©2008 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology

U3 – mining unordered embedded subtrees using TMG candidate generation

Fedja Hadzic, Henry Tan, Tharam S. Dillon,
Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Perth, Australia
{fedja.hadzic, henry.tan, tharam.dillon}@cbs.curtin.edu.au

Abstract

In this paper we present an algorithm for mining of unordered embedded subtrees. This is an important problem for association rule mining from semi-structured documents, and it has important applications in many biomedical, web and scientific domains. The proposed U3 algorithm is an extension of our general tree model guided (TMG) candidate generation framework and it considers both transaction based and occurrence match support. Synthetic and real world data sets are used to experimentally demonstrate the efficiency of our approach to the problem, and the flexibility of our general TMG framework.

1. Introduction

Semi-structured documents such as XML possess a hierarchical document structure, where an element may contain further embedded elements, and each element can be attached with a number of attributes. It is therefore frequently modeled using a rooted ordered labeled tree. The task of frequent subtree mining is to extract all subtree patterns from a tree database, that occur at least as many times as the user specified minimum support threshold. Frequent subtree mining algorithms have many important applications in areas such as Bioinformatics, Web mining, scientific data management, and in other domains where the knowledge can be modeled using a rooted ordered labeled tree. For example, Web logs can be effectively represented using XML documents [1], and a frequent subtree mining algorithm can be used to extract informative substructures that can be very useful for determining common user activity and interests [2]. Other interesting application is mining of online biological databases, and the work presented [3] demonstrated the potential of discovering useful patterns from a protein ontology database.

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 may not be of great importance to the user and is often not available. Our work in the area of frequent subtree mining is characterized by the Tree Model Guided (TMG) candidate generation [4, 5, 6] which utilizes the underlying model of the data structure for efficient candidate subtree 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. While in our previous works the focus was on ordered induced/embedded [4, 5] and unordered induced subtrees [6], in this paper we present an algorithm for mining of unordered embedded subtrees. The main difference between an induced and an embedded subtree is that, while an induced subtree keeps the parent-child relationships from the original tree, an embedded subtree allows a parent in the subtree to be an ancestor in the original tree. In other words, an embedded subtree generalizes the definition of an induced subtree by preserving ancestor-descendant relationships. Mining of embedded subtrees is a much more difficult problem than mining of induced subtrees, as it is necessary to examine several levels within a tree to identify the embedded subtree. It is important to be able to mine embedded subtrees in order to discover interesting relationships between the data objects embedded deeply in the tree database. Previous results in the ordered case [2, 4, 5] indicate that the complexity of mining embedded subtrees is much higher. Therefore, this extension is clearly a nontrivial extension which leads to a strong generalization.

The contributions of this paper are as follows. We present an extension to our general TMG framework to mine unordered embedded subtrees. The space efficiency of our general TMG framework is improved with an adjustment of the representative structure. The

978-0-7695-3496-1/08 $25.00 © 2008 IEEE
DOI 10.1109/WIIAT.2008.403

IEEE computer society

representative structure used previously in our TMG framework, was efficient and conceptually simple for enumerating candidate subtrees. However, as a trade-off the structure was not very space efficient since the information about the descendant nodes was stored for every node of the database tree. This can be an issue when processing a very large tree database. In this paper we improve the space efficiency property of the representative structure by using the dictionary of traversal of nodes in conjunction with the list structure. With this optimization, the information necessary for candidate subtree enumeration is available without the trade-off for space efficiency. A substantial saving of the memory is achieved and this aspect will be demonstrated in the experiments. The proposed algorithm is evaluated by comparing it with the SLEUTH [7] algorithm, which is to the best of our knowledge the only existing algorithm that extracts a complete set of unordered embedded subtrees. In Section 2 we discuss some general tree concepts and define the problem. The proposed U3 algorithm is described in Section 3. Section 4 overviews some related work to unordered subtree mining. An experimental evaluation of our algorithm is given in Section 5. Section 6 concludes the paper.

2. Tree Concepts and Problem Definition

A tree *T* is an acyclic connected graph with the node at the top defined as the *root[T]*. The *Paren*t of node *v* (*parent[v])* is defined as its predecessor. Two nodes that share the same parent are referred to as *sibling nodes*. 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 v_i to v_j, is defined as the finite sequence of edges that connects v_i to v_j. 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. In this paper, the term 'k-subtree' refers to a subtree that consists of k number of nodes. A tree can be denoted as *T(V,L,E)*, where (1) *V* is the set of vertices or nodes; (2) *L* is the set of labels of vertices, for any vertex $v \in V$, *L(v)* is the label of *v*; and (4) $E = \{(x,y) | x,y \in V\}$ 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.

Given a tree $S = (V_S, L_S, E_S)$ and tree $T = (V_T, L_T, E_T)$, *S* is an ***unordered embedded subtree*** of *T*, iff (1) $V_S \subseteq V_T$;(2) $L_S \subseteq L_T$ and $L_S(v)=L_T(v)$; (3) if $(v_1,v_2) \in E_S$ then $parent(v_2) = v_1$ in *S* and v_1 is ancestor of v_2 *in T.* An ***ordered*** subtree would preserve the left to right ordering of sibling nodes in the original tree while in an unordered subtree the order of the sibling nodes (and the subtrees rooted at those nodes) can be exchanged and the resulting subtree would be considered the same. This causes the enumeration and counting of unordered subtrees more difficult since each enumerated subtree needs to be ordered into one logical and consistent form, so that all its variants that have different order among sibling nodes are considered as the same subtree. The group of possible trees obtained by permuting the sibling nodes in all possible ways is referred to as the automorphism group of a tree [7]. During the pre-order traversal of a tree database, ordered subtrees are generated by default. One tree needs to be selected to uniquely represent the unordered tree. This selected tree is known as the canonical form (CF) of an unordered tree. Generally speaking, CF of an entity is a representative form (or a function) for which many equivalent variations of an entity can be represented (mapped) into one standard, conventional, logical form in a consistent manner [8, 9]. Within the algorithm proposed in this paper, the depth-first CF (DFCF) proposed in [8] is used to uniquely map each subtree, and it can be formally explained as follows:

Given two trees T_1 and T_2, with root[T_1] = r_1 and root[T_2] = r_2, let descendants(r_1):$\{c_1,\ldots,c_m\}$ be the set of descendants of node r_1 and descendants(r_2):$\{c_1,\ldots,c_n\}$ be the set of descendants of node r_2, ordered according to the pre-order traversal. Note that these sets can also contain the special backtrack symbol to indicate the backtracking during the traversal of the descendant nodes. Let $ST_x(r)$ denote the subtree of tree T_x with root node r, and |descendants(r)| denote the number of descendants of node r. The label(c_i) < label(c_j) if label(c_i) lexicographically sorts smaller than label (c_j), and so $T_1 < T_2$ iff:

a) label(r1) < label(r2) or,
b) if label(r1) = label(r2) and |descendants(r1)| = m, |descendants(r2)|=n, then either:
   i. $\forall$ $1 \le i < j \le \min(m,n)$ there exist j such that $ST_1(c_i) = ST_2(c_i)$ and $ST_1(c_j) < ST_2(c_j)$
   ii. m≤n, $\forall$ $1 \le i \le m$ and m≤n, $ST_1(ci) = ST_2(ci)$

Transaction-based and occurrence match support definitions exist to count the occurrences of a subtree.

When using the ***transaction-based support*** definition, the transactional support of a subtree t, denoted as $\sigma_{tr}(t)$ in a tree database T_{db} is equal to the number of transactions in T_{db} that contain at least one occurrence of subtree t. Let the notation $t \prec k$, denote the support of subtree t by transaction k, then for TS, $t \prec k = 1$ whenever k contains at least one occurrence of t, and 0 otherwise. Suppose that there are N transactions k_1 to k_N of tree in Tdb, the $\sigma_{tr}(t)$ in T_{db} is defined as $\sum_{i=1}^{N} t \prec k_i$. The ***occurrence-match support*** takes the repetition of items in a transaction into account and counts the subtree occurrences in the database as a whole. Hence, the occurrence-match support (σ) of a subtree t, denoted as $\sigma_{oc}(t)$, in a tree database T_{db} is equal to the total number of occurrences of t in all transactions in T_{db}. Let function $g(t,k)$ denote the total number of occurrences of subtree t in transaction k. If there are N transactions k_1 to k_N of tree in T_{db}, $\sigma_{oc}(t)$ in T_{db} can be defined as $\sum_{i=1}^{N} g(t, k_i)$.

Frequent subtree mining. 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. To ensure that the downward-closure lemma holds [10] 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 [2, 4, 5] and for the transactional support definition, opportunistic approaches [2] 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 [4, 5] would be generated. The rationale of this has already been explained in [4, 5] and an example of a pseudo-frequent subtree will be provided later in the experimental section.

An illustration of different subtrees is given in Figure 1. In this paper, the term 'occurrence coordinate(s) (*oc*)' will be used to refer to the position(s) of a particular node or a subtree in the tree database. In the case of a node, *oc* corresponds to the pre-order position of that node in the tree database (left of circle in Figure 1), whereas for a subtree, *oc* is a sequence of *oc* from nodes that belong to that particular subtree. On the right hand side of Figure 1, the *oc* of each subtree are presented with the corresponding transaction that they occur in. If ordered induced subtrees are mined, *st1* occurs only once in *T1* with *oc*:125; whereas, if unordered induced subtrees are mined, the order of node *'c'* and *'e'* can be exchanged and hence now *st1* occurs in *T2* as well with *oc*:132 (note: the order of occurrence coordinates is exchanged since the order of corresponding nodes is exchanged). If embedded subtrees are mined, a parent in *st1* subtree can be an ancestor in *Tdb* and hence many more occurrences of *st1* are counted as can be seen in the top right corner of Figure 1. For an example of difference in support definitions, suppose that unordered embedded subtrees are mined. The $\sigma_{tr}(st2)$ = 2 since *st2* is supported by both *T1* and *T2*, while $\sigma_{oc}(st2)$ = 7 since *st2* occurs three times in *T1* and four times in *T2*.

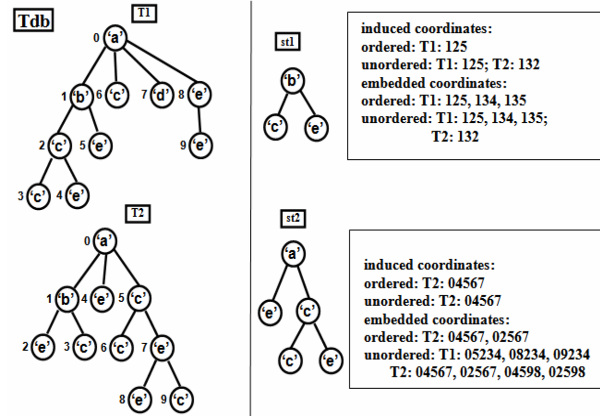


Figure 1. Example Tree Database (TDB) consisting of two transactions (T1 & T2))

3. U3 Algorithm

The basic steps taken by the U3 algorithm are presented in the flowchart of Figure 2, and these are explained in detail later in this section. The tree database is first transformed into a database of rooted integer-labeled trees. It is then ordered into its CF to reduce the average number of generated candidate subtrees that need to be ordered. Recursive List (RL) is constructed which is a global sequence of encountered nodes in the pre-order traversal that stores necessary node information. During this process the set of frequent 1-subtrees (*F1*) is obtained by hashing the encountered node labels. TMG candidate generation using the *RL* structure takes place and the string representations of candidate subtrees are hashed to the Ck hash table and the right-most path (*RMP*) occurrence coordinates are stored. Prior to hashing the string representation of each candidate subtree, it is first ordered into its CF, if necessary. The process repeats until all frequent k-subtrees are enumerated. We next explain the way that trees are represented at

the implementation level and how the DFCF ordering scheme can be re-formulized with respect to this representation.

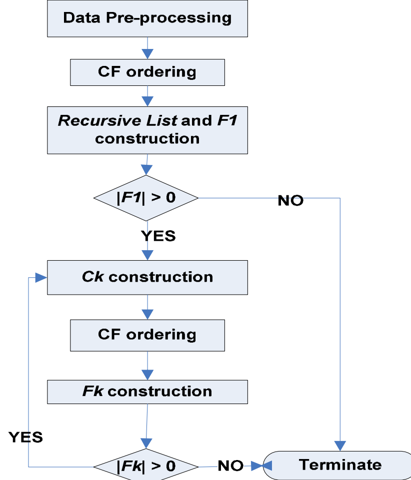


Figure 2. Steps taken by U3 algorithm

Tree Representation and Canonical Form Ordering. A tree is represented using the pre-order string encoding (φ) as described in [2, 4]. It is a sequential representation of the nodes of a tree as encountered during the pre-order traversal of that tree. The backtrack symbol ('/') is used whenever moving up a node in the tree during the pre-order traversal. We denote encoding of a subtree T as *φ(T)* and for example from Figure 1, *φ(T1):* 'a b c c / e / / e / / c / d / e e / / '. The backtrack symbols can be omitted after the last node. We refer to a group of subtrees with the same encoding *L* as candidate subtrees C_L. The canonical form ordering occurs at the start where the whole tree database is ordered into its canonical form and later where candidate subtrees are ordered so that unordered subtrees are correctly enumerated. The DFCF ordering scheme can now be reformulated with respect to the tree representation used.

Given two trees T1 and T2, with root[T1] = r1 and root[T2] = r2, let C(r1) and C(r2) denote the children sets of r1 and r2, respectively. Further, let $\varphi(Tx)_k$ denote the kth element of the pre-order string encoding of tree Tx (x = 1 or 2)(this can be either a node label or the special backtrack ('/') symbol which, as mentioned earlier, is considered smaller than any other label). T1 is considered smaller than T2 iff either:

a.) L(r1) < L(r2), or
b.) L(r1) = L(r2) and either size(C(r1)) < size(C(r2)) and $\varphi(T1)_k = \varphi(T2)_k$ for all $1 \le k \le$ length(φ(T1)), or $\varphi(T1)_k < \varphi(T2)_k$ for some $1 \le k <$ length(φ(T1)).

Recursive List (*RL*) and F1 Construction. The tree database, T_{db}, is scanned once to create the global pre-order sequence *RL* in memory, which provides shared global nodes' related information that can be directly accessed. RL stores each node in T_{db} following the pre-order traversal indexing. *Position*, *label*, *scope*, and *level* information are stored for each node. The level of a node refers here to the level in the T_{db} tree, where this node occurs. An item in the RL at position *i* is referred to as *D[i]*. Every time a node is inserted into the *RL*, we generate a candidate 1-subtree. Based on its label, we increment its support count in the C_1 hash table. If its support count is ≥ σ (user-specified minimum support count), we insert the candidate 1-subtree to the frequent 1-subtree set, F_1. An example RL structure representing the tree T1 from Figure 1 is displayed in Figure 3. The pre-order position of a node in the tree database is equal to the index of the *RL* at which that nodes is stored, and the label, scope and level are shown in that order underneath the entry. All this information is necessary to enumerate only valid subtree candidates and is accessed in the TMG candidate enumeration process explained next.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
a, 9,0	b,5,1	c,4,2	c,3,3	e,4,3	e,5,2	c,6,1	d,7,1	e,9,1	e,9,2

Figure 3. Recursive List representation of tree T1

A particular subtree, as defined by its encoding can be found at many places in the database and these different occurrences need to be stored so that subsequent set of candidates can be generated. We only store the occurrence coordinates of the nodes in the right-most path of the subtree (referred to as *RMP-oc*). Within our framework, this information is sufficient for enumerating candidate *(k+1)*-subtrees from a frequent *k*-subtree. Given a k-subtree *T* with *oc* $[e_0,e_1,\ldots e_{k-1}]$, the *RMP-oc* of *T*, denoted by *Ψ(T)*, is defined by $[e_0,e_1,\ldots,e_j]$ such that $\Psi(T) \subseteq oc(T)$; $e_j = e_{k-1}$; and $j \le k$-1 and the path from e_j to e_0 is the *RMP* of tree *T*. Vertical Occurrence List (*VOL*) [1,3] is used to store all *Ψ(T)* of a subtree *T* represented by its pre-order string encoding *φ(T),* and to determine the occurrence-match and transaction-based support. A transaction identifier (*tid*) is stored for each *Ψ(T)* so that the occurrence match of *T* equals to |*VOL*| while the transaction-based support equals to the number of unique *tid*s in *VOL*.

TMG Candidate Subtree Generation (Ck and Fk construction). TMG is a specialization of the right most path extension method which has been reported to be complete and non-redundant [2, 4]. To enumerate all embedded k-subtrees from a (*k-1*)-subtree, the TMG enumeration approach extends one node at a time for each node in the *RMP* of a (*k-1*)-subtree. Hence, it is a breadth-first (*BF*) enumeration strategy. Suppose that nodes in the *RMP* of a subtree are defined as *extension points*. The TMG can be

formulated as follows. Let $\Psi(T_{k-1}):[e_0,e_1,...e_j]$ denote the *RMP-oc* of a frequent (k-1)-subtree T_{k-1}, and Φ the *scope* of the root node e_0. TMG generates k-subtrees by extending each extension point $n \in \Psi(T_{k-1})$ with a node with *oc t* iff $n < t \leq \Phi$. Suppose that the encoding of T_{k-1} is denoted by $\varphi(T_{k-1})$ and $l(e_j,t)$ is a labeling function for extending extension point *n* with a node at position *t*. $\varphi(T_k)$ would be defined as $\varphi(T_{k-1})+l(e_j,t)$, where $l(e_j,t)$ determines the number of backtrack symbols '/' to be appended before the label of the new node is added to $\varphi(T_k)$. The number of backtrack symbols is calculated as the shortest path length between the extension point *n* and the right-most-node *r,* (notation *pl(n,r)*). To generate RMP at each step of candidate generation, we utilize the computed number of backtrack symbols *b* that need to be appended before the new node with *oc t* is added to the encoding. Given that the $\Psi(T_{k-1})$ is $[e_0,e_1,...,e_j]$, the RMP of the k-subtree ($\Psi(T_k)$) is generated by appending *t* at position $(j+1) - b$ of the ($\Psi(T_{k-1})$) and removing any *RMP-oc* that occur after *t*, thereby making *t* the right most node of T_k. This will make sure that at each extension of (k-1)-subtree, *RMP-oc* of k-subtree are appropriately stored.

To provide an illustrative example let us say that we are extending the T_{k-1} subtree from Figure 4, where $\Psi(T_{k-1}):[0,4,5]$, $\varphi(T_{k-1})$:'a b / b c', and right-most-node 'c' (*oc*:5). If we are extending T_{k-1} from extension point node 'b' (*oc*:4) with node 'e' (*oc*:8) then *l(5,8)* will append one backtrack symbol (*pl(4,5)* = 1) and the label 'e' to $\varphi(T_{k-1})$. The new encoding $\varphi(T_k)$ becomes 'a b / b c / e', and $\Psi(T_k):[0,4,8]$ (i.e. inserting 8 at position (j+1) - b = (3+1) - 1 = 3 of $\Psi(T_{k-1})$).

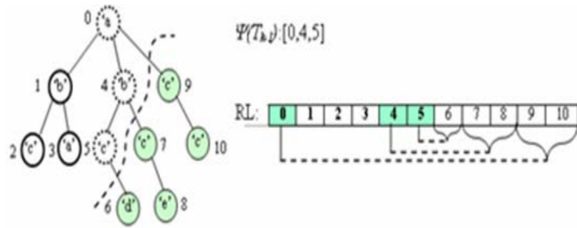


Figure 4. TMG enumeration: extending (k-1)-subtree T_{k-1} ($\varphi(T_{k-1})$:'a b / b c' (oc:0145)) with nodes at positions 6, 7, 8, 9, and 10

In the case of unordered subtrees, the right-most-node may not always correspond to the last node (tail position) in the encoding as it does for the ordered subtree case. We refer to this case as *non-tail expansion*. A notion of pivot position ς is used to denote the position in the subtree encoding that corresponds to the right-most-node. Each RMP-OC of a subtree will store an integer indicating the pivot position ς in the encoding for that particular occurrence of the subtree. Hence, for a *non-tail expansion* of a subtree T_{k-1}, if we are appending a new node with label *l* and *oc t*, rather than appending the backtrack symbols (if any) and *l* to the last node in $\varphi(T_{k-1})$, it will be appended to the pivot position ς by the function $l(\varsigma, t)$, in order to obtain $\varphi(T_k)$. Please note that if there are *b* backtrack symbols to be appended with *l* and there were already some backtrack symbols after the pivot position ς in $\varphi(T_{k-1})$, then *l* will be appended after the bth backtrack symbol. Furthermore, an additional backtrack symbol will be appended after the position in the encoding where *l* has been appended. To illustrate this please consider the subtree *st2* from T2 in Figure 1, with *oc*:05674, $\Psi(st2):[0,5,7]$ *and* $\varphi(st2)$:'a c c / e / / e' (note that *oc* of *st2* is different to the one displayed in Figure 1 because it is ordered according to the DFCF explained earlier. As can be seen the right-most node does not correspond to the last node 'e' in the encoding with *oc*:4, but rather to node 'e' with *oc*:7. Therefore, if we are extending *st2* from extension point node 'e' (*oc*:7) with node 'c' (*oc*:9) then *l(7,9)* will append the label 'c' to $\varphi(st2)$ at pivot position ς and add '/' after 'c'. The new encoding becomes 'a c c / e c / / / e'.

<u>Avoiding CF ordering</u>. 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 k-subtree enumerations are performed for $k > 2$. Hence, within the *CF* ordering scheme 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 was experimentally demonstrated in [3].

<u>Pruning</u>**.** To make sure that all generated subtrees do not contain infrequent subtrees, full (k-1) pruning [7,1,2] must be performed. This implies that at most (k-1) numbers of (k-1)-subtrees need to be generated from the currently expanding k-subtree. When the removal of root node of k-subtree doesn't generate a *forest* [2, 4] then an additional (k-1)-subtree is generated by taking the root node off from the

expanding k-subtree. The expanding k-subtree is pruned when at least one (k-1)-subtree is infrequent. This ensures that the downward-closure lemma [10] is satisfied and in the case of occurrence match support no pseudo-frequent subtrees will be generated. The whole TMG candidate enumeration process repeats until all frequent k-subtrees are generated.

4. Related Works

A number of algorithms have been developed for the problem of unordered subtree mining. The Unot algorithm [11] uses a reverse search technique for incremental computation of unordered subtree occurrences. Nijssen and Kok [12] present a bottom-up strategy for determining the frequency of unordered induced subtrees. Breadth-first canonical form (BFCF) and depth-first canonical form (DFCF) for labeled rooted unordered trees has been presented in [8]. In the same work the authors proposed two algorithms: RootedTreeMiner, which is the authors' re-implementation of Unot (a vertical mining algorithm based upon BFCF) and FreeTreeMiner, which extends the DFCF for discovering labeled free trees. As an extension to the work, HybridTreeMiner [13] is an efficient algorithm that systematically enumerates all subtrees by traversing an enumeration tree which is defined based upon the BFCF for unordered subtrees. All these algorithm mine induced subtree while the SLEUTH [7] algorithm extract all frequent unordered embedded subtrees by using unordered scope-list joins via the descendant and cousin tests. Another algorithm for mining frequent embedded unordered subtrees is TreeFinder [14] that uses an Inductive Logic Programming approach, but which in the process can miss many frequent subtrees. More recently, a related problem of mining maximal embedded unordered subtrees has been addressed in [15]. A maximal pattern is a frequent pattern in which no proper superset is frequent.

5. Experimental Results

Our U3 algorithm is evaluated by comparing it with the SLEUTH [7] algorithm. Since the approach described in [15] extracts maximal patterns rather than the complete set of frequent patterns, it is not as suitable for comparison. For transaction based support our algorithm is preceded with 'T-' (e.g. T-U3). When no full (k-1) pruning is performed (NP) is added at the end (e.g. U3(NP)). TreeGenerator [2] is used to obtain the artificial tree databases. We also use a reduced version (54%) of CSLogs real world data set [2] because when the full dataset is used all of the compared algorithms fail to return any results for a reasonable support threshold. The minimum support σ is denoted as (sxx), where xx is the minimum frequency threshold. Experiments were run on 3Ghz (Intel-CPU), 2Gb RAM, Mandrake 10.2 Linux machine.

Time Performance Test (CSlogs)**.** As can be seen in Figure 5, the T-U3 algorithm enjoys better time performance when compared to SLEUTH which has some performance issues for decreasing support value. We refrained from running it further for lower support thresholds (s100 and s150) since there was already such a large jump in performance from s205 to s200. The time taken would be too long and recording the result would increase the scale of the y-axis from the graph and thereby decrease the clarity of the displayed results. However, we show that the T-U3 algorithm still performed reasonable well for s100 and s50. By not performing full pruning there was an additional performance gain by T-U3(NP). The performance gain firstly comes from avoiding the generation of all possible k-1 subtrees, and secondly because canonical form transformations do not need to be performed for all the (k-1) subtrees of a potentially frequent k-subtree.

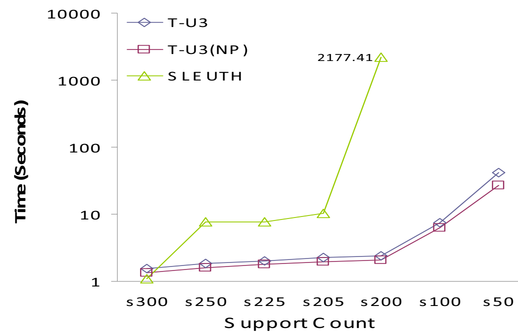
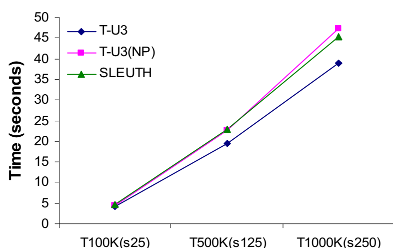


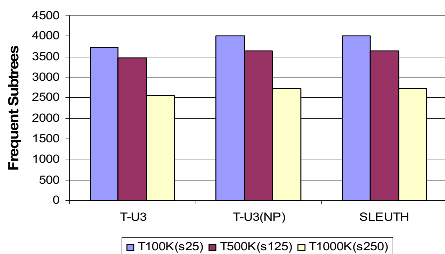
Figure 5. Time performance test on CSLogs data

Scalability Test. We used a synthetic datasets that consists of 10,000 items, with average depth and fan-out of 40. The number of transactions used was 500K, 1M, 2M, and the respective support threshold was 250, 500 and 1000. All the tested algorithms are well scalable for the different dataset sizes (Figure 6(a)). The time performance is comparable among the algorithms with T-U3 performing slightly better than others. In Figure 6(b), we show the total number of frequent subtrees returned by each algorithm. The result indicates that SLEUTH returns the same number of frequent subtrees as T-U3(NP) which hints that SLEUTH does not perform full (k-1) pruning but rather adapts an opportunistic approach as done by

VTreeMiner [2] algorithm for mining ordered embedded subtrees.



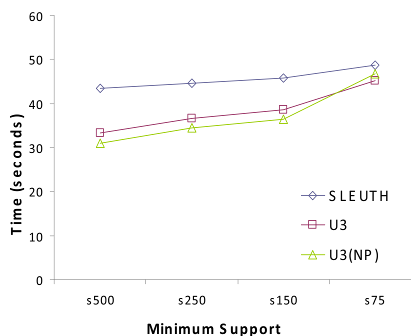
(a) Time performance



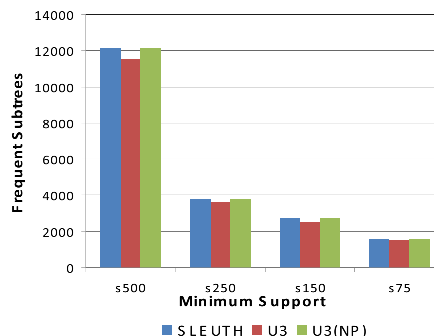
(b) Number of frequent subtrees reported

Figure 6. Scalability test

Occurrence Match Support Testing. An artificial tree data set was used with an average depth and fan-out of 40, and 1000K transactions. U3 algorithm has a better time performance when compared to the SLEUTH algorithm (Figure 7(a)), which generates extra candidate subtrees as frequent (Figure 7(b)). Whenever the minimum support is set to 75, we can see that both U3 variants start to degrade. By analyzing the process we notice that both U3 variants get close to reaching the system memory capacity at which point they start to use the machine virtual memory. Whenever a program uses virtual memory, it swaps data back and forth between the main memory and the secondary storage, which can significantly slow down any kind of processing. The SLEUTH on the other hand conserves the system memory better as it utilizes the depth-first (DF) enumeration approach. A DF enumeration will generate all different length candidate subtrees from each transaction completely before moving to the next transaction and the information about that transaction can be removed from memory. In contrast, the breadth-first (BF) enumeration strategy will need to store the occurrence coordinates of generated (k)-subtrees which is later used for generating (k+1)-subtrees from the same transaction. However, when generating a k-subtree using the DF enumeration method, information regarding the frequency of its (k-1)-subtrees may not be available at that time, whereas with the BF approach the frequency of all its (k-1)-subtrees has been determined. Therefore, full pruning can be done in a more complete way in the BF approach than in the DF approach which is forced to employ an *opportunistic pruning* [2] strategy that only prunes infrequent subtrees in an opportunistic way. In many cases as shown by other experiments, BF enumeration employed by the U3 can be very effective and more efficient whenever the memory space is not an issue.



(a) Time performance



(b) Number of frequent subtrees reported

Figure 7. Occurrence Match Support test

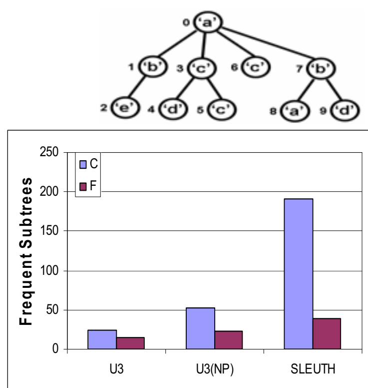
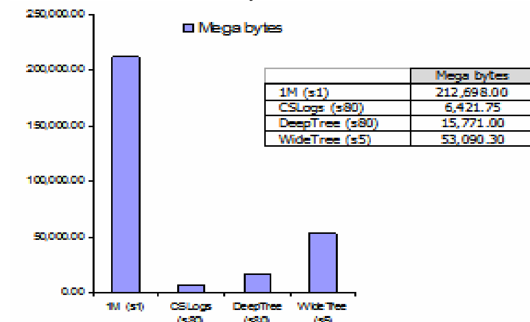


Figure 8. Number of subtrees reported for the example tree at $\sigma_{oc} = 2$

Ck and Fk generation. A simple dataset was developed that represents the tree displayed at the top of Figure 8 above. The aim is to show how the number of candidate and frequent subtrees enumerated by different approaches can vary even for such a simple dataset. As can be seen from the chart of Figure 7,

SLEUTH enumerates many more candidate subtrees due to the employed join approach which enumerates additional invalid subtrees. They are invalid in the sense that they do not conform to the underlying model of the tree structure, whereas the TMG approach ensures valid candidate subtrees [4, 5]. Due to the opportunistic pruning approach employed in SLEUTH, many more subtrees are considered as frequent in comparison to U3 (Figure 8). As mentioned in Section 2 we refer to these additional subtrees as pseudo-frequent subtrees. An example would be the subtree with encoding 'a b / c d' since it occurs twice in the tree (*oc*:0134 and *oc*:0734), whereas its infrequent k-1 subtree with encoding 'a c d' occurs only once (*oc*:034). Furthermore, to our surprise the number of candidate and frequent subtrees enumerated differs between U3(NP) and SLEUTH. When analyzing the extracted frequent subtrees it was noticed that SLEUTH wrongly considers some subtrees as frequent (eg. 'a b / b', 'a d / d'). This error is then further propagated during the candidate enumeration which explains the large number of candidate and frequent subtrees enumerated by SLEUTH.



	Mega bytes
1M (s1)	212,698.00
CSLogs (s80)	6,421.75
DeepTree (s80)	15,771.00
WideTree (s5)	53,090.30

Figure 9. Total memory saved by using Recursive List structure for different datasets

Memory saving through RL implementation. The aim of this experiment is to demonstrate the total amount of memory saved by utilizing the RL structure. In Figure 9 we show the additional memory required by our previous approach [6] which utilized the Embedding List structure. This at the same time corresponds to the total memory saved by the U3 algorithm. The 1M dataset is a synthetic data set consisting of 1M transactions.

6. Conclusions

An extension of our general tree mining framework for the capability of mining unordered embedded subtrees was presented. The flexibility of our general TMG approach is demonstrated through high performance and scalability when experimentally compared to the current state-of-the-art algorithm SLEUTH. The U3 algorithm has the capability of using the transaction support as well as the more complex occurrence match support, which as experimentally demonstrated can be a limitation for SLEUTH.

7. References

[1] Punin, J., Krishnamoorthy, M., and Zaki, M., 2001. "LOGML: Log markup language for web usage mining", In *ACM SIGKDD Workshop on Mining Log Data Across All Customer Touch Points*, August, San Francisco, CA.
[2] Zaki, M. J. Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. *IEEE Transactions on Knowledge & Data Engineering*, 17:8, pp. 1021-1035, 2005.
[3] Hadzic, F., Dillon, T. S., Sidhu, A., Chang, E, and Tan, H., "Mining Substructures in Protein Data", keynote paper in *IEEE ICDM 2006 Workshop on Data Mining in Bioinformatics*, 18-22 December, Hong Kong, 2006.
[4] Tan, H., Dillon, T.S., Hadzic, F., Feng, L., Chang, E., "IMB3-Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding", *PAKDD'06,* Singapore, 2006.
[5] Tan, H., Hadzic, F., Dillon, T.S., Feng, L., Chang, E., "Tree Model Guided Candidate Generation for Mining Frequent Subtrees from XML", To appear in *ACM Transactions on Knowledge Discovery from Data*, 2007.
[6] Hadzic, F., Tan, H., and Dillon, T.S. UNI3: efficient algorithm for mining unordered induced subtrees using TMG candidate generation. *CIDM 2007*, Honolulu, Hawaii, 2007.
[7] Zaki, M. J. Efficiently Mining Frequent Embedded Unordered Trees. *Fundamenta Informaticae 65*, IOS Press, pp. 1-20, 2005.
[8] Chi, Y., Yirong, Y. and Muntz, R. R. Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining, *Knowledge and Information Systems*, 2004.
[9] Valentine, G. *Algorithms on Trees and Graphs*, Springer-Verlag, Berlin, 2002.
[10] Agrawal, R. and Srikant, R. Fast algorithms for mining association rules. *20th Int'l Conf. on Very Large Data Bases (VLDB 1994)*, Santiago de Chile, Chile, 1994, pp. 487-499.
[11] Asai, T., Arimura, H., Uno, T. and Nakano, S. Discovering Frequent Substructures in Large Unordered Trees. *6th Int'l Conf. on Discovery Science*, 2003.
[12] Nijssen, S. and Kok, J. N. Efficient discovery of frequent unordered trees. *Int'l Workshop on Mining Graphs, Trees, and Sequences*, Dubrovnik, Croatia, 2003.
[13] Chi, Y., Yang, Y., and Muntz, R. R. HybridTreeMiner: An efficient algorihtm for mining frequent rooted trees and free trees using canonical forms. In *Proc. of the 16th Int'l Conf. on Scientific and Statistical Database Management*, Santorini Island, Greece, 2004.
[14] Termier, A., Rousset, M-C. and Sebag, M. Treefinder: A First Step Towards XML Data Mining. In *Proc. of IEEE ICDM'02*, 2002.
[15] Chehreghani, M. H., Rahgozar, M., Lucas, C. and Chehreghani, M. H. Mining Maximal Embedded Unordered Tree Patterns. *CIDM 2007*, Honolulu, Hawaii, April 1-5, 2007.