Proceedings of the 2007 IEEE Symposium on
Computational Intelligence and Data Mining (CIDM 2007)

Mining Maximal Embedded Unordered Tree Patterns

Mostafa Haghir Chehreghani[1], Masoud Rahgozar[2], Caro Lucas[3] and Morteza Haghir Chehreghani[4]

***Abstract*— Mining frequent tree patterns has many practical applications in areas such as XML document mining, web mining, bioinformatics, network routing and so on. Most of the previous works used an apriori-based approach for candidate generation and frequency counting in their algorithms. In these approaches the state space grows exponentially since many unreal candidates are generated, especially when there are lots of large patterns among the data. To tackle these problems, we propose *TDU*, a Top-Down approach for mining all maximal, labeled, Unordered, and embedded subtrees from a collection of tree-structured data. We would evaluate the effectiveness of the *TDU* algorithm in comparison to the previous works.**

I. Introduction

Mining frequent tree patterns is very useful in domains like XML document mining, web mining, bioinformatics, network routing and so on. Recently, many algorithms have been proposed to find frequent tree patterns in a collection of tree-structured data. In [5] Feng et al. initiated an XML-enabled association rule template. They continued their work by presenting templates for XML-enabled association rule mining [6].

In [7] Zaki presented *TREEMINER* to mine embedded ordered frequent tree patterns. He used an efficient data structure called *scope-list* for frequency counting and proposed rightmost extension to generate non-redundant candidates. Later, by proposing the *SLEUTH* algorithm, he extended his work to mine embedded unordered tree patterns [8]. In [13] Asai et al. independently proposed the rightmost candidate generation. They developed *FreqT* for mining frequent induced ordered tree patterns. Chi et al. in [15] proposed *FreeTreeMiner* for mining induced unordered free trees.

In [16] Chi et al. proposed *CMTreeMiner* for mining both *closed* and *maximal* frequent subtrees in a database of rooted unordered trees. This algorithm traverse an enumeration tree that systematically enumerates all subtrees, and use an *enumeration DAG* to prune the branches of the enumeration tree that do not correspond to *closed* or *maximal* frequent subtrees. Recently, *XSpanner*, a pattern growth-based method, has been proposed in [1] for mining embedded ordered subtrees.

For finding unordered frequent tree patterns, proposed algorithms use *canonical form* and extend only candidates that are in canonical form. A canonical form is a unique way to represent a labeled tree. In [3], [4], Luccio et al. defined *sorted pre-order string* method. This method for a rooted *unordered tree* is defined as the lexicographically smallest one among those pre-order strings for the *rooted ordered trees* that can be obtained from the *rooted unordered tree*. To determine the lexicographical order of the string encodings, a total order on the alphabet of vertex labels is defined. Later, Asai et al. [12], Nijssen et al. [11], and Chi et al. [2], [15] independently defined similar canonical representations.

Most of the previous researches on mining frequent tree patterns use *apriori* or *anti-antimonotone* property for efficient candidate generation and frequency counting. This property says that the frequency of a superpattern is less than or equal to the frequency of all of its subpatterns. This property considers only a known frequent pattern for extension and as a result limits the candidate's lattice. *Apriori-based* algorithms show good performance with sparse data sets, where the frequent patterns are very short.

However, as [10] showed, this property has less efficiency when data are dense and there are a lot of large patterns in data or the minimum support is quite low. For example if there are 10^3 frequent 1-subtree, *apriori-based* approaches will need to generate 10^6 2-subtrees and check their frequencies. Many of these candidates are unreal and there are no instances of them in input trees.

To solve these problems, we propose the *TDU* algorithm, a Top-Down approach for mining Unordered maximal tree patterns. *TDU* begins by constructing a special representation from tree-structure data, called *IRTree*. All frequent tree patterns are subtree of *IRTree*. Moreover *IRTree* defines the canonical ordered form for *unordered trees* and therefore makes the 'canonical test' unnecessary. Then *TDU* finds the set of all maximal patterns by fragmenting *IRTree*.

A performance study has been conducted to compare the performance of *TDU* with an *apriori*-based algorithm, *SLEUTH* [8]. Our study shows that when the dataset is dense or the frequent patterns are large, the *TDU* algorithm outperforms the *SLEUTH* algorithm.

The rest of this paper is organized as follows. In section 2, the tree mining problem statement and required definitions

[1] Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email m.haghir@ece.ut.ac.ir.
[2] Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email rahgozar@ut.ac.ir.
3 Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email lucas@ipm.ir.
[4] Faculty of CE, Sharif University, Tehran, Iran, email haghir@ce.sharif.edu.

are given. Section 3 describes the *IRTree* representation. In section 4, we provide an efficient and systematic approach for candidate generation. Section 5 is dedicated to our frequency counting method. Section 6 describes our proposed algorithm; *TDU*. We empirically evaluate the effectiveness of the algorithm in section 7 and the paper is concluded in section 8.

II. Problem Definition and Statement Review Stage

To explain the problem of mining frequent subtrees in a collection of trees we provide the following definitions:

Rooted labeled tree [9]: A rooted labeled tree $T=(V,E)$, is a *directed*, *acyclic* and *connected* graph with $V=\{0,1,...n\}$ as the set of vertices and $E=\{(x,y)\,|\,x,y\in V\}$ as the set of edges. One distinguished vertex $r\in V$ is selected the root, so that for all $x\in V$, there is a unique path from *r* to *x*. Further, $l:V\rightarrow L$ is a labeling function mapping vertices to a set of labels $L=\{l_1,l_2,...\}$.

Induced subtree [14]: For a tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an induced subtree of T, if and only if (1) $V'\subseteq V$, (2) $E'\subseteq E$, (3) the labeling of V' and E' is preserved in T', (4) if defined for rooted ordered trees, the left-to-right ordering among the siblings in T' should be a subordering of the corresponding vertices in T.

Embedded subtree [14]: For a rooted unordered tree *T* with vertex set *V*, edge set *E*, and no labels on the edges, a tree T' with vertex set V', edge set E', and no labels on the edges, is an embedded subtree of *T* if and only if (1) $V'\subseteq V$, (2) the labeling of the nodes of V' in *T* is preserved in T' and (3) $(v_1,v_2)\in E'$, where v_1 is the parent of v_2 in T', only if v_1 is an ancestor of v_2 in *T*. If *T* and T' are rooted ordered trees, then for T' to be an embedded subtree of *T*, a fourth condition must hold: (4) for $(v_1,v_2)\in V'$, $preorder(v_1)<preorder(v_2)$ in T' if and only if $preorder(v_1)<preorder(v_2)$ in *T*, where the *preorder* of a node is its index in the tree according to the preorder traversal.

Support & weighted support [9]: Let $\delta_T(S)$ indicate the number of occurrences of the subtree *S* in a tree *T*. Let d_T be an indicator variable, with $d_T=1$ if $\delta_T(S)>0$ and $d_T=0$ if $\delta_T(S)=0$. Let *D* denote a database of trees. The support of a subtree *S* in the database is defined as $\sigma(S)=\sum_{T\in D}d_T(S)$. The weighted support of *S* is defined as $\sigma_w(S)=\sum_{T\in D}\delta_T(S)$. Support is given as a percentage of the total number of trees in *D*.

Frequent subtree: An *l*-subtree *S*, which is a subtree with *l* nodes, is frequent if its (weighted) support is more than or equal to a user-specified minimum (weighted) support value.

Maximal frequent subtree: A maximal frequent subtree is a frequent subtree which none of its proper supertrees are frequent.

The problem of mining frequent tree patterns in a database of tree-structured data is to find all of the frequent k-subtrees, $1\le k\le M$ where M is the maximum number of nodes in data. The desired type of frequent subtree patterns which is aimed in the mining process can differ based on the kind of application. In this paper, our goal is to mine all maximal, labeled, unordered, and embedded subtrees in a forest, by proposing the *TDU* algorithm.

III. Intermediate Representation Tree

In this section, we propose the *Intermediate Representation Tree* or *IRTree* in short, which is a novel and compact representation of input trees. *IRTree* is a *rooted, ordered tree* that is constructed from some *rooted unordered trees* and has the following properties:

- The set of its nodes is exactly equal to the set of the frequent nodes in input trees,
- Two frequent nodes have ascendant-descendent relation in *IRTree*, iff this relation does exist in at least one of the input trees,
- There is no repeated path which is started from the root and its length is greater than zero.

As an example, consider trees displayed in figure 1. Tree (a) is an *IRTree* but tree (b) is not. In tree (b) the path “1—2” is repeated two times.

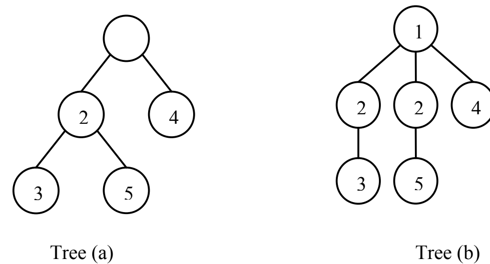

Figure 1: tree (a) is an *IRTree* but tree b is not.

Constructing IRTree. For simplicity (and without loss of generality), we assume that the roots in all input trees have the same label. For each input tree, at first the non-frequent nodes are eliminated and the resulted tree is called *t*. Then, for each path *r* from *t*’s root to a leaf:

- If *IRTree* has no node, *r* is added to it.
- If path *r* is the prefix of one existing path in *IRTree*, do nothing.
- In other cases, one of *IRTree*’s paths that has the longest common prefix with path *r* is selected. We refer to this path as *q*. The nodes of *r* that are placed after the common prefix are added to the last node of the common prefix in path *q* as its right most subpath.

Eliminating non-frequent nodes from the tree *T* is relatively straightforward: after deleting the node *a*, the parent of node *a*’s children would be the parent of node *a*. If

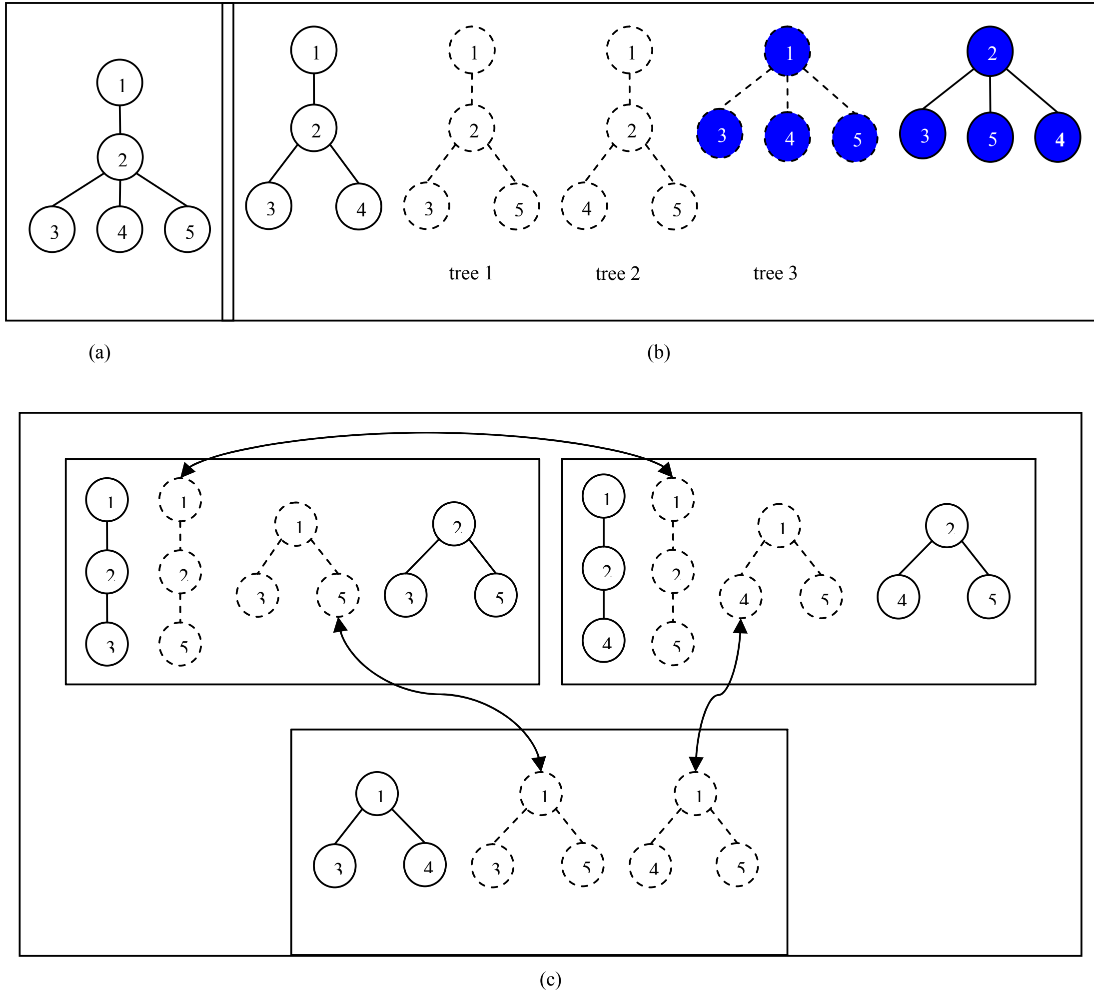

Figure 2: An example of candidate generation

node *a* has no parent node (node *a* is the root of tree) and the number of *a*'childern is more than one, *T* would be partitioned to some subtrees.

It must be noted that we use *IRTree* only for *'Candidate Generation'* and not for *'Frequency Counting'*.

Theorem 1: All of the frequent tree patterns in a collection of trees are subtrees of their *IRTree* representation.

Proof. Omitted due to lack of space. ■

IV. CANDIDATE GENERATION

Consider a non-frequent tree T with k nodes. Since it is non-frequent, it must be fragmented into some trees with (k-1) nodes. This is done by eliminating each of nodes of T. Each of generated smaller subtrees can be frequent or non-frequent. Non-frequent subtrees must be fragmented again. When fragmenting non-frequent trees, two possible problems can occur. 1) Some generated candidates could be subtrees of a frequent tree. 2) Some candidates could be generated more than once. For example, consider figure 2. In this figure, the tree 2-(a) is non-frequent and must be fragmented. By eliminating each of its nodes, as has been shown in figure 2-(b), five smaller trees have been generated. From there, assume that dashed trees are non-frequent and therefore must be fragmented further. After doing this, the dashed trees in figure 2-(c) are generated two times and other trees are subtrees of the frequent *supertrees*.

The reason is that in figure 2-(c) each dashed tree is the subtree of two non-frequent supertrees, but other trees are subtrees of one frequent supertree and one non-frequent supertree. Our goal is to generate only the dashed trees exactly once.

To solve the first problem, we must generate only the subtrees which are common between two non-frequent trees. Solving the second problem needs to define an order on the trees when computing their common subtrees. For example, one can generate the common subtree between non-frequent tree *t* and each non-frequent tree which is generated after *t* (not between any two arbitrary non-frequent trees).

By using these solutions, the trees presented in figure

3-(a) are generated from non-frequent trees showed in figure 2-(b). Each of these trees can be frequent or non-frequent. Assume that all of them are non-frequent. The common subtrees between (tree 1 and tree 2), (tree 1 and tree 3) and (tree 2 and tree 3) have been shown in figure 3-(b). The Subtree (1—5) have been generated for three times. To tackle this problem, we introduce the *equivalence class* concept.

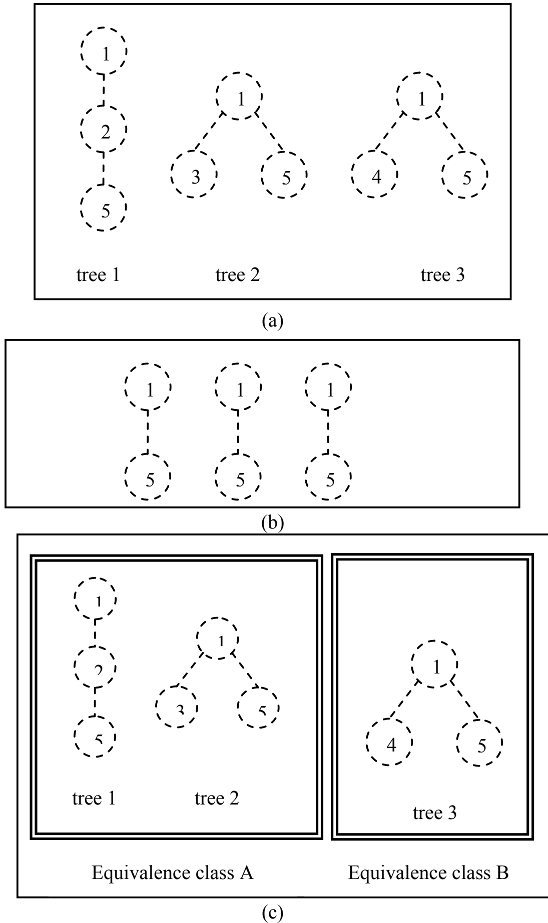

Figure 3: Candidate generation considering equivalence classes.

Equivalence class. Every two trees belong to the same *equivalence class*, iff they have the same *first-supertree*. Each subtree with length k is the common part of two supertrees with length k+1. The first supertree is called *first-supertree*. For example, in figure 3-(c), these trees placed in a block belong to the same equivalence class. The *first-supertree* of equivalence class A is the tree 1 from figure 2-(b) and the *first-supertree* of equivalence class B is the tree 2 from figure 2-(b).

***Theorem 2 (tree fragmentation approach)*:** Assume that tree T is non-frequent and therefore, must be fragmented. By eliminating each of its nodes a smaller candidate is generated. Assume that between these candidates, n tree(s) are non-frequent. If $n = 1$, this candidate tree is ignored. Otherwise, the common subtree of each non-frequent tree t and non-frequent trees generated after t is determined. For generated subtrees that: 1) are non-frequent and 2) belong to the same equivalence class, the above 'If' is repeated until all equivalence classes have at most one non-frequent member.

Proof. Omitted due to lack of space. ■

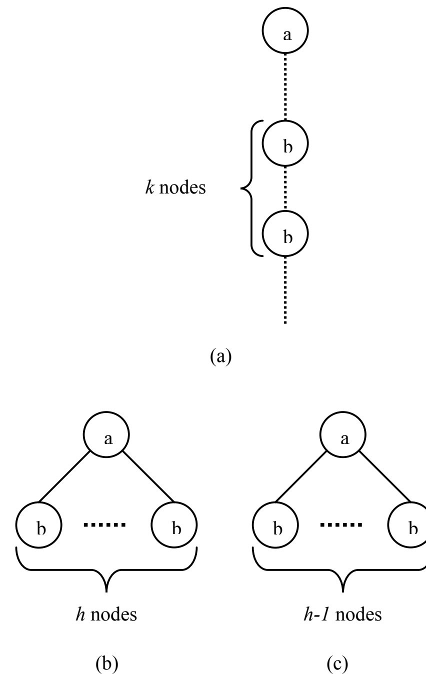

Figure 4. Two special situations which in some candidates are generated more than once.

Note that when deleting the root of tree T with k nodes, if its root has more than one child, T can be partitioned to some subtrees with the size less than (k-1). At this state, each partition is the subtree of a previously generated candidate and therefore no new candidate is generated. For example, consider two colored trees in figure 2-(b). Their common subtree looses their roots and their roots have more than one child. Thus we do not generate any candidate for them in figure 2-(c).

Theorem 3: Tree fragmentation approach generates each subtree of IRTree exactly once.

Proof. Omitted due to lack of space. ■

More precisely, when a tree is non-frequent and each of its nodes must be deleted, there are two situations in which the redundant candidates can be generated. As showed in figure 4-(a), the first case occurs when in one path of the non-frequent tree there are $\{k \mid k \geq 2\}$ nodes with the same labels and all of these k nodes (except the last node) have the same children. In this case, to generate new candidates non-redundantly, only one of these k nodes is deleted. In the next case, as reflected in figure 4-

(b), the non-frequent tree has one root and $\{h \mid h \geq 2\}$ leaves with the same label. In this case, only one candidate is generated. This candidate is presented in figure 4-(c).

V. FREQUENCY COUNTING

In vertical approaches for frequency counting, the main step is to determine the kind of relation (ascendant-descendant or sibling) between two nodes. To determine the kind of relation, we use a preorder-traverse-based approach. In this traverse, the order of all the children of a node is greater than or equal to the order of that node and less than or equal to the order of its rightmost child. For the first time, Zaki in [7] used this property to count the frequency of subtrees efficiently and presented a special data structure, *scope-list*. Recently, a different preorder-traversal-based method was proposed in [1].

We employ Zaki's method, by some alterations. We present the *occ-list* data structure that expresses the list of occurrences of a subtree in all of input trees. Each occurrence consists of three elements. The first element is called *tid* and is a tree *Id* in which the tree occurs. The second element is *NodeId* and shows the preorder number of the last node of the tree. The third element is *scope* and shows the preorder number of the rightmost child of the last node of the tree.

Based on scope-list data structure, Zaki has defined In-scope and Out-Scope tests in [7] for determining the kind of relation between two nodes.

To find whether there exists the *ascendant-descendant* relation between two nodes x and y or not, we act as follows. If there is an occurrence $occ1 \in occ-list\ of\ \ node\ x$ and an occurrence $occ2 \in occ-list\ of\ \ node\ y$ such that 1) the tid of *occ1* is equal to the *tid* of *occ2*, 2) the *scope* (or *NodeId*) of *occ2* is between the *NodeId* and *scope* of *occ1*, there exists the ascendant-descendant relation between the two nodes *x* and *y* in occurrence *occ2*. We refer to this test as *ascendant-descendant* test.

To find whether or not there exists the *sibling* relation between two nodes *x* and *y*, we act as follows. If there is an occurrence $occ1 \in occ-list\ of\ node\ x$ and an occurrence $occ2 \in occ-list\ of\ node\ y$ such that 1) the *tid* of *occ1* is equal to the *tid* of occ2, 2) the *scope* (or *NodeId*) of *occ2* is greater than the *scope* of *occ1* or less than the *NodeId* of *occ1*, there exists the *sibling* relation between the two nodes *x* and *y* in occurrence *occ1*. We refer to this test as *sibling* test.

These tests have been completely discussed in [7], with some differences. For more details and illustrative examples, the interested reader can refer to [7].

Zaki's method for frequency counting has a weakness. For example, in figure 5, assume that there exists ascendant-descendant relation between two nodes '1' and '4' in *n* occurrences. And, assume that from these *n* occurrences, in $\{m \mid m \leq n\}$ occurrences, the node '5' is the descendant of the node '4'. In these conditions, in $\{k \mid k \geq m\}$ occurrences, the node '5' would be the descendant of node '1'. Therefore, if $m \geq minimum\text{-}support$, it is not necessary to perform the ascendant-descendant test between nodes '1' and '5'.

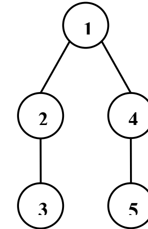

Figure 5. Unnecessary ascendant-descendant tests done by *TreeMiner*

It seems that Zaki's method does not consider this property and performs some unnecessary ascendant-descendant tests. Also, consider a frequent path which has l nodes. Zaki's method performs $\sum_{i=0}^{i-1} i = l(l-1)/2$ ascendant-descendant tests for this path, while only $l-1$ ascendant-descendant tests are required. Note that the sibling relation does not have such a property. Our algorithm avoids these unnecessary tests by constructing candidates from top to down.

```
TDU (a collection of tree-structured data)
1. Begin
2.      Compute F1; // F1 is the set of all frequent nodes
3.      Construct IRTree from the input data;
4.      Fragment (IRTree)
5. End  // of TDU
```

Figure 6. The high level pseudo-code of *TDU* algorithm

VI. *TDU* ALGORITHM

In this section, we propose the *TDU* algorithm for finding frequent, unordered, embedded and maximal tree patterns from a collection of tree-structured data. The high level pseudo-code of this algorithm is presented in figure 6. The *TDU* algorithm begins by computing the set of all frequent nodes. It is done simply by incrementing the count of each node i in a 1-dimensional array. Then it constructs the *IRTree* according to the proposed algorithm discussed in section 3. *IRTree* is always non-frequent, and therefore must be fragmented. This fragmentation is done based on *tree fragmentation approach* proposed in section 4. This approach extracts the set of all maximal subtrees from *IRTree*. To do this, the *tree fragmentation approach* needs a frequency counting method explained bellow.

Frequency counting method. Assume that tree *T* is

frequent to node *i*-1 in preorder traverse. This method reads node *i* and performs the *ascendant-descendant* test between this node and only its direct parent (not all of its ascendants, and as a result avoids unnecessary *ascendant-descendant* tests). After performing this test, if the number of occurrences in *occ-list* of node *i* is less than *min-sup*, the tree *T* would be non-frequent. Otherwise, this method performs the *sibling* test. This test is done between nodes *i* and all of its siblings in scanned part of tree *T*. If for each sibling the number of these occurrences is less than the *min-sup*, this tree would not be frequent.

The following lemma can help to determine the *siblings* of a node.

Lemma: when traversing a tree in preorder, whenever a node is reached, it is pushed on the stack and whenever a backtrack happens (move from a node to its parent) the stack is popped, then the content of the stack would show the rightmost path of the scanned part of the tree and the popped elements would show the sibling(s) of the top element of the stack.

Proof. Directly from the preorder traverse of trees. ■

VII. Experimental Results

We performed extensive experiments to evaluate the performance of the *TDU* algorithm using both synthetic data and data from real applications. Due to the space limitation, here we only report the results for a synthetic dataset. We did our experiments on a 3GHz Intel Pentium IV PC with a 1GB main memory, running windows XP operating system. All algorithms were implemented in C++.

To the best of our knowledge, at the time of writing this paper, there is no algorithm for mining unordered, embedded and maximal tree patterns. We think that the best algorithm for our comparisons is *SLEUTH* [8], which is proposed by Zaki for mining unordered embedded frequent tree patterns. To the best of our knowledge, this algorithm is the most efficient algorithm in the context of mining unordered and embedded tree patterns.

We implemented the tree generator program described in [1]. In this program there are 8 parameters for adjustment: number of the labels $|S|$, the probability threshold of one node in the tree to generate children or not p, number of the basic pattern trees (BPT)$|L|$, average height of the BPT$|I|$, the maximum fanout of nodes in the BPT $|C|$, the data size of synthetic trees$|N|$, the maximum height of synthetic trees $|H|$ and the maximum fanout of nodes $|F|$ in the synthetic trees.

In our synthetic dataset, we specified a large fanout and depth in combination with a low number of distinct labels: number of distinct labels is 100, number of trees (transactions) is 1000, total number of nodes is 100000, the average fanout is equal to 10 and maximum depth of trees is equal to 20.

Figure 7 shows the performance of *TDU* on synthetic dataset for different values of minimum support, and compares the run time against the *SLEUTH* algorithm. As reflected in Figure 7-(a), number of subtrees increases by decreasing the minimum support. Due to the large differences in number of frequent subtrees, we show this chart in a logarithmic scale. *TDU* scales linearly in the minimum support, but *SLEUTH* does not.

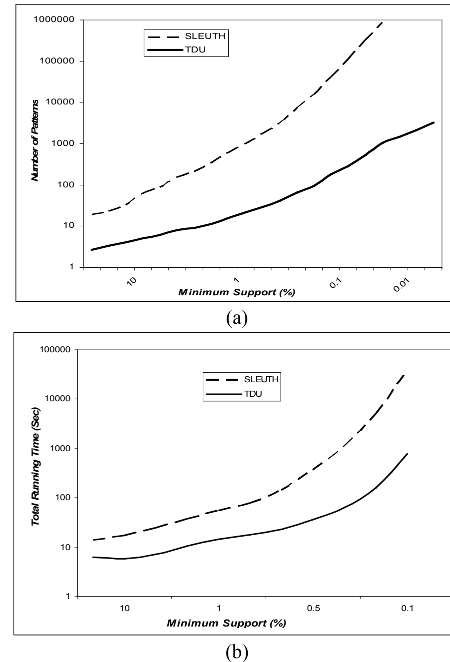

Figure 7: Performance evaluation on synthetic dataset

Figure 7-(b) shows the effect of minimum support on running time. Efficiency of *TDU* increases when the data have complex and large tree patterns. There are two reasons for this behavior: first, *TDU* constructs the lattice in a top-down manner and as patterns are larger, *TDU* can find them with less tree fragmentations. Second, by increasing the complexity of patterns, the intersection between data increases and therefore the size of *IRTree* reduces. The number of candidates increases exponentially with the size of *IRTree*. In contrast, *SLEUTH* is very efficient for datasets with small tree patterns and as the size of patterns increases; this algorithm suffers from the exponential growth of computation time.

VIII. Conclusion

In this paper, we proposed the *TDU* algorithm, for mining all maximal, labeled, unordered, and embedded subtrees from a collection of tree-structured data. To the best of our knowledge, this algorithm is the first top-down approach and is the only algorithm that mines maximal and embedded tree patterns. *TDU* starts its work by constructing a novel representation of tree-structured

data, named *IRTree*, which contains all embedded frequent patterns. Moreover, *IRTree* defines the canonical ordered form for unordered trees and therefore makes the 'canonical test' unnecessary. Then, *TDU* generates non-redundantly, all embedded real candidate trees by imposing an efficient tree fragmentation approach on *IRTree*.

In general, as experimental results show, when data are dense and there are lots of large and complex frequent tree patterns, the *TDU* algorithm performs better than the *SLEUTH* algorithm. In dense data, the required fragmentations and the size of *IRTree* are reduced. On the other hand, the *SLEUTH* algorithm shows better performance with sparse data sets, where frequent patterns are very short.

REFERENCES

[1] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi, "Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining", Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining, 2004.

[2] D. Cook and L. Holder, "Substructure discovery using minimal description length and background knowledge", *Journal of Artificial Intelligence Research*, 1, 231–255, 1994.

[3] F. Luccio, A. M. Enriquez, P. O. Rieumont and L. Pagli, "Exact Rooted Subtree Matching in Sublinear Time", Technical Report TR-01-14, Universita Di Pisa, 2001.

[4] F. Luccio, A. M. Enriquez, P. O. Rieumont and L. Pagli, "Bottom-up Subtree Isomorphism for Unordered Labeled Trees", Technical Report TR-04-13, Universita Di Pisa, 2004.

[5] L. Feng, T. S. Dillon, H. Weigand and E. Chang, "An XML-Enabled Association Rule Framework." In Proceedings of DEXA 2003, pp 88-97, Prague, Czech Republic, 2003.

[6] L. Feng and T. Dillon, "Mining XML-Enabled Association Rule with Templates." In Proceedings of KDID 04, 2004.

[7] M. J. Zaki, "Efficiently Mining Frequent Trees in a Forest." Proc of the 2002 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'02), July 2002.

[8] M. J. Zaki, "Efficiently Mining Frequent Embedded Unordered Trees." Fundam. Inform. 66(1-2): 33-52, 2005.

[9] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications", in IEEE Transaction on Knowledge and Data Engineering, vol. 17, no. 8, pp. 1021-1035, 2005.

[10] R. J. B. Jr, "Efficiently Mining Long Patterns from Databases", SIGMOD' 98, Seattle, WA, USA, ACM, 1998.

[11] S. Nijssen and J.N. Kok, "Efficient Discovery of Frequent Unordered Trees", Proc. First Int'l Workshop Mining Graphs, Trees, and Sequences, 2003.

[12] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering Frequent Substructures in Large Unordered Trees," Proc. Sixth Int'l Conf. Discovery Science, Oct. 2003.

[13] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa, "Efficient Substructure Discovery from Large Semi-Structured Data" Proc. Second SIAM Int'l Conf. Data Mining, Apr. 2002.

[14] Y. Chi, R. R. Muntz, S. Nijssen, J. N. Kok, "Frequent Subtree Mining - An Overview", Fundam. Inform. 66(1-2): 161-198, 2005.

[15] Y. Chi, Y. Yang, and R.R. Muntz, "Indexing and Mining Free Trees," Proc. Third IEEE Int'l Conf. Data Mining, 2003.

[16] Y. Chi, Y. Yang, Y. Xia and R. R. Muntz, "CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees." PAKDD 2004: 63-73, 2004.