MB3-Miner: mining eMBedded subTREEs using Tree Model Guided candidate generation

Henry Tan[1], Tharam S. Dillon[1], Fedja Hadzic[1], Ling Feng[2] and Elizabeth Chang[3]
[1]*Faculty of Information Technology, University of Technology Sydney, Australia*
e-mail: (henryws, tharam, fhadzic)@it.uts.edu.au
[2]*University of Twente, Netherlands*
e-mail: ling@cs.utwente.nl
[3]*School of Information Systems, Curtin University of Technology Perth, Australia*
e-mail: Elizabeth.Chang@cbs.curtin.edu.au

Abstract

Tree mining has many useful applications in areas such as Bioinformatics, XML mining, Web mining, etc. In general, most of the formally represented information in these domains is a tree structured form. In this paper we focus on mining frequent embedded subtrees from databases of rooted labeled ordered subtrees. We propose a novel and unique embedding list representation that is suitable for describing embedded subtrees. This representation is completely different from the string-like or conventional adjacency list representation previously utilized for trees. We present the mathematical model of a breadth-first-search Tree Model Guided (TMG) candidate generation approach previously introduced in [8]. The key characteristic of the TMG approach is that it enumerates fewer candidates by ensuring that only valid candidates that conform to the structural aspects of the data are generated as opposed to the join approach. Our experiments with both synthetic and real-life datasets provide comparisons against one of the state-of-the-art algorithms, TreeMiner [15], and they demonstrate the effectiveness and the efficiency of the technique.

Keywords: treeminer, tree mining, frequent tree mining, embedded subtree,.

1. Introduction

Research in both theory and applications of data mining is expanding driven by a need to consider more complex structures, relationships and semantics expressed in the data [3,4,7,8,12,13,15]. As the complexity of the structures to be discovered increases, more informative patterns could be extracted [13]. Tree as a special type of graph has attracted considerable amount of interest [2,3,4,8,9,10,11,12,13,15]. Recently, XML has become very popular. There is a growing number of XML-enabled multimedia data encoding. VoiceXML [5] is developed to enable the rapid deployment of voice applications. X3D [6] is used to enable real-time communication of 3D data. Ling et. al. [4] has initiated an XML-enabled association rule framework. It extends the notion of associated items to XML fragments to present associations among trees. Tan et. al. [8] suggested that one of the important XML mining tasks is to mine frequent tree patterns in a collection of XML documents. Wang and Liu [11] developed an algorithm to mine frequently occurring induced subtrees in XML documents. In general, most of the formally represented information in these domains is a tree structured form. Frequent structure mining (FSM) refers to an important aspect of mining that deals with pattern extraction in massive databases and involves discovering frequent complex structures [15].

Frequent subtree mining as one instance of FSM has many useful applications in areas such as Bioinformatics, XML mining, Web mining, etc. One practical application of frequent subtree mining in the above areas is to discover associations between data entities in tree structured form. This is known as association mining which consists of two important problems, i.e. frequent itemset discovery and rule construction. The former task is considered to be a more difficult problem to solve than the latter. Our study is mainly focused on developing an efficient approach for frequent embedded subtrees discovery from a database of rooted ordered labeled subtrees.

We propose a novel and unique embedding list representation that is suitable for describing embedded subtrees. This representation is adapted from the conventional adjacency list format, by relaxing the adjacency notion into an embedding notion in order to capture the embedding relationships between nodes in trees. The list not only contains adjacent nodes (children), but also takes all its descendants into an embedded list in pre-order ordering. For speed considerations, each embedded list is represented in a string-like format so that each item in the list can be accessed in O(1) time. This representation is completely different from the string-like [2,3,8,10,12,15] or adjacency list [7] representation utilized previously for trees. There are different strategies for efficient candidate generation such as join and enumeration by extension [3]. An idea of utilizing XML schema to guide the candidate generation appeared in [12]. The approach generates

candidates that conform to the schema. Recently, Tree Model Guided (TMG) candidate generation for mining embedded rooted ordered subtrees is proposed in [8]. TMG generalizes the concept of schema guided into tree model guided, so that the enumeration model can be applied to any data in tree structure form, especially for enumeration of embedded subtrees. This non-redundant systematic enumeration model ensures only valid candidates are generated which conform to the actual tree structure of the data. An example of a tree model would be the structural aspects of a document in an XML schema, and a valid candidate would conform to this. In general, the TMG would be applicable to any area with structural models with clearly defined semantics that have tree like structures. Contrary to the extension method used for relational data, TMG generates fewer candidates as opposed to the join approach [1,3,8,15]. Even though fewer candidates are generated, the TMG enumeration ensures the generation of a complete candidate set. Thus, TMG approach is an optimal enumeration model for mining embedded, rooted and ordered subtrees. This is in contrast to an incomplete method TreeFinder [9] that can miss many frequent subtrees, especially when the support is lowered or when different trees in the database have common node labels. Independently, XSpanner [10] extends the Pattern-Growth concept into tree structured data and its enumeration model also generates only valid candidates.

The occurrences of candidate subtrees need to be counted in order to determine if they are frequent whilst the infrequent ones would be pruned. As the number of candidates to be counted can be enormous, an efficient and fast counting approach is extremely important. Efficiency of candidate counting is highly determined by the data structure used. More conventional approaches use a direct checking approach. For each candidate generated its frequency is increased by one if it exists in the transaction. A Hash-tree [1,3] data structure can be used to accelerate direct checking. Another approach projects each candidate generated into a vertical representation [8,14,15], which associates an occurrence list with each candidate subtree. If *transaction based support* [3,15] is used, the vertical format will consist of transaction IDs of the transactions that support it. In contrast, if *occurrence match* [3,8] or *weighted support* definition [15] is used each list will correspond to each candidate occurrence in the whole database of trees [8]. Occurrence match support takes repetition of items in a transaction into account whilst transaction based support only checks for existence of items in a transaction. With the vertical representation approach the frequency of a candidate subtree corresponds to the size of the occurrence list. With the advantage of being able to determine the support count of each candidate directly the vertical format has been reported to be faster than the direct checking approach [14,15]. In this paper, we improved over the vertical list format as previously utilized in our earlier approach [8] in two ways. First, the performance is expedited by storing only the hyperlinks [10] of subtrees in the tree database instead of creating a local copy for each generated subtree. The format is different than the scope-list [15] representation as our vertical list does not store any scope information. Secondly, we transform and map the tree structure data into integer indexes as opposed to processing time consuming string labels directly. Representing labels as integer opposed to string labels has performance and space advantages. For instance a node with label '123' would require 3 bytes of characters if its label is represented as a string, whereas only 1 byte is required if it is represented as an integer. Therefore, when a hashtable is used for candidate frequency counting, hashing integer labels over string labels can have significant impact on the overall candidates counting performance.

Our experiments with both synthetic and real-life data sets provide comparisons against one of the state of the art algorithms, TreeMiner [15], and they demonstrate the effectiveness and efficiency of the technique. The paper is organized as follows. In section 2 the problem decomposition is given. Section 3 describes the details of the algorithm. The mathematical model of TMG approach is provided in section 4. We empirically evaluate the performance of the algorithms and study their scale-up properties in section 5, and the paper is concluded in section 6.

2. Problem Definitions

General tree concepts and definitions. A tree is an acyclic connected graph with one node defined as the root. 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. A *root* is the topmost node in the tree. In labeled tree, there is a labeling function mapping vertices to a set of labels so that a label can be shared among many vertices. *Parent* of node v is defined as the predecessor of node v. There is only one parent for each v in the tree. A node v can have one or more *children* which are defined as its successors. A node without any child is a *leaf* node; otherwise, it is an *internal* node. If for each internal node, all the children are ordered, then the tree is an *ordered tree*. In an ordered tree, the *rightmost* child is referred to as the last child. The number of children of a node is commonly termed as *fan-out*/*degree* of the node. A path from vertex v_i to v_j, is defined as a 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 a path exists from node p to node q, then p is an *ancestor* of q and q is a *descendant* of p. *Height* of a node is length of path from that node to its furthest leaf. The *rightmost path* of T is defined as the path connecting the *rightmost leaf* with the root node. Height of a tree is defined as height of its root node. *Depth/level* of a node is the length of the path from

root to that node. The *size* of a tree is determined by the number of nodes in the tree. *Uniform tree T(n,r)* is a tree with height equal to *n* and all of its internal nodes have degree *r*. *Closed form* of an arbitrary tree is defined as a uniform tree with degree equal to the maximum degree of internal nodes in the arbitrary tree. In this paper, all trees we consider are ordered, labeled, and rooted trees. We are concerned with mining embedded subtree. An *embedded subtree* [3,8,10,15] is a generalization of *induced subtree* [2,3], where parent-child as well as ancestor-descendant relationships are preserved. By extracting embedded subtrees, patterns hidden deeply within large tree structures can be found. Mining embedded subtrees is more complex than mining induced subtrees, as induced subtree $\subseteq$ embedded subtree [3]. In figure 1, T2 and T4 are examples of induced subtrees of T while T1-4 are examples of embedded subtrees of T. In case of induced subtrees T2 and T4, only the parent-child relationship of each node is preserved while for embedded subtrees T1-4 the ancestor-descendant relationship is also preserved. For each node in T (figure 1), its label is shown as a single-quoted symbol inside the circle whereas its position is shown as indexes at the left/right side of the circle.

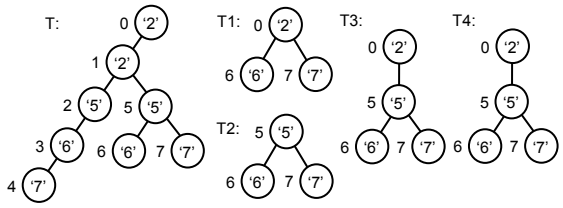


Figure 1. Example of induced (T2, T4) and embedded (T1, T3) subtrees

String encoding (φ). In a database of labeled subtrees, many subtrees can have the same *string encoding (φ)* [2,8,10,15]. We denote encoding of subtree *T* as *φ(T)*. From figure 1, *φ(T1)*:'2 6 / 7 /'; *φ(T3)*:'2 5 6 / 7 /', etc. We could omit backtrack symbols after the last node, i.e. *φ(T1)*:'2 6 / 7'. We refer to a group of subtrees with the same encoding *L* as *candidate subtree* C_L. A subtree with k number of nodes is denoted as *k-subtree*. Throughout the paper, the '+' operator is used to conceptualize an operation of appending two or more tree encodings. However, this operator should be contrasted with the conventional string append operator, as in the encoding used the backtrack symbols need to be computed accordingly.

Mining frequent subtrees. Given that T_{db} is a tree database consisting of *N* transactions of trees, K_N. The task of frequent subtree mining from T_{db} with given minimum support (σ), is to find all the candidate subtrees that occur at least σ times in T_{db}. Unless otherwise stated, occurrence match/weighted support definition is used [3,8,15]. Based on the downward-closure lemma [1], every sub-pattern of a frequent pattern is also frequent. In relational data, given a frequent itemset all its subsets are also frequent. A question however arises if whether the same principle applies to tree structure data when the occurrence match support definition is used? To show that the same principle doesn't apply, we need to find a counter-example where the relation doesn't hold for tree structure data.

Lemma 1. Given a tree database T_{db}. If there exists candidate subtree C_L and $C_{L'}$, where $C_L \subseteq C_{L'}$, such that $C_{L'}$ is frequent and C_L is infrequent, we say that $C_{L'}$ is a *pseudo-frequent candidate subtree*.

Lemma 2. Given an infrequent candidate subtree C_L and a subtree *T* where *φ(T):L*. $V_{rm}(T)$ is a set of nodes in rightmost path of *T*. A pseudo-frequent candidate subtree $C_{L'}$ with support *m* where *L':L+l* can be generated from *T* by attaching *m* number of children with the same encoding *l* to any node$\in V_{rm}(T)$ and $m \geq$ minimum support σ such that there are *m* number of subtrees $T_1,\ldots,T_m$ with encoding *L'* and $T \subseteq T_1,\ldots,T_m$.

Lemma 3. Given an infrequent embedded candidate subtree C_L and a subtree *T* with root node v_r where *φ(T):L*. $v_1,\ldots,v_m$ is a set of nodes with the same encoding *l* where v_1 is a parent of v_2, v_{m-1} is a parent of v_m and $m \geq$ minimum support σ. A pseudo-frequent candidate subtree $C_{L'}$ with support *m* where *L':l+L* can be obtained by connecting $v_1,\ldots,v_m$ to *T* such that there is a path with length ≥ 1 from v_m *to* v_r and *m* numbers of embedded subtrees $T_1,\ldots,T_m$ with encoding *L'* can be generated where $T \subseteq T_1,\ldots,T_m$.

Theorem 1. Antimonotone property of frequent patterns suggests that the frequency of a superpattern is less than or equal to the frequency of a subpattern. Lemma 2 and 3 say that there can be pseudo-frequent candidate subtrees generated from an infrequent subtree. Thus, antimonotone property does not always hold in frequent subtrees mining when occurrence match support is considered.

In the light of downward closure lemma, the pseudo-frequent candidate subtrees are equivalent to infrequent candidate subtrees. From figure 1, if σ is set to 2, subtrees with encoding '2 5 6 / 7' are examples of pseudo-frequent candidate subtrees. Although support of '2 5 6 / 7' is 2, it is a pseudo-frequent candidate subtree since '5 6 / 7' is infrequent. Thus, '2 5 6 / 7' should be pruned.

3. MB3-Miner Algorithm

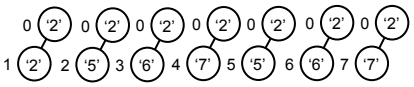
3.1. Generating Candidate Subtrees

We are concerned with a systematic way of generating candidate subtrees. An optimal enumeration method should generate each subtree at most once and only generate valid candidates according to the tree model. It should also be complete, i.e. it generates all possible candidate subtrees from a given database of trees. Our candidate generation approach utilizes embedded list representation to guide the enumeration of embedded subtrees.

Database scanning. The process of frequent subtree mining is initiated by scanning a tree database, T_{db}, and

generating a global sequence *D* in memory. We refer to this sequence as a *dictionary*. The dictionary consists of each node in T_{db} following the pre-order traversal indexing. For each node its position, label, right-most descendant position (scope) [8,10,15], and parent position is stored. Thus each dictionary item is defined as a tuple of position (pos), label (l), scope (s), parent (p), {pos, l, s, p}. We refer to an item in the dictionary at position *i* as *dictionary[i]*. Unless otherwise mentioned, the notion of position of an item refers to its index position in the dictionary. At the same time, when generating the dictionary, we compute all the frequent 1-subtrees, F_1. After the in-memory database (*dictionary*) is constructed our approach does not require further database scanning.

Constructing Embedding List (EL). For each frequent internal node in F_1, a list is generated which stores its descendant nodes' hyperlinks [10] in pre-order traversal ordering such that the embedding relationships between nodes are preserved. The notion of hyperlinks of nodes refers here to the positions of node in the dictionary. For a given internal node at position *i*, such ordering reflects the enumeration sequence of generating 2-subtree candidates rooted at *i* (see figure 2). Thus, the EL construction is equivalent to the process of enumerating all 2-subtree candidates from a database of trees. Hereafter, we call this list as *embedded list (EL)*. As there can be more than one ELs, we use notation *i-EL* to refer to an embedded list of node at position *i*. Position of an item in EL is referred to as *slot* as opposed to *position* when referring item in the dictionary. Thus, i-EL[n] refers to the item in the list at *slot* n. Whereas |i-EL| refers to the size of the embedded list of node at position *i*. In figure 2, *0-EL* for example refers to the list: *0:[1,2,3,4,5,6,7]*, *0-EL[0]*=1 and *0-EL[6]*=7. Figure 2 illustrates an example of the EL representation of subtree T (figure 1) with encoding: '2 2 5 6 7 / / / 5 6 / 7'.



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

Figure 2. The EL representation of tree T in figure 1

Occurrence Coordinate (OC). We have adopted an extension by a single item at a time. When generating k-subtree candidates from (k-1)-subtree, we consider only frequent (k-1)-subtrees for extension. Each occurrence of k-subtree in T_{db} is encoded as *occurrence coordinate* $r:[e_1,\ldots e_{k-1}]$; *r* refers to k-subtree root position and $e_1,\ldots,e_{k-1}$ refer to slots in r-EL. Each e_i corresponds to node $(i+1)$ in k-subtree and $e_1 < e_{k-1}$. We refer to e_{k-1} as *tail* slot. From figure 1 and 2, the OC of 3-subtree (T2) with encoding '5 6 7' is encoded as 2:[0,1]; two 4-subtrees (T3 & T4) with encoding '2 5 6 / 7' are encoded as *0:[4,5,6]* and *1:[3,4,5]* respectively, and so on. Each OC of a subtree describes an instance of each occurrence of the subtree in T_{db}. Hence, each *candidate instance* should have an OC associated with it.

The scope of node. EL representation preserves the ordering as well as the embedding relationships of nodes in a tree. *i-EL* defines the scope of node *i* such that the scope of *i* spans from *i-EL[0]* to *i-EL[j]* where $j = |i\text{-}EL|-1$. We refer to the first scope position as the *leftmost scope* and the last scope position as the *rightmost scope*. Consequently, given a 4-subtree *T* with occurrence coordinate *1:[3,4,5]*, the leftmost scope of T is defined by *1-EL[3]* and the rightmost scope of *T* is defined by *1-EL[5]*. An occurrence coordinate of a valid candidate is defined by *r:[m,…n]* where m < n. Thus, a valid candidate has an increasing scope ordering such that *r-EL[m] < r-EL[n]*.

TMG enumeration formulation. An enumeration approach guided by tree model (TMG) was introduced in our previous work [8]. TMG enumeration approach extends a candidate (k-1)-subtree by one node at the time starting from the last node of its right most path up to its root. We have constructed EL in such a way that the pre-order ordering of the embedding relationship of nodes is preserved. As a corollary, given the tail position of a (k-1)-subtree the enumeration sequence provided by EL starts from the next slot after the tail to the end of the EL follows the correct right most extension ordering. Thus the TMG enumeration is formulated as follows. *l(i)* denotes a labeling function of node at position *i*. Given frequent (k-1)-subtree t_{k-1} with $\varphi(t_{k-1}):L$, the root position *r*, tail position *t*, left-most scope *m*, right-most scope *n*, and occurrence coordinate *r:[m,…,n]*, k-subtrees are generated by extending t_{k-1} with $j \in r\text{-}EL$ such that $t<j\leq|r\text{-}EL|-1$. Thus its occurrence coordinate becomes *r:[m,…,n,j]* and its encoding becomes *L':L+l(i)* where *i=r-EL[j]* and *m<n<j*.

Pruning. Apriori theory says that a pattern is frequent if and only if all of its sub-patterns are frequent. Theorem 1 suggests that this property doesn't always hold for tree structure, and as such a more specialized approach is needed when mining frequent subtrees. In mining frequent subtrees, this problem may occur because the semantics of a tree structure is determined by its values and hierarchical structure. Hence, each individual node may be frequent, but the structural relationships between the nodes are infrequent. The approach taken was to prune those candidates that have one or more infrequent subtrees. Thus, (k-1) full pruning [15] must be performed when generating k-subtrees. This implies that at most (k-1) numbers of (k-1)-subtrees need to be generated from the currently expanding k-subtrees. The expanding k-subtree is pruned when at least one (k-1)-subtree is infrequent, otherwise it is added to the frequent k-subtree set. This ensures that the method generates no pseudo-frequent subtrees and is correct as opposed to the opportunistic pruning utilized in DFS method such as VTreeMiner [15]. As for each k-subtree candidate there can be (k-1) checks involved for determining whether all its (k-1)-subtrees are frequent, the

process can be quite time consuming and expensive. Fortunately, some time is saved by checking whether a candidate is already a part of the frequent k-subtree set. This way if a (k-1)-subtree candidate is already in the frequent k-subtree set, it is known that all its subtrees are frequent, and hence the (k-1) full pruning can be accelerated as only 1 comparison is required.

3.2. Candidate Subtree counting

In the candidate enumeration step, the process utilized the notion of coordinates. To determine if a subtree is frequent, we count the occurrences of that subtree and check if it is greater or equal to the specified minimum support σ. In a database of labeled trees many instances of subtrees can occur with the same encoding. Hence, the notion of encoding is utilized in the candidate counting process. We say that a subtree has a frequency n if there are n instances of subtrees with same encoding, i.e. we group subtree occurrences by its encoding.

Vertical Occurrence List (VOL). Each occurrence of a subtree is stored as an occurrence coordinate as previously described. The vertical occurrence list of a subtree groups the occurrence coordinates of that subtree by its encoding. Hence, computing 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)*|. As an example, the frequency of a subtree of tree T with encoding '2 5 6', |*VOL('2 5 6')*| is equal to 4.

1	5	6
0	5	6
1	2	3
0	2	3
' 2	5	6 '

size : 4

Figure 3. VOL representation of subtree with encoding '2 5 6' of tree T in figure 1

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 following pseudo-code:

```
Inputs  : T_db (Tree database), σ (min. support)
Outputs : Frequent subtrees (F_k), D (dictionary)
{D, F_1}  = DatabaseScanning (T_db)
{EL, F_2} = ConstructEmbeddedList (F_1, D)
k=3
while( |F_k| ≥ 0 )
    F_k = GenerateCandidateSubtrees(F_k-1)
    k = k+1

GenerateCandidateSubtrees(F_k-1):
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)
    for each occurrence coordinate oc_k-1 (r:[m,…n]) ∈ VOL-t_k-1
        for (j = n+1 to |r-EL|-1 )
            {oc_k, L_k} = TMG-extend(oc_k-1, L_k-1, j)
            If( Contains(L_k, F_k) )
                Insert(h(L_k), oc_k, F_k)
            else
                If( k-1Pruning (L_k) == false)
                    Insert(h(L_k), oc_k, F_k)
return F_k
```

Figure 4. MB3-Miner algorithm pseudo-code

4. TMG Mathematical Model

In this section we will develop the mathematical model of TMG approach for mining embedded subtree. Such a model would allow us to calculate the worst case complexity of enumerating all possible candidates from data in a tree structure form. There is no simple way to parameterize a tree structure unless it is specified as a uniform tree. The size of a uniform tree follows a geometrical series. Thus, a size of uniform tree *T(n,r)* can be computed using geometrical series formula $(1-r^{n+1})/(1-r)$. Alternatively, when the root is omitted the following formula is used, $r(r^n-1)/(r-1)$. When r = 1, the size of the uniform tree is equal to its height n.

The complexity of enumeration using TMG approach is bounded by the actual tree structure. By definition of closed form of an arbitrary tree, the worst case scenario of enumerating candidates from an arbitrary tree is bounded by its closed form enumeration complexity. We have developed a knowledge representation called embedded list (EL) to represent any arbitrary tree and enumerate candidates using TMG approach in a systematic way. The TMG enumeration mathematical model is formulated as follows. Given a uniform tree *T* with height *n* and degree *r* the worst case complexity of candidate generation of *T* is expressed mathematically in term of its height *n* and degree *r*. We define that the cost of enumeration is expressed as the number of candidate instances enumerated throughout the candidate generation process as opposed to the number of candidate subtrees generated (section 2).

Complexity of 1-subtree & 2-subtree enumeration $\|T\|_1$ **&** $\|T\|_2$**.** The complexity of 1-subtree enumeration is equal to the size of the tree |T|. Previously we have developed a corollary that the construction of EL

representation from a database of trees reflects the enumeration sequence of generating 2-subtree candidates. From figure 2, the visualization of EL representation of tree T suggests that the number of generated candidate instances is equal to the sum of the size of the lists in EL. Let *s* be a set with n objects. Combinations of k objects from this set s ($_sC_k$) are subsets of *s* having *k* elements each (where the order of listing the elements does not distinguish two subsets). $_sC_k$ formula is given by s!/(s-k)!k!. Thus, for 2-subtrees enumeration the following relation exists. Let *r-EL* consist of *l* number of slots where each slot is denoted by *j*. The number of all generated valid 2-subtree candidates (*r:[j]*) rooted at *r* is equal to the number of combinations of *l* nodes from *r-EL* having 1 element each. As the corollary, complexity of 2-subtrees enumeration of tree T with size |T| is equal to the sum of all generated 2-subtree candidates from each node in T is given by eq 1 below.

$$\sum_{r=1}^{|T|} {}_{|r-EL|}C_1 \qquad \text{eq. 1}$$

Complexity of k-subtree enumeration $\|T\|_k$. The generalization of 2-subtrees enumeration complexity can be formulated as follows. Let *r-EL* consist of *l* number of items; each item is denoted by *j*. The number of all generated valid k-subtree candidates (*r:[e1,…,ek-1]*) rooted at *r* is equal to the number of combinations of *l* nodes from *i-EL* having (*k-1)* element each. In section 3, valid occurrence coordinate of valid candidates has the property that $e_1<e_{k-1}$. Thus all valid combinations have the (k-1) element in increasing order. As a corollary, the complexity of k-subtrees enumeration of tree T with size |T| is equal to the sum of all generated k-subtree candidates:

$$\sum_{r=1}^{|T|} {}_{|r-EL|}C_{k-1} \qquad \text{eq. 2}$$

In eq.1 and 2, the size of each EL (r-EL) is unknown. If we consider T as a uniform tree *T(n,r),* a relationship between height *n* and degree *r* of a uniform tree T with the size of each EL for each node can be derived.

Determining $_r\delta^{n-d}$ of uniform tree T(n, r). $_r\delta^{n-d}$ is denoted as the size of the embedded list, |EL| of a node in *T(n,r)* with depth *d*. A close look at EL definition in section 2 and |T(n,r)|, suggests that $_r\delta^{n-d}$ is described by a geometrical series formula $r(r^{(n-d)}-1)/(r-1)$. In a uniform tree *T(n, r)*, there are r^d number of nodes at each level *d*. Thus, for each level in T(n,r) there are r^d number of lists that have the same size $_r\delta^{n-d}$.

$$r^d \,{}_r\delta^{n-d} \qquad \text{eq. 3}$$

Using the fact that for each level in *T(n,r)* there are r^d number of lists that have the same size $_r\delta^{n-d}$, eq 2 can be expressed as shown below.

$$r^0 \,{}_{r}\delta^{n} C_{k-1} + r^1 \,{}_{r}\delta^{n-1} C_{k-1} + \ldots + r^n \,{}_{r}\delta^{0} C_{k-1} \qquad \text{eq. 4}$$

Further, eq. 4 can be written as follows:

$$\sum_{i=0}^{n-1} r^i \,{}_{r}\delta^{n-i} C_{k-1}, \text{ for } {}_r\delta^{n-i} \geq (k-1) \qquad \text{eq. 5}$$

Please note that whenever the |EL| < (k-1) no candidate subtrees would be generated, thus the constraint $_r\delta^{n-I} \geq (k-1)$ takes care of this condition. Hence, using the developed equations, calculating the complexity of total k-subtree candidates from a uniform tree *T(n,r)* for k=1,…,|*T(n,r)*| is given by the following equations:

$$\sum_{k=1}^{|T(n,r)|} \|T(n,r)\|_k = \|T(n,r)\|_1 + \sum_{k=2}^{|T(n,r)|} \|T(n,r)\|_k \qquad \text{eq. 6}$$

Thus, given an arbitrary tree T and its closed form T'(n,r), the worst case complexity of enumerating embedded subtrees using TMG approach from T can be computed using eq. 6 where n is the height of T' and r is the degree of T'. Due to space limitation we would reserve a more details discussion about complexity issue in our future work.

5. Results and Discussions

This section provides some comparisons between the MB3-Miner (MB3), X3-Miner (X3), VTreeMiner (VTM) and PatternMatcher (PM) algorithms. We have synthetic database of trees with varying: size (s), max. height (h_{max}), max. fan-out (f_{max}), and number of transactions ($|T_r|$) We use a short hand notation XXX–T, XXX-C, and XXX–F to denote total execution time (including the data preprocessing, variables declaration, etc); number of subtree candidates generated, and the number of frequent subtrees generated with XXX approach respectively. The minimum support σ is denoted as (sxx), where xx is the minimum frequency. Experiments were run on a machine using 3Ghz (Intel-CPU), 2Gb RAM, Mandrake 10.2 Linux where each algorithm was run exclusively. The source code for each algorithm is compiled using GNU g++ version 3.4.3 with –g and –O3 parameters. We run TreeMiner with –u parameter (weighted support).

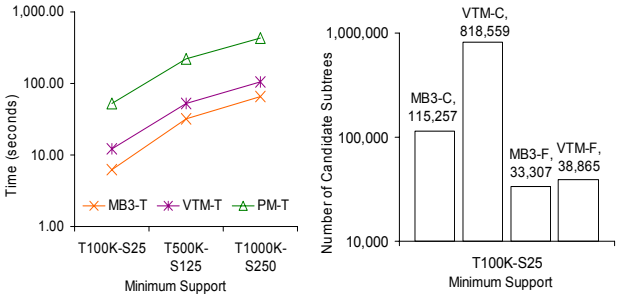


Figure 5. Scalability test: (a) time performance (b) number of subtrees generated

Scalability (s:10,h_{max}:3,f_{max}:3). In this experiment, $|T_r|$ were varied to 100K, 500K and 1000K, with σ of 25, 125 and 250, respectively. From the figure 5a we can see that all three algorithms are well scalable, and that MB3

outperforms others with respect to time. Figure 5b compares MB3 with VTM in the number of candidates generated versus the determined number of frequent candidates. Using the join approach, it can be seen that VTM generates more candidates (VTM-C). These candidates are in fact invalid candidates, in the sense that they do not conform to the tree model.

Deep (s:28,h_{max}:17,f_{max}:3) vs wide (s:428,h_{max}:3, f_{max}:50) trees. This experiment was conducted to verify the worse performance of a DFS approach VTM on deep trees, and BSF approach (MB3 & PM) on wide trees. The deep tree data has $|T_r|$:10,000 with a total of 273,090 nodes. In figure 6a we can see the performance of VTM degrades significantly after support is lowered. VTM struggles to finish within a reasonable time and jumps significantly to 7,177.85 seconds at $\sigma \leq 150$. Overall, the PM algorithm performs better than the VTM while the MB3 algorithm enjoys the best performance and stability even at very low support.

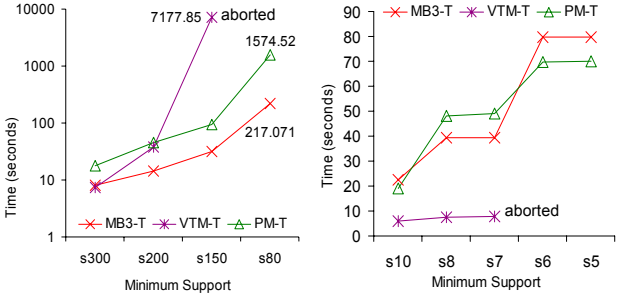


Figure 6. (a) Deep tree (b) Wide tree

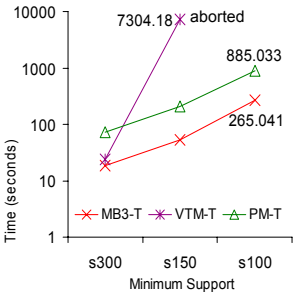


Figure 7. Mix dataset

For the wide tree data, $|T_r|$ = 6,000 with a total of 1,303,424 nodes. As expected the DFS based approach like VTM outperforms MB3 on this dataset. However when the support threshold was decreased below 7, VTM failed to finish the task. As the DFS based approach and BFS based approach suffer from, deep and wide trees respectively, we tested the performance on a mixed dataset *(s:428,h_{max}:17,f_{max}:50,$|T_r|$=76,000)*. MB3 performs the best in this case as is shown in figure 7.

Uniform trees (s:20,h_{max}:3,f_{max}:4). As most real world tree structured datasets are usually not in the form of a complete tree we have created an artificial dataset that represents a uniform tree. In this dataset, $|T_r|$:20,000 with a total of 246,110 nodes.

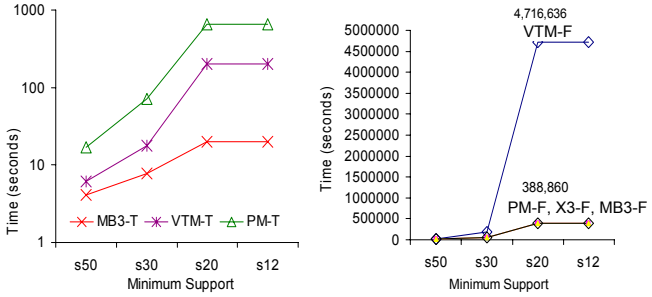


Figure 8. Uniform Trees: (a) time performance graph (b) number of frequent subtrees graph

Figure 8a shows that MB3 has the best time performance. As illustrated in figure 8b, the number of frequent candidates generated by VTM (VTM-F) is substantially larger (~12x) than MB3-F, X3-F, PM-F as the support decreases. Performing full (k-1) pruning is a challenge in a DFS based approach [15]. A DFS based approach such as VTM has to rely on the opportunistic pruning. This results in many pseudo-frequent candidate subtrees that should be pruned.

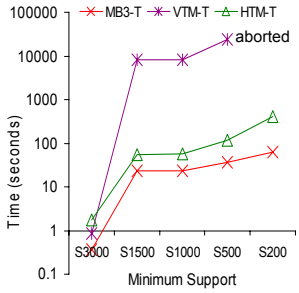


Figure 9. 54% transactions of original CSLogs data

CSLogs (s:214,h_{max}:28,f_{max}:21). This real-world data set was previously used by Zaki in [15] to test the VTM using transactional support definition. When we tried to use it for occurrence match support, all the tested algorithms had problems in returning frequent subtrees. We start to see interesting result when we cut $|T_r|$ from 56,291 to 32,241 randomly. With this partial data set there were problems with VTM returning the result with $\sigma \leq 500$. From figure 9 it can be seen that MB3 again has the best performance.

Enumeration complexity. We created 4 datasets of uniform tree T(2,2), T(3,2), T(2,3), T(2,4) where all nodes have distinct labels. We specify σ:1. Figure 10 shows that the enumeration cost using the join approach (VTM) is higher than the TMG approach (MB3, X3M). For data with higher complexity such as T(3,3) and T(4,3) all algorithms used get aborted. We verify with eq 6 that the enumeration cost turns out to be very large 549,755,826,275 and 1.33×10^{36} respectively. Furthermore, the cost of enumeration as shown in figure 10 can be verified using eq 6. A program to calculate enumeration complexity of

complete tree *T(n,r)* can be requested from the authors. Thus, the TMG approach is a predictable enumeration model where its enumeration cost can be measured and verified mathematically, allowing one to isolate difficult situations.

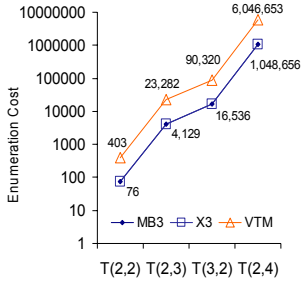


Figure 10. VTM, MB3, & X3 enumeration complexity

Overall Discussion. MB3 demonstrates high performance and scalability. In general, the performance increase comes from the efficient use of the EL representation and the optimal TMG candidate generation approach that ensures only valid candidate subtrees are enumerated. In figure 5b it is shown that the number of invalid subtrees generated by the join approach can be enormous. This can degrade the performance. Furthermore, MB3 performs expensive full (k-1) pruning and produces no pseudo-frequent candidate subtrees. VTM utilizes less expensive opportunistic pruning which as a trade-off generates many pseudo-frequent candidate subtrees. This can be seen from figure 8b, at σ:20 VTM generates ~12x more than MB3,PM, & X3. One of the consequences is that this greatly degrades VTM performance. In figure 6,7, & 9 VTM even failed to provide results within a reasonable time at a very low minimum support σ. In the context of association mining, regardless of which approach is used, for a given dataset with minimum support σ specified, the discovered frequent patterns should be identical and consistent. Considering that pseudo-frequent subtrees are infrequent subtrees, techniques that don't perform full pruning would generate pseudo-frequent subtrees and therefore would have limited applicability to association rule mining.

6. Conclusions

In this study we have provided some detailed discussions about various theoretical and performance issues of the different approaches. We proposed a novel and unique embedding list representation and showed the strength of the TMG enumeration approach which was formulized mathematically. High performance and scalability of the MB3 algorithm was demonstrated in our experiments by contrasting it with the state of the art algorithm TreeMiner.

Acknowledgement

A special thanks to Prof. M. J. Zaki [15] for providing us the TreeMiner source code and discussing the results obtained from it with us.

References

[1]. R. Agrawal, R. Srikant, "Fast Algo. for Mining Assoiciation Rules," *In Proc. the 20th VLDB,* 487–499, 1994.

[2]. K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa, "Optimized Substructure Discovery for Semistructured Data," *In Proc. PKDD'02*, 1–14, LNAI 2431, *2002.*

[3]. Y. Chi, S. Nijssen, R.R. Muntz, J. N. Kok, "Frequent Subtree Mining An Overview," *Fundamenta Informaticae*, Special Issue on Graph and Tree Mining, 2005.

[4]. L. Feng, T. S. Dillon, H. Weigand, E. Chang, "An XML-Enabled Association Rule Framework," *In Proc. of DEXA'03,* pp 88-97, 2003.

[5]. J. Ferrans, B. Lucas, K. Rehor, B. Porter, A. Hunt, S. McGlashan, et. al., "Voice Extensible Markup Language (VoiceXML) Version 2.0," W3C Technical Report, March, 2004

[6]. V. Geroimenko, C. Chen, *Visualizing Information Using SVG and X3D*, Springer, 2004

[7]. M. Kuramochi, and G. Karypis, "An Efficient Algorithm for Discovering Freq. Subgraphs," *IEEE Transactions Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1038-1051, 2004.

[8]. H. Tan, T.S. Dillon, L. Feng, E. Chang, F. Hadzic, "X3-Miner: Mining Patterns from XML Database*," In Proc. Data Mining '05. Skiathos, Greece*, 2005.

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

[10]. Wang, M. Hong, J. Pei, H. Zhou, W. Wang and B. Shi, "Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining," in *Proc. of PAKDD'04*, 2004.

[11]. K. Wang and H. Liu, "Discovering Typical Structures of Documents: A Road Map Approach," *In Proc. ACM SIGIR Conf. Information Retrieval,* 1998.

[12]. L. H. Yang, M. L. Lee, & W. Hsu, "Efficient Mining of XML Query Patterns for Caching," *In Proc. the 29th VLDB Conf.,* 2003.

[13]. Zhang, J., Ling, T. W., Bruckner, R. M., Tjoa, A. M., Liu, H., "On Efficient and Effective Association Rule Mining from XML Data," *In Proc. of DEXA'04,* pp. 497 - 507, 2004.

[14]. M.J. Zaki, "Fast Vertical Mining Using Diffsets," *In. Proc. of SIGKDD'03,* 2003.

[15]. 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.