c -1 b f -1 -1 c e -1 -1 -1 a b e -1 c c -1 -1 -1 d -1
Minsup	2
RE_{scope}	$((bd)\|(be)\|b))*$
$RE_{central}$	$(*)bc$

TABLE I: Mining Input

Pattern	Result	Explanation
b c -1 a -1 -1	discarded	pattern generated but infrequent
a b c -1 b -1 -1 -1	pruned	violate scope constraint
b e -1 c b -1 -1	pruned	violate central pattern constraint
b c -1 e -1 -1	accepted	

TABLE II: Mining Results (both filtered out and qualified)

V. ALGORITHM FRAMEWORK

A. Canonical Form of Tree Representation

In consideration of equivalent representations of an unordered tree, we adopted the canonical form of the unordered tree representation introduced in [7].

Given a node v_i in a tree, its depth-label pair is denoted as $dl(v_i)=(depth(v_i), label(v_i))$ or simply $dl(i)=(d_i, l_i)$ when the context is clear. For two tree nodes v_i and v_j, we say $dl(i) < dl(j)$ if either (i) $d_i > d_j$ or (ii) $l_i < l_j$ if $d_i = d_j$. When $l_i=l_j$ and $d_i=d_j$, we say $dl(i)=dl(j)$. Furthermore, given an ordered k-tree t, its depth label sequence, denoted as $dls(t)$, is a concatenation of the depth label pairs of all the nodes of the tree visited by the preorder left-to-right depth-first tree traversal: $dls(t)=dl(1), dl(2)\ldots, dl(k)$. Let t_1 and t_2 be two labeled ordered trees, t_1 is called a prefix subtree of t_2 if $dls(t_1)$ is a prefix of $dls(t_2)$. We say $dls(t_1) > dls(t_2)$, if either t_1 is a prefix of t_2 or the two trees differ at the leftmost position i by having $(d_i, l_i)_{t_1} > (d_i, l_i)_{t_2}$, where the subscripts refer to the two trees respectively. A unordered tree t is in its canonical form if no equivalent ordered tree t' exists with $dls(t') < dls(t)$.

B. Equivalence Class

Two unordered k-trees are said to be in the same equivalence class, if their canonical forms share exactly the same $k{-}1$ dls prefix. Thus, any two trees in the same equivalence class differ only at the rightmost positions.

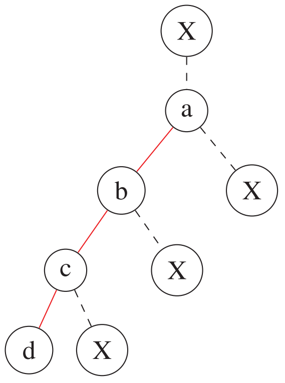
C. Candidate Pattern Generation

The proposed algorithm first generates $k{+}1$ candidates from all the frequent k trees, then uses pruning methods in subsection V-D to prune candidates that do not obey constraints, and uses detection and counting algorithms in subsection V-E to find the frequent $k{+}1$ trees. In this subsection, we focus on the candidate generation algorithm.

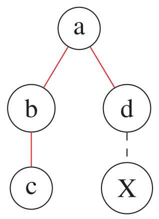
Classical right-most based candidate generation algorithms, such as [7], [5], are designed for Apriori based candidate patterns. However, in this approach, the frequent subtrees do not obey the Apriori law, in the sense that a subtree of a frequent tree can be non-frequent, for the user-defined central pattern constraints. To solve this problem, different categories of pattern trees are handled separately, as shown in the following text.

1) Sequence based generation: In this category, the frequent k tree, that is used to generate $k+1$ candidates, is assumed a valid sequence (A tree with only one leaf). Its generation of $k+1$ trees can be further classified into two sub-cases. **1.1. Upward growth.** Attach the k tree (sequence) a new root label. **1.2. Horizontal growth.** For each node n except the leaf node, attach a new frequent label as its child.

Fig.6 (a) presents an example of sequence based generation. The new node x is attached to a frequent 4-tree (sequence).



(a) Sequence based Growth



(b) Leg attachment

Fig. 6: Subtrees that require constraints checking (a) Sequence based growth (b) Leg attachment in tree based growth

2) Tree based generation: In this case, the candidate generation is operated on all the frequent k trees that have at least two leaves. The growth methods are further classified into two sub-categories similarly.

2.1. Horizontal join. If two trees t_{k_1} and t_{k_2} (possibly the same) are in the same equivalence class, that they share the same $k{-}1$ depth-label list and only differ at the right most node, they are eligible for further joining. It is noted that despite they share the same $k-1$ prefix, their topological structures could be the same or different, as shown in Fig. 7 and Fig. 8. Therefore, the following two cases should be considered.

- case 1: Two trees t_{k_1} and t_{k_2} have the same topology, as shown in Fig. 7. As a result, the last nodes of two trees are in the same height. Assuming their depth-label lists of are:

$$t_{k_1} : (d_1, l_1) \dots (d_{k-1}, l_{k-1}), (d_k^{t_1}, l_k^{t_1})$$

$$t_{k_2} : (d_1, l_1) \dots (d_{k-1}, l_{k-1}), (d_k^{t_2}, l_k^{t_2}).$$

  Without losing of generality, it is assumed that $(d_k^{t_1}, l_k^{t_1}) < (d_k^{t_2}, l_k^{t_2})$, then the resulting $k+1$ tree is:

$$(d_1, l_1) \dots (d_{k-1}, l_{k-1}), (d_k^{t_1}, l_k^{t_1}), (d_k^{t_2}, l_k^{t_2}).$$

  An important special case of the same-topology tree-based generation is the self-joining of each k-tree.
- Case 2: Two trees t_{k_1} and t_{k_2} have different typologies, as shown in Fig. 8. As their $k-1$ prefix are the same, the depth of the last nodes of two trees must be different. In the newly generated tree, the last two leaves from the two joined trees will not have any parent-child or sibling relationship. If $d_k^{t_1} < d_k^{k_2}$, then the generated tree is

$$(d_1, l_1) \dots (d_{k-1}, l_{k-1}), (d_k^{t_1}, l_k^{t_1}), (d_k^{t_2}, l_k^{t_2}).$$

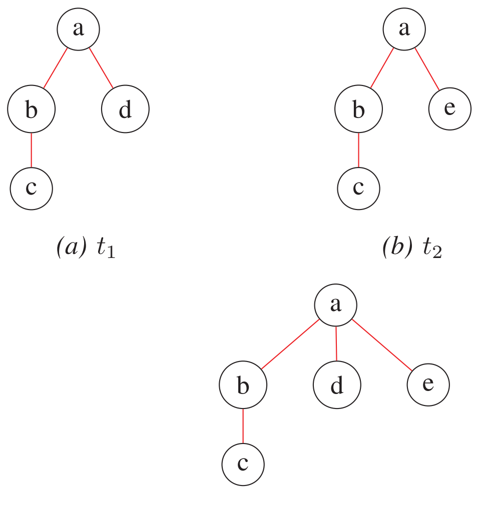


Fig. 7: Subtrees that require no constraints check: horizontal join with the same tree topology

2.2 Leg Attachment. To grow a tree vertically from each frequent k-subtree, frequent labels are attached to its rightmost leaf to get $k+1$ candidates. As shown in Fig.6 (b), the new node x is attached to the rightmost node d of the frequent 4-tree.

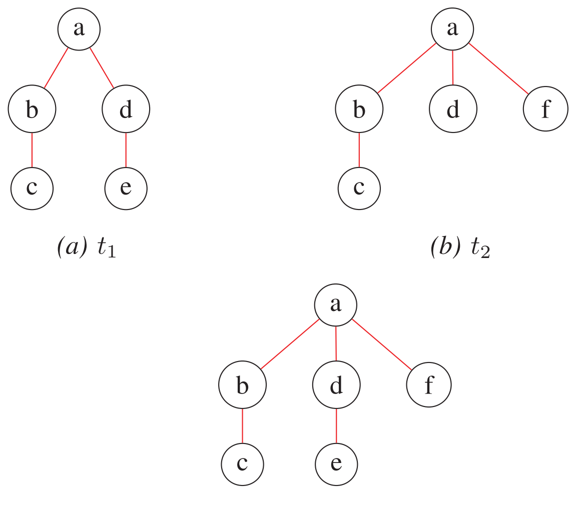


Fig. 8: Subtrees that require no constraints check: Horizontal join with different tree topology

D. Pattern Pruning

In subsection V-C, the algorithms to generate $k+1$ candidate trees from frequent k trees are presented. As the newly generated patterns might not obey user defined constraints or not be in their canonical forms, methods to prune those illegal $k+1$ patterns are explained in this subsection.

1) Constraints based pruning: The user-defined sequential constraints, that the RE_{scope} and $RE_{central}$, are used to exclude $k+1$ candidates with uninterested items. As all the frequent k-trees obey both the scope and the central pattern constraints, only candidates that contain the newly added sequences need to be checked.

For a $k+1$ candidate t_{k+1}, if it is generated by horizontal join from two frequent k trees, as shown in Fig. 7 and Fig. 8, no constraints based pruning are performed, as there are no newly generated sequences. If a t_{k+1} is generated by the sequence based upward growth, the $RE_{central}$ needs to be checked, as shown in Fig. 6(a). If t_{k+1} is generated by the sequence based horizontal growth or the tree based leg attachment, the constraints RE_{scope} is used to check t_{k+1}, as shown in Fig. 6.

To perform constraints based pruning, a straightforward method is to extract newly generated root-leaf sequences and check them with the constraints. To improve the computational efficiency, ideas presented in [1] are adopted. As shown in Fig. 3, the REs are transformed into DFAs, where node labels are the transitions of the DFAs. For each node n in a k-tree that is possible to generate a new sequential pattern, such as the nodes on the central patterns and the right most nodes, as shown in Fig. 6, its associated states in the DFA is stored. Once a new label l is attached to n to generate a $k+1$ candidate, whether there is a feasible transition from the current state defined by l is checked. If the transition is defined, then this newly added label l obeys the constraints, and the states information is updated; otherwise, this $k+1$

pattern tree is pruned. For a comprehensive explanation of this algorithm, please refer to [1].

For example, suppose the right most node d in Fig.6(b) holds the state $s2$ in the RE shown in Fig. 3. If the newly added node is d, then this new candidate is pruned, for there is no transition defined for label d from state $s2$; if the newly added label is c, then state of the right most node (c currently) is updated to state $s3$, and this newly generated candidate is used to perform support counting to check its frequency, as shown in subsection V-E.

2) Canonical form based pruning: The $k{+}1$ candidates generated from frequent k trees might not be in their canonical forms. Thus, methods in [7] is used to test whether or not it is in its canonical form. If the tree is already in its canonical form, it can be kept as a candidate for further checking; otherwise, the subtree can be safely discarded, as explained in [7].

E. Embedded Detection

Our embedded detection algorithm is inspired by the scope-list representation in [6]. For a tree node n_x in a data tree t, let n_y be the right-most node of the downward subtree rooted at n_x. The scope of n_x is $s(n_x)$=$[x, y]$, where x and y are the positions of n_x and n_y in pre-order traversal. For any two nodes n_1, n_2 in the same tree, $s(n_1) \subset s(n_2)$ iff $x_1 > x_2$ and $y_1 < y_2$, $s(n_1) < s(n_2)$ iff $y_1 < x_2$.

For each sequence s_{sub} (tree that has one leaf) in frequent k-trees, its scope list is a list of the triplets (t, m, S_{list}), where t is the data tree id that s is embedded in, m is the matching list of s in tree t, such that $m(i)$ is the depth-first position of the ith node in t. $S_{list}(i)$ is the scope of $m(i)$. For each tree t_{sub} (tree that has at least two leaves) in frequent k patterns, its scope list is a list of the triplets (t, m, s), where t and m have the same meaning, and s is the scope of the right-most node of t_{sub}.

The following tests are performed to determine the occurrence of a newly generated candidate.

1) Descendant Test: Suppose a $k{+}1$ candidate t_{k+1} is generated from a frequent k-tree t_k with scope list $l(t_k)$, by adding a frequent label n with scope list $l(n)$, to the position n_{pos} in t_k. To check if t_{k+1} is embedded in a data tree t_{data}, it is required to find whether n_{pos} is an ancestor of n in the data tree t_{data}. Formally, it is required to find if there exist elements in the scope-list of t_k and n, such that

1) $t_{id,k} = t_{id,n} = t_{data}$.

2) $s(n) \subset s(n_{pos})$.

If the above conditions are satisfied, the triplet $(t_{data}, m \cup n, s(n))$ is added to the scope list of the $k{+}1$ candidate t_{k+1}.

This test is for $k{+}1$ candidates generated by sequence based growth 1.2 and the leg-attachment in tree based growth (as shown in Fig. 6 (b)). For sequence based growth 1.1, that the upward growth, it is required to test whether the newly added label n is a ancestor of original root of t_k in some tree t_{data}, the test conditions become: 1) $t_{id,k} = t_{id,n} = t_{data}$ and 2) $s(n_{root}) \subset s(n)$.

2) Cousin Test: Suppose a $k + 1$ candidate t_{k+1} is generated from two frequent k trees t_{k_1} and t_{k_2} by horizontal joining, in order to check if t_{k+1} is embedded in a data tree t_{data}, it is required to find if there exists elements in the scope-list of t_{k_1} and t_{k_2}, such that

1)$t_{id,k1} = t_{id,k2} = t_{data}$;

2)$m_{k1} = m_{k2}$; and

3)$s_{rm1} < s_{rm2}$ or $s_{rm1} > s_{rm2}$.

If the above conditions are satisfied, the triplets $(t_{data}, m_{k_2} \cup n_{rm1}, S_{rm2})$ is added to the scope list of $k+1$ candidate t_{k+1}. This test is for $k{+}1$ candidates generated by horizontal joint in tree based growth, as shown in Fig. 7 and Fig. 8.

VI. Experimental Evaluations

The proposed algorithm is implemented in C++ language. The platform for all experiments presented in this section is a Dell computer with 2.7 GHz Intel Core quad processor and 16 GB DDR4 RAM, running windows 10 operating system.

The proposed algorithm is tested on synthetic datasets. The synthetic datasets are generated by the generator in [6]. The generator supports the following parameters: 1) the number of node labels (N); 2) the size of the master tree (M); 3) the maximum fan-out of the master tree (F); 4) the maximum depth of the master tree (D); and (5) the total number of trees in the dataset (T).

In the first experiment, the performance of the proposed algorithm on datasets of different sizes is evaluated. The parameters are set as follows: N=20, M=10000, D=20, F=10, and the sizes of the datasets are 1000, 2000, 3000 and 4000, respectively. The minimum threshold $minsup$ is 3% and the same RE constraints are used for all the four tests. We run the algorithm five times and take the average value as the final performance results. Fig. 9 (a) shows the running time of the algorithm on each dataset. The running time is approximately linear on the size of the dataset. For larger datasets, the time consumption mainly results from the embedded detection phase of the algorithm. The number of discovered patterns are approximately stable since our constraints are fixed.

Figure 9 (b) depicts the space needed when running the algorithm on the datasets, where space refers to the heap peak memory (measured in Mb) used by the program. From the figure, the used space increases approximately linearly on the dataset size.

To study how the minimum support value $minsup$ affects the performance, the algorithm is run on the same dataset with the $minsup$ value gradually decreased. Other parameters, like size of data set and constraints, are fixed. The size of the dataset is 2000 in this experiment. Fig. 10 shows the impact of $minsup$ value on the time performance. From the figure, as the $minsup$ decreases, more subtrees qualified to be frequent, which subsequently leads to longer running time. On the contrary, under large $minsup$ value, only a small amount of subtrees are frequent.

In the next experiment, the impact of changing the number of node labels, the parameter N, on the performance of mining algorithm is investigated. The parameters are set to

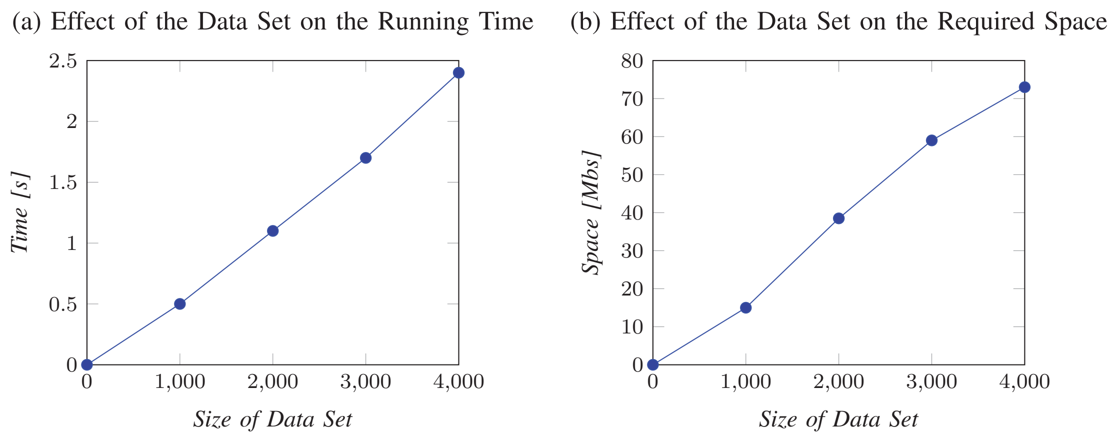


Fig. 9: Effect of dataset sizes on the running time and the required space

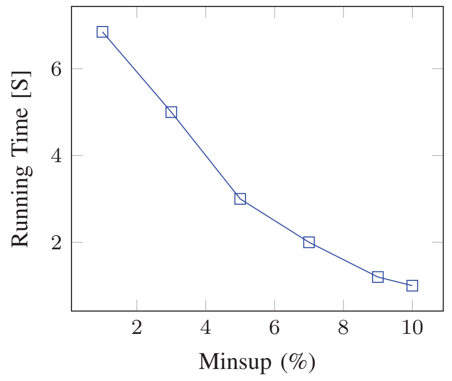


Fig. 10: Effect of the minsup value on the running time

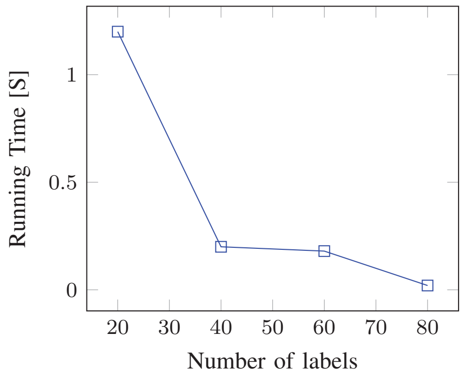


Fig. 11: Effect of the number of labels on the running time

be: M=10000, D=20, F=10, and N from 20 to 80. The total running time is shown in Fig. 11. For our constraints and the $minsup$ value is static, the introduction of new labels will result in many unqualified subtrees on our constraints. Thus, the number of frequent subtrees decreases in this experiment, which leads to the drop of running time.

VII. Conclusion

We have described a new unordered tree mining problem that allows users to specify scope and central constraints. We also present an algorithmic solution, which consists of candidate pattern generation, pattern pruning, embedded detection and support counting, to mine frequent subtrees from a given dataset and constraints. In the future, we plan to improve the algorithm and optimize its implementation. We would also like to extend the algorithms to handle more complicated constraints.

References

[1] M. Garofalakis, R. Rastogi and K. Shim, "Mining sequential patterns with regular expression constraints," *IEEE Transactions on Knowledge and Data Engineering*, 2002, pp. 530-552.

[2] L. Hua, T. L. Wang, X. Ji, A. Malhotra, M. Khaladkar, B. A. Shapiro and K. Zhang, "A method for discovering common patterns from two RNA secondary structures and its application to structural repeat detection," *Journal of bioinformatics and computational biology*, 2012.

[3] M. L. Lee, L. H. Yang, W. Hsu and X. Yang, "XClust: clustering XML schemas for effective integration," *ACM Proceedings of the international conference on Information and knowledge management*, 2002, pp. 292-299.

[4] D. Shasha, T. L. Wang and S. Zhang, "Unordered tree mining with applications to phylogeny", *IEEE International Conference on Data Engineering*, 2004, pp. 708-719.

[5] M. J. Zaki, "Efficiently mining frequent trees in a forest: Algorithms and applications," *IEEE Transactions on Knowledge and Data Engineering*, 2005, pp. 1021-1035.

[6] M. J. Zaki, "Efficiently mining frequent embedded unordered trees," *Fundamenta Informaticae*, 2005, pp. 33-52.

[7] S. Zhang, Z. Du and T. L. Wang, "New Techniques for Mining Frequent Patterns in Unordered Trees," *IEEE Transactions on Cybernetics*, 2015, pp. 1113-1125.

[8] R. Srikant, R. Agrawal, "Mining sequential patterns: generalizations and performance improvements," in *Proceedings of the 5th International Conference on Extending Database Technology*, vol. 1057, Springer, Berlin, 1996, pp. 1-17.

[9] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, B. Shi, "Efficient pattern-growth methods for frequent tree pattern mining," *Pacific-Asia conference on knowledge discovery and data mining*, Springer Berlin Heidelberg, 2004, pp. 441-451.

[10] D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih, "Exact and approximate algorithms for unordered tree matching," *IEEE Transactions on Systems, Man, and Cybernetics*, 1994, pp. 668-678.

[11] J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A system for approximate tree matching," *IEEE Transactions on Knowledge and Data Engineering*, 1994, pp. 559-571.

[12] S. Zhang and J. T. L. Wang, "Discovering frequent agreement subtrees from phylogenetic data," *IEEE Transactions on Knowledge and Data Engineering*, 2008, pp. 68-82.