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

UNI3 – efficient algorithm for mining **un**ordered **i**nduced sub**tree**s using TMG candidate generation

Fedja Hadzic[1], Henry Tan[1] and Tharam S. Dillon[1]

[1]*Faculty of Information Technology, University of Technology Sydney, Australia*
E-mail: (fhadzic, henryws, tharam)@it.uts.edu.au

***Abstract* - Semi-structured data sources are increasingly in use today because of their capability of representing information through more complex structures where semantics and relationships of data objects are more easily expressed. Extraction of frequent sub-structures from such data has found important applications in areas such as Bioinformatics, XML mining, Web mining, scientific data management etc. This paper is concerned with the task of mining frequent unordered induced subtrees from a database of rooted ordered labeled subtrees. Our previous work in the area of frequent subtree mining is characterized by the efficient Tree Model Guided (TMG) candidate enumeration, where candidate subtrees conform to the data's underlying tree structure. We apply the same approach to the unordered case, motivated by the fact that in many applications of frequent subtree mining the order among siblings is not considered important. The proposed UNI3 algorithm considers both transaction based and occurrence match support. Synthetic and real world data are used to evaluate the time performance of our approach in comparison to the well known algorithms developed for the same problem.**

***Index Terms*: frequent subtree mining, induced unordered subtrees, tree isomorphism, canonical form**

I. INTRODUCTION

Frequent subtree mining has attracted lots of interest among the data mining community, due to the increasing use of semi-structured data sources for more meaningful knowledge representations. Generally the problem of frequent subtree mining can be stated as: given a tree database Tdb and minimum support threshold (σ), find all subtrees that occur at least σ times in Tdb. Applications in Bioinformatics, XML Mining, scientific data management, increasingly make use of tree mining algorithms for analysis of domain knowledge represented in a tree-structured form. The scope of their application usually depends on the assumptions made about the data structure that the algorithm can be applied to. These assumptions depend upon the domain of interest, and many algorithms have been developed that mine different types of subtrees.

Even though the tree structures underlying semi-structured data sources are ordered, interesting associations or queries are commonly based on unordered trees since the ordering among sibling data objects is not of great importance to the user and is often not available. The focus of this paper is on the problem of extracting all frequent unordered induced subtrees from a database of rooted ordered labeled subtrees (eg. XML), and a few algorithms have been developed and applied to the problem. The Unot algorithm [1] uses a reverse search technique for incremental computation of unordered subtree occurrences. Nijssen and Kok [2] present a bottom-up strategy for determining the frequency of unordered induced subtrees, and argue that the complexity of enumerating unordered trees as opposed to ordered is not much higher. Breadth-first canonical form (BFCF) and depth-first canonical form (DFCF) for labeled rooted unordered trees has been presented in [3]. In the same work the authors proposed two algorithms: RootedTreeMiner, a vertical mining algorithm based upon BFCF and FreeTreeMiner, based on extension of DFCF for discovering labeled free trees. As an extension to the work, HybridTreeMiner [4] is an efficient algorithm that systematically enumerates all subtrees by traversing an enumeration tree which is defined based upon the BFCF for unordered subtrees. SLEUTH [5] is an efficient algorithm for mining frequent embedded unordered subtrees, where frequent patterns are enumerated by unordered scope-list joins via the descendant and cousin tests. Another algorithm for mining frequent embedded unordered subtrees is TreeFinder [6] that uses an Inductive Logic Programming approach, but which in the process can miss many frequent subtrees. Unordered tree mining has been successfully applied in [7] for the analysis of phylogenetic databases.

Our work in the area of frequent subtree mining is characterized by the Tree Model Guided (TMG) candidate generation [8, 9] which utilizes the underlying model of the data structure for efficient candidate generation. This non-redundant systematic enumeration technique ensures that

all the candidate subtrees generated are valid, in the sense that they conform to the actual tree structure of the data. For an efficient implementation of the TMG approach we utilized our novel Embedding List (EL) representation of the tree structure, an this resulted in efficient algorithms for mining embedded (MB3) [8] and induced (IMB3) [10] subtrees, from a databases of labeled rooted ordered subtrees. We have also provided some theoretical analysis of the worse case complexity of enumerating all possible embedded [8, 9] and induced [11] subtrees. In [12] we have developed an algorithm for mining distance-constrained embedded subtrees, which is useful for applications where the distance between the nodes in the sub-structure is considered important and used as an additional candidate grouping criterion. From the application perspective, in [13] we have indicated the potential of the tree mining algorithms for providing interesting biological information when applied to tree structured biological data. Our research focus has shifted to the unordered tree mining and in this paper we present an algorithm for mining unordered induced subtrees from a database of rooted ordered labeled subtrees. Rather than developing an algorithm specifically tailored for mining induced unordered subtrees we extend our general tree mining framework in order to indicate the flexibility of our general approach to the tree mining problem. Furthermore, our algorithm has the capability of using the occurrence match support which is absent in the previously developed algorithms for mining unordered induced subtrees. In [9] the need and the application of the occurrence match support was discussed.

The rest of the paper is organized as follows. In Section II we give the problem definition. Section III discusses some of the necessary aspects of unordered tree mining. Our algorithm is described in Section IV and it is experimentally evaluated and compared to existing techniques in Section V. Section VI concludes the paper.

II. PRELIMINARIES

A tree T is an acyclic connected graph with the node at the top defined as the *root[T]*. A tree can be denoted as $T(v_0,V,L,E)$, where (1) $v_0 \in V$ is the root vertex; (2) V is the set of vertices or nodes; (3) L is the set of labels of vertices, for any vertex $v \in V$, $L(v)$ is the label of v; and (4) E is the set of edges in the tree. In a labeled tree, there is a labeling function mapping vertices to a set of labels and a label can be shared among many vertices. The *Parent* of node v (*parent[v]*) is defined as its predecessor. Each node in the tree can only have one parent, but it can have one or more children, which are defined as its successors. The fan-out or degree of a node corresponds to the number of children of that node. A *leaf node* is a node without a child; otherwise, it is an internal node. A path from vertex vi to vj, is defined as the finite sequence of edges that connects vi to vj. The length of a path p is the number of edges in p. If p is an ancestor of q and q is a descendant of p, then there exists a path from p to q. The *rightmost path* (RMP) of T is defined as the (shortest) path connecting the rightmost leaf with the root node. The *Depth/level* of a node is the length of the path from root to that node. The *size* of a tree equals to the total number of nodes in the tree.

Mining frequent subtrees. Let *Tdb* be a tree database consisting of N transactions of trees, K_N. The task of frequent subtree mining from Tdb with given minimum support *(σ),* is to find all candidate subtrees that occur at least σ times in *Tdb*.

Definition 1: A tree $T'(r', V', L', E')$ is an ordered induced subtree of a tree $T(r, V, L, E)$ iff (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) $L' \subseteq L$ and $L'(v)=L(v)$, (4) $\forall v' \in V'$, $\forall v \in V$, v' is not the root node, and v' has a parent in T', then parent(v')=parent(v), (5) the left-to-right ordering among the siblings in T' is preserved. An induced subtree T' of T can be obtained by repeatedly removing leaf nodes or the root node if its removal doesn't create a forest in T.

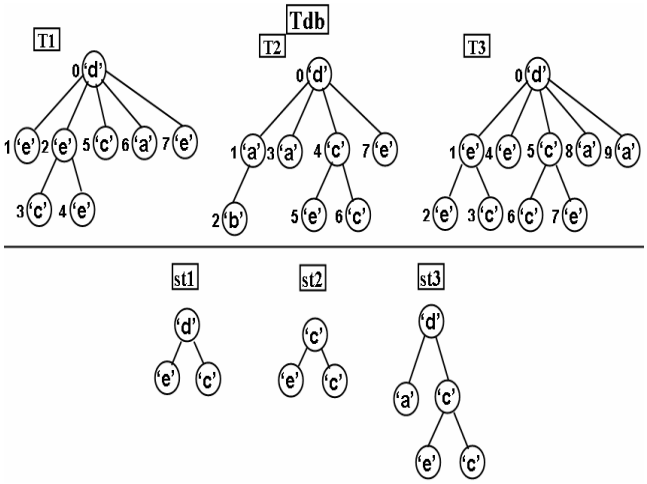
Definition 2: A tree $T'(r', V', L', E')$ is an unordered induced subtree of a tree $T(r, V, L, E)$ iff conditions 1, 2, 3 and 4 from Definition 1 are met, and the condition 5 is relaxed so that the left-to-right ordering among the siblings in T' does not need to be preserved. In other words the left-to-right ordering among the siblings (taking the subtrees rooted at sibling nodes into account) can be exchanged and the resulting subtree would be considered the same.

The difference between counting the occurrences of ordered and unordered induced subtrees can be seen from fig. 1, by comparing columns 3 and 4 respectively.

Definition 3: A tree $T'(r', V', L', E')$ is an ordered embedded subtree of a tree $T(r, V, L, E)$ iff it satisfies property 1, 2, 3, 5 of an induced subtree (Definition 1) and it generalizes property (4) such that $v' \in V'$, $v \in V$ and v' is not the root node, the sets *ancestor(v')* and *ancestor(v)* form a non-empty intersection. An example of an ordered embedded subtree would be the occurrence of 'st1' in T3 from Fig. 1, where it occurs at node positions of '023'.

Definition 4: If $T'(r', V', L', E')$ is an embedded subtree of T, and there is a path between two nodes p and q, the level of embedding between p and q, denoted by $\Delta(p,q)$, is defined as the length of the shortest path between p and q, where $p \in V'$ and $q \in V'$, and p and q form an ancestor-descendant relationship. In other words, given a tree database *Tdb* and the maximum level of embedding constraint δ then any two ancestor-descendant nodes present in an embedded subtree of *Tdb*, will be connected in *Tdb* by a path that has the maximum length of δ. In this regard, we could define induced subtree T as an embedded subtree where the maximum *level of embedding* that can occur in T is equal to 1, since the level of embedding of two nodes that form a parent-child relationship equals to 1. Throughout the paper, we will use notation $\Delta(p,q)$ to refer to the level of embedding between two nodes p and q. We will occasionally use notation Δ for the level of embedding concept when no reference to two nodes. When referring to

the maximum level of embedding constraint on the other hand we will use notation δ as to avoid confusion.



Subtree	String encoding (φ)	Induced(i) occurrence coordinates	Unordered induced (u) occurrence coordinates	TS	OM
st1	d e / c	T1:015, 025 T3: 015, 045	T1: 015, 025, 075 T2: 074 T3: 015, 045	i:2 u:3	i:4 u:6
st2	c e / c	T2: 456	T2: 456 T3: 576	i:1 u:2	i:1 u:2
st3	d a /c e / c	T2: 01456, 03456	T2: 01456, 03456 T3: 08576, 09567	i:1 u:2	i:2 u:4

Fig. 1: Example of ordered induced (i) and unordered induced (u) subtrees and implications when transaction based (TS) and occurrence match (OM) support are used

Definition 5: The notation $t \prec k$, is used to denote a subtree *t* which is supported by transaction *k* in database of tree *Tdb*. A transaction *k* supports subtree *t* if it contains at least one occurrence of subtree *t*. If there are *L* occurrences of *t* in *k*, a function *g(t,k)* denotes the number of occurrences of *t* in transaction *k*. For transaction-based support, $t \prec k=1$ when there exists at least one occurrence of t in transaction k. In other words, for transaction-based support, the support of a subtree *t* is equal to the numbers of transactions that support subtree *t*.

Definition 6: For occurrence-match support, $t \prec k$ corresponds to the number of all occurrences of t in transaction *k*, $t \prec k=g(t,k)$. Suppose that there are N transactions k_1 to k_N of tree in *Tdb*, the support of an subtree *t* in *Tdb* is defined as:

$$\sum_{i=1}^{N} t \prec k_i \qquad (1)$$

An example that illustrates the effect of applying different support definitions described above follows. In fig. 1 there are three transactions, T1, T2 and T3. Suppose that transaction-based support is used and that unordered induced subtrees are considered. The support of subtree st3 is equal to 2 since st3 is supported by T2 andT3 but not T1, i.e. $st3 \prec T2$ and $st3 \prec T3$. On the other hand, if occurrence-match support is considered, the support of subtree st3 is equal to the sum of its occurrences in T2,and T3 i.e. $g(st3,T2)+g(st3,T3)$. It can be seen from fig. 1 that there are two occurrences of st3 in T2 and two occurrences of st3 in T3, but none in T1. Hence the occurrence-match support of subtree st3 equals to 4.

Choosing an appropriate tree encoding is another important requirement in tree mining. Our work utilizes the pre-ordering string encoding (φ) as described in [14, 8, 9]. We denote encoding of a subtree T as φ(T). For each node in Tdb (Fig. 1), its label is shown as a single-quoted symbol inside the circle whereas its pre-order position is shown as an index at the left side of the circle. From fig. 1, *φ(T1):* 'd e / e c / e / / c / a / e / '; *φ(T2):* 'd a b / / a / c e / c / / e / ', etc. The backtrack symbol ('/') is used whenever we have to move up a node in the tree during the pre-order traversal of the tree being represented by the encoding. We could omit the backtrack symbols after the last node like it was done in the second column of Fig.1. We refer to a group of subtrees with the same encoding *L* as candidate subtrees C_L. A k-subtree. is a subtree with k number of nodes. Throughout the paper, the '+' operator is used to denote the operation of appending two or more tree encodings. However, this operator should be contrasted with the conventional string append operator, since the backtrack symbols need to be computed accordingly.

To ensure that the downward-closure lemma holds [15], each k-1-subtree of a frequent k-subtree has to be frequent. Hence, during the candidate enumeration and counting phase the k-subtrees that contain any infrequent k-1 subtrees have to be pruned from the frequent set ('Fk'). This problem is known as k-1 pruning [14, 8, 9], and for the transactional support definition, opportunistic approaches [14] have been employed to achieve the desired result in less time. However, when using occurrence-match support, full (k-1) pruning should be performed at each iteration of generating a k-subtree from a (k-1)-subtree so that no 'pseudo-frequent' subtrees [8] would be generated. The rationale of this has been explained in [8, 9].

III. MINING UNORDERED SUBTREES

The main difference in mining unordered subtrees to other subtree mining approaches lies in the candidate enumeration phase. The candidates should be enumerated in a complete and non-redundant manner, and each candidate subtree should be uniquely distinguished by its encoding. It is the problem of determining whether two trees are equal to one another and it is a known problem of tree isomorphism. Two trees are isomorphic if there is a bijective correspondence between their node sets which preserves and reflects the structure of the trees [16].

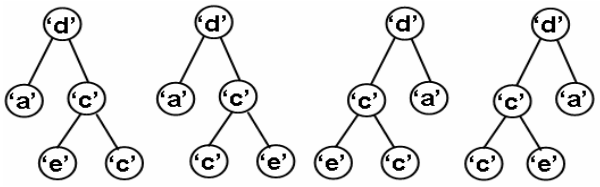


Fig. 2: Possible subtree permutations of subtree st3 (Fig.1)

The isomorphism problem is more complex when mining unordered subtrees. When permutations of subtrees rooted at any node in a particular tree T are performed, the resulting trees are non-isomorphic ordered trees and isomorphic unordered trees to T and among themselves. This aspect is demonstrated in Fig. 2 where all possible ordered trees of the subtree st3 (Fig. 1) are shown. They are all isomorphic unordered trees since each one can be mapped into another by permuting the children of vertices. In Fig. 1 it can be seen that the subtree st3 is counted twice for the ordered case (I) and four times for the unordered case (UI) since additional subtree permutations are allowed. Each tree encoding should uniquely map to only one subtree, which enables the use of traditional hashing methods for efficient tree counting. Therefore, from the set of ordered trees obtained through all possible permutations of subtrees rooted at any node, one has to be selected to represent the unordered subtree. This selected tree is known as the canonical form (CF) of an unordered subtree. In the context of tree mining the selected CF is usually dependant on the particular candidate enumeration technique used. The aim should be that the enumerated candidate subtrees will require less sorting on average, since sorting the tree encodings may become one of the performance bottlenecks. Within our implementation framework, we have used the depth-first CF (DFCF) proposed in [3], with the difference that 'smaller' subtrees are placed to the left. Using the DFCF, the sorting of candidate subtrees is done solely based upon the alphabetical order of the labels. Additionally the backtrack ('/) symbol is considered smaller than any other label. This ordering of nodes starts from the leaf nodes and continues up the tree structure. When the labels of non-leaf nodes are equal the subtrees rooted at those nodes are traversed in a depth-first manner during which encountered nodes are compared. As soon as a label is encountered that is larger than its corresponding sibling node label (or if there is no such node in the sibling subtree), the right order of those sibling nodes is known. If necessary the sibling nodes and the subtrees rooted at those nodes are swapped around to satisfy the CF order. This CF ordering scheme (CFOS) maps each candidate subtree uniquely. As an example consider the tree T (*φ(T)*:'*d b / a d / / a b / / a d / b / /a b c*), and the result of applying the CFOS, Tcf (*φ(Tcf)*:'*d a b / / a b / d / / a b c / / / a d / / b)* shown in Fig.3.

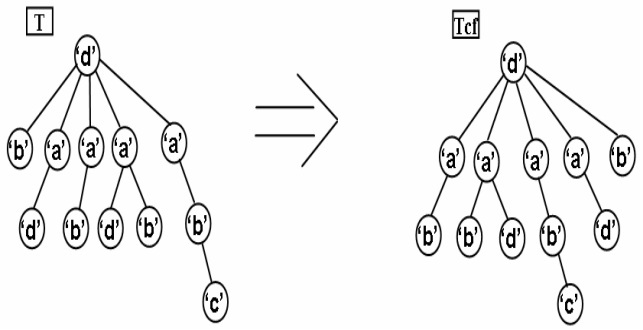


Fig. 3: Result of applying the employed CFOS to tree T

Ordering a subtree into its CF can be quite expensive due to the expensive traversal of the string encodings in order to determine and compare the sibling nodes. Due to the fact that in our approach the whole tree database is first sorted into its CF and each previously enumerated k-1 subtree is ordered, many subtrees may already be in their CF when new extensions are performed. We have determined a few preconditions that allow us to assume that a subtree is already in its CF and that no ordering is required. These preconditions occur when we are appending a new node 'n' to the right-most node 'r' of the currently expanding subtree.

Precondition 1: parent(n) = r;

Precondition 2: Let the left sibling of n be 'ln', then children(ln) = null and L(ln) = L(n), OR L(ln) < L(n).

If any of the above conditions are met the ordering can be skipped which results in a run time reduction as will be demonstrated in our experimental section.

IV. UNI3 ALGORITHM

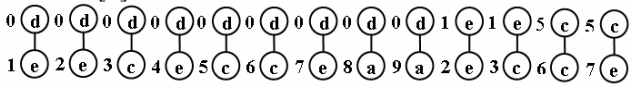
We start this section by first giving a brief overview of the algorithm and then we explain each step in more detail. For faster processing, the XML database is first transformed into a database of rooted integer-labeled ordered trees. The tree structure is then ordered into its CF, using the CFOS described above. The tree database is traversed once to create a global sequence which stores each node in the pre-order traversal together with the necessary node information (position, label, scope, and level*)*. At the same time the set of frequent 1-subtrees (*'F1'*) is obtained by hashing the encountered node labels. Embedding List (EL) is constructed for a suitable representation of the tree structure and the set of frequent 2-subtrees (*'F2'*) is obtained at the same time. TMG candidate generation using the EL structure takes place and for each $k \geq 1$ the right most path (RMP) coordinates of each frequent (k-1)-subtree are stored in 'F_{k-1}' hashtable. Prior to hashing the string encoding of each subtree, it is first ordered into its CF if necessary (i.e precondition 1 and 2 from section III are not met). Each frequent (k-1)-subtree is extended one node at a time, starting from the last node of its RMP (right most node), up to its root. The whole process is repeated until all k-subtrees are enumerated and counted

XML Data Pre-processing. To expedite the frequency counting, the database of XML documents is first transformed into a database of rooted integer-labeled ordered trees. One format to represent the database of rooted integer-labeled ordered trees is proposed in *(Zaki [2005])*. For each XML tag, we consider tagname, attribute(s) and value(s). To mine the structure of XML documents one can modify this easily by omitting the presence of attribute(s) and value(s) for each tag. Each tag in an XML document is mapped to a unique integer. Since the string labels can be long, performing the mapping will optimize the frequency counting process by avoiding additional hash key computations.

Another pre-processing step consists of ordering the whole tree structure into its CF. We have found that this step is necessary for optimization purposes, since the number of the generated candidate subtrees that need to be sorted into their CF greatly reduces.

Database Scanning. The process of frequent subtree mining is initiated by scanning a tree database, T_{db}, and generating a global pre-order sequence D in memory (*dictionary*). The purpose of the dictionary is to provide a shared global nodes' related information that allows for direct access and thereby avoids the space cost which would be caused if this information is copied (stored) locally for every occurrence of a node. The dictionary stores each node in T_{db} following the pre-order traversal indexing. For each node its *position*, *label*, *scope*, and *level* are stored. The level of a node refers here to the level of the T_{db} tree, at which this node occurs. An item in the dictionary D at position i is referred to as $D[i]$. The notion of the position of an item refers to its index position in the dictionary. When generating the dictionary, we compute all the frequent 1-subtrees, F_1. After this step no further database scanning is required.

Embedding List (EL) Construction. For each frequent internal node in F_1, a list is generated which stores its descendant nodes' positions (from dictionary) in pre-order traversal ordering such that the embedding relationships between nodes are preserved. For a given internal node at position i, such ordering reflects the enumeration sequence of generating 2-subtree candidates rooted at i (Fig. 4). Hereafter, we call this list as *embedded list (EL)*. We use notation *i-EL* to refer to an embedded list of node at position i. The position of an item in EL is referred to as *slot*. Thus, *i-EL[n]* refers to the item in the list at *slot* n. Whereas $|i\text{-}EL|$ refers to the size of the embedded list of node at position i. Fig. 4 illustrates an example of the EL representation of tree T3 (Fig. 1). In Fig. 4, 0-EL for example refers to the list: *0:[1,2,3,4,5,6,7,8,9]*, *0-EL[0]*=1 and *0-EL[6]*=7.



0:	1	2	3	4	5	6	7	8	9
1:	2	3							
5:	6	7							

Fig. 4: The EL representation of T3 in fig 1

RMP Occurrence Coordinate (RMP-OC). By its definition, RMP is the shortest path from the right most node to the root node. Thus storing RMP coordinates is always guaranteed to be maximal. The worst case of storing the RMP coordinates would be equal to storing every coordinate of a node in a subtree, i.e. when the subtree becomes a sequence (each node has degree 1). The best case of storing RMP coordinates for k-subtrees where $k>1$ is that it stores only 2 coordinates, i.e. whenever the length of the RMP is equal to 1. Given a k-subtree T with OC $[e_0,e_1,\ldots e_{k-1}]$, the RMP-OC of T, denoted by $\Psi(T)$, is defined by $[e_0,e_1,\ldots,e_j]$ such that $\Psi(T) \subseteq OC(T)$; $e_j = e_{k-1}$; and $j \leq k-1$ and the path from e_j to e_0 is the RMP of tree T.

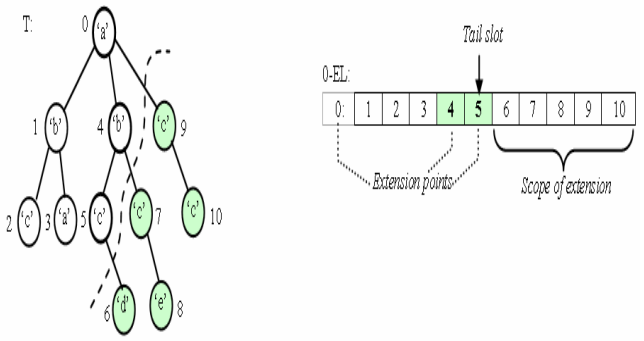


Fig. 5: TMG enumeration: extending (k-1)-subtree t_{k-1} where $\varphi(t_{k-1})$*:'a b / b c'* occurs at position *(0,1,4,5)* with node at position 6, 7, 8, 9, and 10

TMG enumeration formulation. TMG is a specialization of the right most path extension method which has been reported to be complete and where all valid candidates are enumerated at most once (non-redundant) *[8, 14]*. To enumerate all embedded k-subtrees from a (k-1)-subtree, the TMG enumeration approach extends one node at a time to each node in the RMP of (k-1)-subtree as illustrated in the Fig. 5. This is our general tree mining framework that allows for mining of induced and embedded subtrees. However, in this work we are concerned with mining induced subtrees and hence the maximum level of embedding δ will always be constrained to 1. Suppose that nodes in the RMP of a subtree are defined as *extension points* and the level of embedding between two nodes at position n and t is denoted by $\Delta(n,t)$. The TMG can be formulated as follows. Given an RMP-OC of a frequent (k-1)-subtree T_{k-1}, $\Psi(T_{k-1}):[e_0,e_1,\ldots e_j]$, the *scope* of the root node e_0 is Φ, and the maximum level of embedding constraint is 1, k-subtrees are generated by extending each extension point $n \in \Psi(T_{k-1})$ with t for which it satisfies the following conditions: (1) $n < t \leq \Phi$, (2) $\Delta(n,t) = \delta$. Suppose that the encoding of T_{k-1} is denoted by L_{k-1} and $l(n,t)$ is a labeling function of extending extension point n with a node at position t, L_k is defined as $L_{k-1}+l(n,t)$. $l(n,t)$ computes the length between the extension point n and the right most node f_{k-1} such that when the length τ is > 0, τ

numbers of backtrack symbols '/' are appended before the label of node at position $t, \varphi(t)$. To generate RMP at each step of candidate generation, we utilize the computed numbers of backtracking τ between the extension point n and the extending node t. Given that the $\Psi(T_{k-1})$ is $[e_0,e_1,\ldots,e_j]$, the RMP of the k-subtree can be generated by appending t at position $(j+1)-\tau$ of the $\Psi(T_k)$ and t will be the right most node. Thus, the bigger the value of τ is, the shorter the length of the generated RMP. The best case is given when the extension point is the root node. This will make sure that at each extension of (k-1)-subtree we store the RMP coordinates of k-subtree.

Pruning. As mentioned previously k-1 pruning [8, 9, 14] needs to be performed in order to check that all subtrees of a currently expanding subtree are frequent. Each of those subtrees has to be ordered into its CF if necessary, in order to correctly check for its frequency. The expanding subtree is pruned if at least one of its subtrees is infrequent. Doing full k-1 pruning is quite time consuming and expensive. To accelerate full k-1 pruning, a caching technique is used by checking whether a candidate is already in the frequent k-subtree hashtable (Fk). If a k-subtree candidate is already in Fk, it is known that all its (k-1)-subtrees are frequent, and hence only one comparison is made.

Vertical Occurrence List (VOL). Each occurrence of a subtree is stored as an RMP-OCs as previously described. The vertical occurrence list of a subtree groups the RMP-OCs of the subtree by its encoding. Hence, the frequency of a subtree can be easily determined from the size of the VOL. We use the notation *VOL(L)* to refer to the vertical occurrence list of a subtree with encoding *L*. Consequently, the frequency of a subtree with encoding *L* is denoted as *|VOL(L)|*. However, when the transaction-based support is used there is a transaction identifier (tid) associated with each occurrence and the support is determined by the number of unique transaction identifiers.

The cost of the frequency counting process comes from at least two main areas. First, it comes from the VOL construction itself. With numerous numbers of occurrences of subtrees the list can grow very large. Secondly, for each candidate generated its encoding needs to be computed. Constructing an encoding from a long tree pattern can be very expensive. An efficient and fast encoding construction can be employed by a step-wise encoding construction so that at each step the computed value is remembered and used in the next step. This way a constant processing cost that is independent of the length of the encoding is achieved. Thus, fast candidate counting can be achieved. Overall, our algorithm can be described by the pseudo-code provided in Fig. 6.

```
Inputs  : T_db(Tree database), σ(min.support), δ(maximum level of embedding)
Outputs : F_1(Frequent subtrees), D(dictionary)
{D, F_1}  : DatabaseScanning (T_db)

{EL, F_2} : ConstructEmbeddingList (F_1, D, δ)
k=3

while( |F_k| ≥ 0 )

  F_k = GenerateCandidateSubtrees (F_k-1, δ)
  k = k+1


GenerateCandidateSubtrees (F_k-1, δ):
for each frequent k-subtree t_k-1 ∈ F_k-1
  L_k-1 = GetEncoding (t_k-1)
  VOL-t_k-1 = GetVOL(t_k-1) // get occurrences of t_k-1
  for each occurrence coordinate oc_k-1 (r:[m,..n]) ∈ VOL-t_k-1
    for (j = n+1 to scope(r))
      {extpoint,slashcount} = CalcExtPointAndSlashcount(oc_k-1, j);
        L_k = L_k-1 + append('/',slashcount) + Label(j)
        If (Precondition1(L_k) == false && Precondition2(L_k)== false)
          ApplyCFOS(L_k) // order L_k into its CF

        if(EmbeddingLevel(extpoint,j)≤ δ) then
          oc_k = TMG-extend( oc_k-1, j )
          if( Contains(L_k, F_k) )
            Insert( h(L_k),oc_k, F_k )
          else
            If( k-1Pruning (L_k) == false) //all k-1 patterns frequent?
              Insert( h(L_k),oc_k,F_k )
return F_k

TMG-extend( oc_k-1, j ):
/* right-most-path computation */
|oc_k|= |oc_k-1|+1-slashcount; //compute size of right-most-path coordinate
α = |oc_k|-1; // update the tail index α
oc_k[α] = j; // store the new node pos at tail index α
return {oc_k}
```

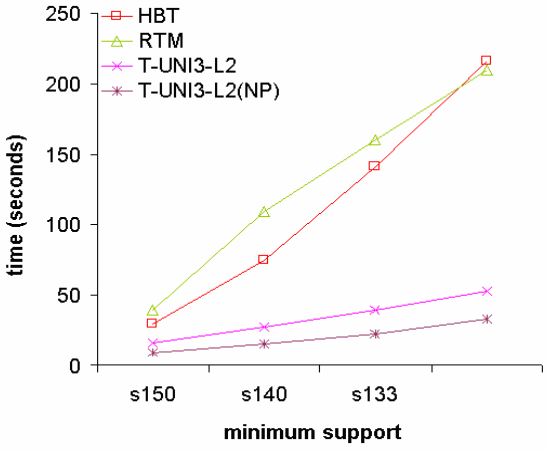
Fig. 6: UNI3 pseudo code

V. EXPERIMENTAL EVALUATION

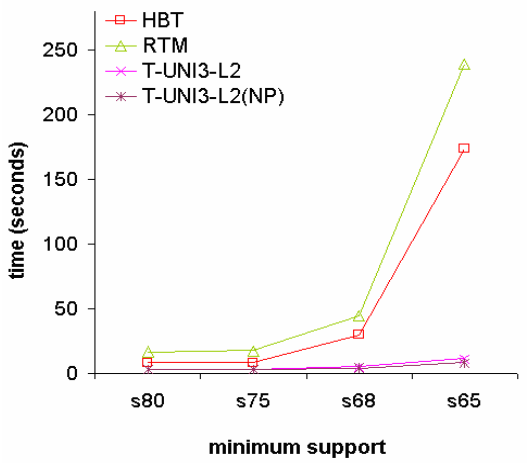
In this section we provide some performance evaluation of the UNI3 algorithm by comparing it with the HybridTreeMiner (HBT) [4] and RootedTreeMiner [3] which is Chi's implementation of the Unot algorithm [1]. We will refer to each algorithm in this section using its abbreviation as indicated in the brackets. For transaction based support our algorithm is preceded with 'T-' (e.g. T-UNI3), when the exceptions to CF ordering are used a '-Lx' symbol is appended at the end, where x corresponds to the precondition explained in section III (e.g. UNI3-L1), and if no full k-1 subtree pruning is performed (NP) is added at the end (e.g. UNI3(NP)). Real world and artificial databases of trees are used. 'CSLogs' is a real world data set previously used by Zaki for testing his TreeMiner algorithms [14]. The minimum support σ is denoted as (sxx), where xx is the minimum frequency. The first three experiments use the transaction based support whereas the last one is based on occurrence match support. Experiments

were run on 3Ghz (Intel-CPU), 2Gb RAM, Mandrake 10.2 Linux machine and compilations were performed using GNU g++ (3.4.3) with -g and -O3 parameters.

Time Performance Test. For this test we use the CSLogs datasets and a synthetic dataset characterized by deep tree structures consisting of 20000 transactions. We have used the T-UNI3-L2 implementation where both preconditions explained in section III are used to prevent unnecessary CF checking of candidate subtrees. As can be seen in Fig. 7, for both datasets the T-UNI3-L2 algorithm enjoys the best time performance. Additionally by not performing full k-1 pruning there was an additional performance gain because CF checking does not need to be performed for all the k-1 subtrees of a potentially frequent k-subtree.



(a) CSLogs dataset



(b) Deep tree dataset

Fig. 7: Time performance test

Scalability Test. For this experiment we have generated a synthetic datasets that consists of 10000 items, and has an average depth and fan-out of 40. The number of transactions used was 2.5M, 5M, 10M, and the respective support threshold was 162, 325 and 650. From Fig. 8, one can see that all the tested algorithms are well scalable for the different dataset sizes used, and the time performance is comparable among the algorithms with T-UNI3-L2 performing slightly better than others.

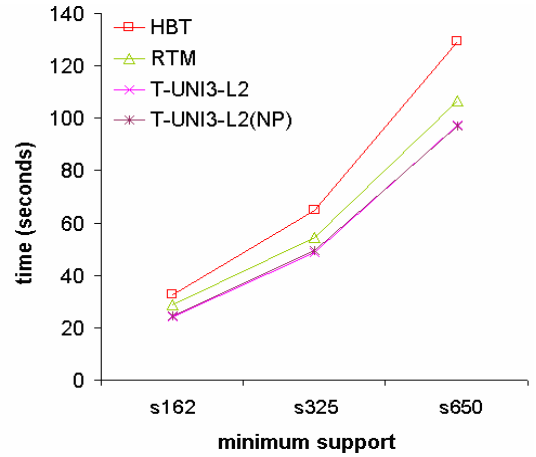


Fig. 8: Scalability test

Variations of UNI3 Test. In this experiment we compare the performance of the UNI3 variations. The aim is to demonstrate some of the important implementation issues that need to be taken into account when developing unordered tree mining algorithms. The 'L0' version does not implement any of the preconditions for detecting exceptions to CF subtree ordering, L1 implements the first precondition from section III and L2 enforces both preconditions. Looking at the graph from Fig. 9, implementing both preconditions for CF ordering exceptions results in best time performance. This is because the CF ordering and checking is quite expensive and avoiding any of these operations will result in a time gain. Additionally performing no full k-1 pruning (NP) results in further time performance gain since again less CF ordering is needed among the k-1 subtrees of a potentially frequent k-subtree.

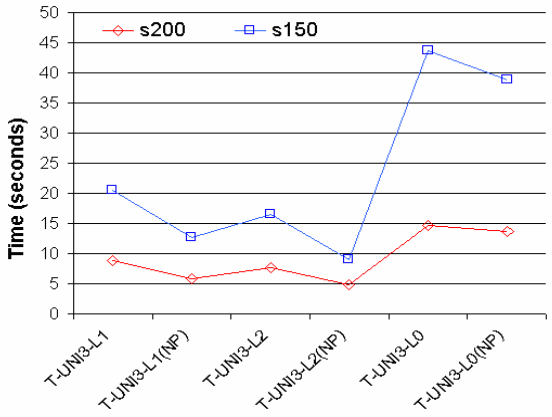


Fig. 9: Testing UNI3 variations

Occurrence Match Support (OMS). The purpose of this experiment is to show the performance of the UNI3 algorithm when OMS threshold is used. Since to our knowledge there are no current algorithms for mining induced unordered subtree using the OMS, we have included the performance of our algorithm for mining

ordered induced subtrees for extra comparison. The dataset was artificially created and it consists of 1M transactions, 10000 items, and has average depth and fan-out of 40. Fig. 10 shows that our algorithm is well scalable even when the more complex occurrence match support is used. Furthermore, our T-UNI3-L2 algorithm enjoys the best time performance for this dataset when compared to HBT and RTM.

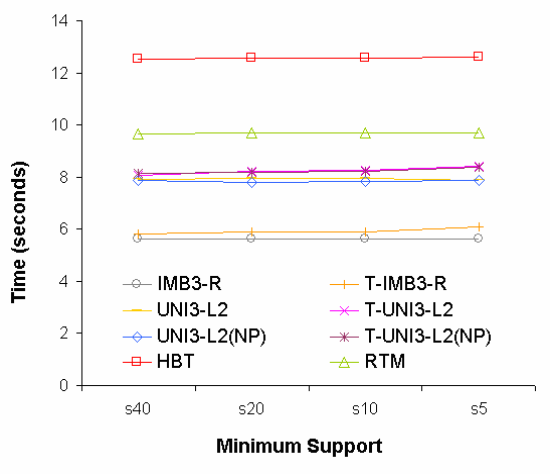


Fig. 7: Occurrence Match support test

VI. Conclusions and Future Work

In this paper we have extended our general tree mining framework for the capability of mining induced unordered subtrees. The flexibility of our general approach to tree mining is again demonstrated through better time performance when compared to some of the existing state-of-the-art algorithms. Within our implementation framework, two exceptions were indicated where the expensive CF ordering can be avoided and an improvement in time was experimentally demonstrated. Furthermore, our algorithm has the capability of using the more complex occurrence match support which is absent in the previously developed algorithms for mining unordered induced subtrees. Our future work consists in exploring further space and time efficiency issues of the unordered tree mining problem, and presenting an efficient algorithm for mining embedded unordered subtrees.

Acknowledgment

We would like to thank Yun Chi for providing us with the source code of the HybridTreeMiner and RootedTreeMiner algorithms, and for discussing some of the results with us.

References

[1] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering Frequent Substructures in Large Unordered Trees", The *6th International Conference on Discovery Science*, 2003.

[2] S. Nijssen, and J.N. Kok, "Efficient discovery of frequent unordered trees", In Proc. of the *1st International Workshop Mining Graphs, Trees, and Sequences (MGTS-2003)*, Dubrovnik, Croatia, 2003.

[3] Y. Chi, Y. Yirong, and R. R. Muntz, "Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining", *Knowledge and Information Systems*, 2004.

[4] Y. Chi, Y. Yang, and R.R. Muntz, "HybridTreeMiner: An efficient algorihtm for mining frequent rooted trees and free trees using canonical forms", In *Proc. of the 16th International Conference on Scientific and Statistical Database Management*, Santorini Island, Greece, 2004.

[5] M.J. Zaki, "Efficiently Mining Frequent Embedded Unordered Trees", *Fundamenta Informaticae 65*, IOS Press, 2005, pp. 1-20.

[6] A. Termier, M-C. Rousset, and M. Sebag, "Treefinder: A First Step Towards XML Data Mining" In *Proc. of IEEE ICDM'02*, 2002.

[7] D. Shasha, J.T.L. Wang, S. Zhang, "Unordered Tree Mining with Applications to Phylogeny", *20th International Conference on Data Engineering*, 2004.

[8] H. Tan, T.S. Dillon, F. Hadzic, E. Chang, and L. Feng, "MB3 Miner: mining eMBedded sub-TREEs using Tree Model Guided candidate generation", In Proc. of the *1st International Workshop on Mining Complex Data*, held in conjunction with ICDM'05, Houston, Texas, USA, 2005.

[9] H. Tan, T.S. Dillon, F. Hadzic, L. Feng, and E. Chang, "Tree Model Guided Candidate Generation for Mining Frequent Subtrees from XML", Submitted to *Transactions on Knowledge Discovery from Data (TKDD)*, January, 2006, unpublished.

[10] H. Tan, T.S. Dillon, F. Hadzic, L. Feng, and E. Chang, "IMB3 Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding", In *Proc. of PAKDD'06*, Singapore, 2006.

[11] H. Tan, T.S. Dillon, F. Hadzic, E. Chang, and L. Feng, "Mining induced/embedded subtrees using the level of embedding constraint", Submitted to *Knowledge and Information Systems An International Journal*, Springer, 2006, unpublished.

[12] Tan, H., Dillon, T.S., Hadzic, F , and Chang, E., 2006. "Distance constrained mining of embedded subtrees", *Workshop on Ontology Mining and Knowledge Discovery from Semistructured documents (MSD 2006)*, in conjunction with the 2006 International Conference on Data Mining, 18-22 December, Hong Kong, in press.

[13] F. Hadzic, T.S. Dillon, A. Sidhu, E. Chang, and H. Tan, "Mining Substructures in Protein Data", *IEEE ICDM 2006 Workshop on Data Mining in Bioinformatics (DMB 2006)*, in conjunction with the 2006 International Conference on Data Mining, 18-22 December, Hong Kong, in press.

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

[15] R. Agrawal, and R. Srikant, "Fast algorithm for mining association rules", In *Proceedings of the 20th Very Large Data Bases (VLDB 1994)*, Santiago de Chile, Chile, 1994, pp. 487-499.

[16] G., Valentine, *Algorithms on Trees and Graphs*, Springer-Verlag, Berlin, 2002.