Tree Model Guided Candidate Generation for Mining Frequent Subtrees from XML Documents

Henry Tan, Fedja Hadzic, Tharam S. Dillon, Elizabeth Chang

Curtin University of Technology, Digital Ecosystems and Business Intelligence Institute, Perth, Australia

Ling Feng

Tsinghua University, Beijing, China

Due to the inherent flexibilities in both structure and semantics, XML association rules mining faces few challenges, such as: a more complicated hierarchical data structure and ordered data context. Mining frequent patterns from XML documents can be recast as mining frequent tree structures from a database of XML documents. In this study, we model a database of XML documents as a database of rooted labeled ordered subtrees. In particular, we are mainly concerned with mining frequent induced and embedded ordered subtrees. Our main contributions are as follows. We describe our unique *embedding list* representation of the tree structure, which enables efficient implementation of our *Tree Model Guided* (*TMG*) candidate generation. *TMG* is an optimal, non-redundant enumeration strategy which enumerates all the valid candidates that conform to the structural aspects of the data. We show through a mathematical model and experiments that *TMG* has better complexity compared to the commonly used join approach. In this paper, we propose two algorithms, MB3-Miner and iMB3-Miner. MB3-Miner mines embedded subtrees. iMB3-Miner mines induced and/or embedded subtrees by using the *maximum level of embedding constraint*. Our experiments with both synthetic and real datasets against two well known algorithms for mining induced and embedded subtrees, demonstrate the effectiveness and the efficiency of the proposed techniques.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Information filtering*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *Performance evaluation* (*efficiency and effectiveness*); I.7.0 [**Document and Text Processing**]: General

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Tree Mining, TreeMiner, FREQT, TMG, Tree Model Guided

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. Association mining has been very successful in discovering useful associations between data, particularly for relational data. Due to the inherent flexibilities in both structure and semantics, XML association mining faces several challenges, such as: 1) more complicated hierarchical data structure; 2) ordered data context; and 3) much bigger data size. The bigger data size arises from two sources: a) an XML record is more annotated through the tags than a relational record, and b) the actual amount of semi-structured (or unstructured) data (documents) greatly exceeds the amount of relational data [Luk et al. 2002]. Most of the research done in association mining was tailored for structured data with only a few cases addressing semi-structured data. While some approaches have focused on mining for patterns in databases containing general graphs [Ruckert and Kramer 2004; Yan and Han 2002], the increase in the amount of XML data and the need for mining semi-structured data has sparked a lot of interest in finding frequent trees from a database of rooted ordered labeled trees. This problem is known as frequent subtree mining and can be generally stated as: given a tree database T_{db} and minimum support threshold (σ), find all subtrees that occur at least σ times in T_{db}.

The two known types of subtrees are induced and embedded [Abe et al. 2002; Chi et al. 2005, Tan et al. 2005a, 2005b, 2006a; Zaki 2005]. An induced subtree is a subtree where the parent-child relationships must be the same to those in the original tree. In addition to this, an embedded subtree allows a parent in the subtree to be an ancestor in the original tree and hence the information about ancestor-descendant relationships is kept. Furthermore, if the subtree is ordered then the left-to-right ordering among sibling nodes in the database tree is preserved. Examples of ordered induced and embedded subtrees are given in Figure 1 and formal definitions are provided in Section 3.

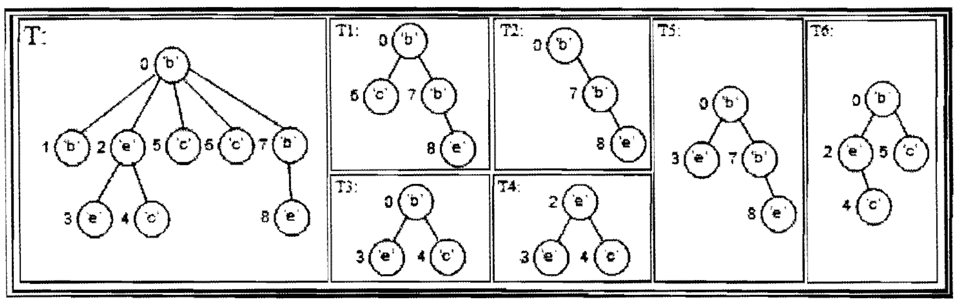


Figure 1: example of induced subtrees (*T1, T2, T4, T6*) and embedded subtrees (*T3, T5*) of tree *T*
(note that induced subtrees are also embedded subtrees)

While more interesting patterns can be obtained when mining embedded subtrees, unfortunately mining such embedding relationships can be very costly. Induced subtrees are subset of embedded subtrees and the complexity of mining embedded subtrees is higher than mining induced subtrees [Chi et al. 2005; Tan et al. 2005b; Zaki 2005].

Another issue is how subtree occurrences are counted in the database. Currently the most commonly used support definitions are *occurrence-match* and *transaction-based* support [Zaki 2005; Tan et al. 2005b]. The term transaction was originally introduced in the data management field where it refers an atomic interaction with a database management system. However, in the data mining field the term transaction has adopted a different meaning. To clarify its use in the context of tree mining, we find the following definition suitable. A transaction is a set of one or more items obtained from a finite item domain, and a dataset is a collection of transactions [Bayardo et al. 1999]. Hence, in the context of a tree database, a transaction would correspond to a fragment of the database tree whereby an independent instance is described. *Transaction-based* support is used when only the existence of items within a transaction is considered important, whereas *occurrence-match* support takes the repetition of items in a transaction into account and counts the subtree occurrences in the database as a whole. Formal definitions will be provided in Section 3.

In this study, we are mainly concerned with mining frequent ordered induced/embedded subtrees from a database of rooted ordered labeled subtrees. Our primary objectives are as follows: (1) to present an efficient and scalable technique, (2) to provide a method to control and limit the inherent complexity present in mining frequent embedded subtrees, (3) to evaluate the use of *occurrence-match* support as well as *transaction-based* support. To achieve the first objective, we utilize a novel tree representation called *embedding list* (*EL*), and employ an optimal enumeration strategy called *Tree Model Guided* (*TMG*). The second objective can be attained by restricting the *maximum level of embedding* that can occur in each embedded subtree. The *level of embedding* is defined as the length of the path between two nodes that form an ancestor-descendant relationship. Intuitively, when the *level of embedding* inherent in the database of trees is high, numerous numbers of embedded subtrees exist. Thus, when it is too costly to mine all frequent embedded subtrees, one can restrict the *maximum level of embedding* gradually up to 1, from which all the obtained frequent subtrees are induced subtrees. Finally, we analyze the issue of using *occurrence-match* support for mining frequent subtrees and highlight the importance of full (*k-1*) pruning [Zaki 2005; Tan et al. 2005b] when this support definition is considered.

In contrast to our previous works [Tan et al. 2005b, 2006a], this paper provides supplementary theoretical and experimental discussion of some important aspects of induced and embedded subtree mining. Introducing the *level of embedding* concept has established a clearer picture of the relationship between the tasks of mining induced and embedded subtrees. A discussion of implementation issues that commonly arise when developing algorithms for frequent subtree mining is provided and experimentally confirmed. Some of the commonly used hash functions were experimentally evaluated in regards to their efficiency for storing and counting subtree occurrences. We have also extended our technique so that the *transaction-based* support definition can be used. Comparisons of Algorithms were done on a more diverse set of experiments, choosing tree databases with different characteristics, and different settings of support thresholds. This indicated certain database characteristics where applying one particular technique would be more advantageous over others, and in more general terms it provided further insight into some strengths as well as limitations of the compared techniques for mining induced/embedded subtrees. The key contribution is the development of a comprehensive theoretical and implementation framework, which addresses induced and embedded subtrees, clarifies the relationship between those and utilizes both transaction-based and occurrence-match support. This is experimentally shown to be more efficient than existing algorithms. A powerful tree model guided approach heavily reduces candidate generation. The novel data structure, embedding list, is used to speed up the handling of enumeration at the implementation level. In addition, the nature of data and its influence on the efficacy of different algorithms is investigated.

The paper is organized as follows. Section 2 presents related works. The emergence of semi-structured documents and the challenges of mining such data are discussed together with some of the important issues that arise when developing tree mining algorithms. In Section 3, the problem definitions are given and the important aspects of the tree mining area necessary for understanding the work presented in this paper are discussed. Section 4 describes the details of the algorithm. The mathematical model of the *TMG* candidate enumeration technique is provided in Section 5. iMB3-Miner as extension to MB3-Miner is presented in Section 6. In Section 7 we empirically evaluate the performance of our algorithms by comparing it with some state-of-the-art algorithms for mining induced and embedded subtrees. The paper is concluded in Section 8.

2. Previous works

Unlike traditional well-structured data whose schema is known in advance, XML data may not have a fixed schema, and the structure of data may be incomplete or irregular. This is why XML data is referred to as semi-structured data [Suciu 2000]. Several works have been proposed for mining XML documents [Abe et al. 2002; Chi et al. 2005; Feng et al. 2003; Feng and Dillon 2004; Wang and Liu 1998; Yang et al. 2003, Zhang et al. 2004; Zhang et al. 2005]. If the focus is purely on values associated with the tags, this is by and large no different from traditional association rule mining. One interesting work is to discover similar structures among a collection of semi-structured objects [Abe et al. 2002; Feng et al. 2003; Feng and Dillon 2004; Tan et al. 2005a].

Association mining consists of two important problems, i.e. frequent patterns discovery and rule construction [Agrawal et al. 1993, Agrawal and Srikant 1994; Agrawal et al. 1996]. The former task is considered to be a more difficult problem to solve than the latter and has become the focus of many studies [Abe et al. 2002; Chi et al. 2005; Nijssen and Kok 2003; Ruckert and Kramer 2004; Tan et al. 2005a, 2005b, 2006a; Wang et al. 2004; Zaki 2005].

An XML-enabled framework for representation of association rules in databases was first presented in Feng and Dillon [2003]. It extends the notion of associated items to XML fragments to present associations among trees. Despite the strong foundation established in Feng and Dillon [2003], an efficient way to implement the framework had not been discussed. Recently, a hybrid approach XAR-Miner was proposed in Zhang et al. [2004] and Zhang et al. [2005] for efficient data selection and association rule mining. Depending on the size of the XML documents, the data is either transformed into Indexed XML Tree (IX-Tree) or Multi-relational databases (Multi-DB) through which the hierarchical information is maintained and data is indexed. The desire to focus on certain interesting rules leads Feng and Dillon [2004] to use a template approach to focus the search on the interestingness of the rule. An extension of this approach to define language constructs that allow one to carry out rule mining for a language such as XQuery is put forward in Feng and Dillon [2005]. In general, XQuery-based approaches [Feng and Dillon 2004, 2005; Wan and Dobbie 2003] suffer from a poor performance if they are used to mine association rules by exhausting a large search space. It has been suggested in Zaki [2005] and Tan et al. [2005a] that one of the main issues in XML association rule mining is mining frequent subtrees in a database of XML documents.

There are different types of trees. One can distinguish between unrooted unordered trees (free trees) [Chi et al. 2004; Ruckert and Kramer 2004], rooted unordered trees [Nijssen and Kok 2003], and rooted ordered trees [Abe et al. 2002; Tan et al. 2005a, 2005b, 2006a]. The three types of trees have increasing topological structure as one progresses from the first to the third [Chi et al. 2005]. A rooted tree is a tree with a special node called the root node which does not have a parent. Tan et al. [2005a] suggested that mining frequent patterns from XML documents can be recast as mining frequent tree structures from the database of rooted labeled ordered subtrees.

A related but not identical problem is to consider the issue of mining sub-sequences from a database of sequences. A sequence contains no hierarchical relationship but only horizontal (linear) relationships. Each item in a sequence has fan-out 1. By definitions the order of items in a sequence is important. A tree structure on the other hand has hierarchical relationships and horizontal relationships. A uniform tree with degree 1 has only hierarchical relationships. By definition, hierarchical relationships imply that the order between nodes is vertically significant. By corollary, we can then view a sequence as a uniform tree with nodes degree 1. With this definition, we can define a sequence of itemsets as a collection of uniform trees with nodes degree 1 rooted on the same root node, and we refer to this tree as a *vertical tree* [Tan et al. 2006b].

A tree structure can be represented using the *adjacency matrix* and the *adjacency list* representation. In the data mining community, a string-like representation is becoming very popular [Abe et al. 2002; Chi et al. 2005; Wang et al. 2004; Yang et al. 2003; Zaki 2005]. Each item in the string can be accessed in *O(1)* time and the representation itself has been reported to be space efficient and provides ease of manipulation [Chi et al. [2005]; Tan et al. 2005a, 2005b, 2006a]. When using depth-first string-like representation, a notion of scope is used to denote the position of its descendant's node position. Thus the hierarchical structure embedded in tree data is semantically preserved and the original tree structure can be reconstructed from the string-like representation.

There are various algorithms that mine different types of tree patterns. FreeTreeMiner for graphs [Ruckert and Kramer 2004] extracts free trees in a graph database. PathJoin [Xiao et al. 2003], uFreqt [Nijssen and Kok 2003], and HybridTreeMiner [Chi et al. 2004], mine induced, unordered trees. Zaki presented TreeMiner [Zaki 2005], an algorithm to discover all frequent embedded subtrees in a forest using a data structure called the vertical scope-list and utilizing the join approach for candidate generation. TreeMiner consists of two versions, one, which adopts a depth-first search (VTreeMiner) and one which uses the breadth-first search (PatternMatcher) for frequent subtrees. Generally speaking the depth-first approach is more efficient for processing long-patterns and is also more space efficient than the breadth-first approach. The breadth-first approach requires more memory, since when enumerating all *k-subtrees* (i.e. subtrees consisting of *k* nodes) the occurrence of all *(k-1)-subtrees* is kept in memory to perform the extension. On the other hand, the depth-first approach does not need to keep all subtree occurrences in memory since all possible extensions of a particular subtree have already been enumerated. However one significant drawback of depth-first approach is that it cannot ensure that all *(k-1)-subtrees* of a *k-subtree* are frequent (i.e.

perform full *k-1* pruning). It becomes a challenge because information of (*k-1*) subtrees is not guaranteed to be readily available. VTreeMiner overcomes this issue by implementing opportunistic pruning [Zaki 2005] which is a work around over performing full pruning (i.e. enumerating (*k+1*)-subtree from *k*-subtrees). TreeMiner is one of the most efficient current approaches to tree mining and the algorithm could be extended for the purpose of mining frequent tree structures in XML documents.

The two known enumeration strategies are enumeration by extension and join [Chi et al. 2005]. Recently, Zaki [2005] adapted the join enumeration strategy for mining frequent embedded rooted ordered subtrees. Another kind of enumeration technique is to utilize structural information from the data. The utilization of schema or structural information was essential for many tasks. An idea of utilizing schema information for mining frequent patterns from XML documents appeared in Yang et al. [2003]. The approach uses the XML schema to guide the candidate generation so that all candidates generated are valid because they conform to the schema. Another study about utilization of schema information was reported in Papakonstantinou and Vianu [2000]. They developed a technique to generate views of XML data from its schema. The technique utilizes the schematic information of the data to enable an automatic inference of Data Type Definitions (DTDs) for views of XML data. If it is done manually, this is not only difficult to do but also error-prone.

We have developed a candidate enumeration method for mining embedded rooted ordered labeled subtree, which we refer to as *Tree Model Guided* (*TMG*) [Tan et al. 2005a, 2005b, 2008]. The *TMG* can be applied to any data that has a model representation with clearly defined semantics (schema) that have tree-like structures. However, the *TMG* does not need an explicit schema definition to perform the candidate generation as it can infer the tree structural information just from the document itself. Hence, this enables the *TMG* to generate only valid candidate subtrees. A candidate subtree can be considered valid in two ways. Firstly, by conforming to an available model representation of the document tree structure, and secondly by conforming to the tree structure through which the information presents in the examined document is represented. By defining a sequence as a *vertical tree*, Tan et al. [2006b] has demonstrated that the *TMG* approach can also be generalized and applied to a database of sequences.

The enumeration strategy used by *TMG* is a specialization of the *right-most-path* extension approach [Abe et al. 2002; Zaki 2005]. However, it is different from the one that is proposed in FREQT [Abe et al. 2002] as *TMG* enumerates embedded subtrees and

FREQT enumerates only induced subtrees. The *right-most-path* extension method is reported to be complete and all valid candidates are enumerated at most once (non-redundant) [Abe et al. 2002; Tan et al. 2005a, 2005b]. This is in contrast to the incomplete method TreeFinder [Termier et al. 2002] that uses an Inductive Logic Programming approach to mine unordered, embedded subtrees. TreeFinder can miss many frequent subtrees. The extension approach utilized in the *TMG* generates fewer candidates as opposed to the join approach [Zaki 2005]. Independently, XSpanner [Wang et al. 2004] extends the Pattern-Growth concept into tree structured data and its enumeration model also generates only valid candidates. XSpanner only reports distinct embedded subtrees similar to the recently published TreeMinerD [Zaki 2005]. TreeMinerD is different from TreeMiner in the sense that TreeMiner reports all embedding subtrees. Despite the fact that the experimental study performed by the XSpanner authors suggested that XSpanner outperforms TreeMiner, a very recent study by Tatikonda et al. [2006] suggested the opposite. They reported that XSpanner performs much worse than that of TreeMiner for the many datasets they used. XSpanner suffers from poor cache performance due to expensive pseudo-projection step. They suggested that in general the problems with the FP-growth based approaches are very large memory footprint, memory trashing issue, and costly I/O processing [Ghoting et al. 2005].

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 rapid counting approach is extremely important. Efficiency of candidate counting is heavily 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 [Agrawal and Srikant 1994; Chi et al. 2005] data structure can be used to accelerate direct checking. Another approach projects each candidate generated into a vertical representation [Chi et al. 2005; Zaki 2003, 2005], which associates an occurrence list with each candidate subtree. If *transaction-based support* [Chi et al. 2005] is used, the vertical format will consist of transaction IDs of the transactions that support it. In contrast, if *occurrence-match* [Tan et al. 2005b] or *weighted-support* definition [Zaki 2005] is used, each list will correspond to each candidate occurrence in the whole database of trees. *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 [Chi et al. 2005; Tan et al. 2005b; Zaki 2003, 2005].

In Tan et al. [2005a], a vertical list format is utilized for performing efficient frequency counting. In this paper, we modify this vertical list format in two ways. First, the performance is expedited by storing only the hyperlinks [Wang et al. 2004] of subtrees in the tree database instead of creating a local copy for each generated subtree. The format is different than the scope-list [Zaki 2005] representation as our vertical list does not store any scope information. Secondly, we transform and map the string-labeled trees data into integer-labeled trees as opposed to processing time consuming string labels directly. Representing labels as integers instead of string labels has performance and space advantages [Tan et al. 2005b]. 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.

3. 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) $E = \{(x,y) | x,y \in V\}$ is the set of edges in the tree. A *root* is the topmost node in the tree. In a labeled tree, there is a labeling function mapping vertices to a set of labels so that a label can be shared among many vertices. The parent of node v is defined as its predecessor, denoted as *parent*(v). The predecessor of *parent*(v) is defined as its ancestor, denoted as *ancestor*(v). The ancestor of *ancestor*(v) is also defined as *ancestor*(v). Each node in the tree can have only one parent, but it can have one or more children, which are defined as its successors.The *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 *right-most-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 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 *height of a node* is the length of the path from that node to its furthest leaf. The *right-most-path* of T is defined as the path connecting the *right-most-leaf* with the root node. The *height of a tree* is defined as the height of its root node. The *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. A *uniform tree* $T(n,r)$ is a tree with height equal to n and all of its internal nodes have degree r. The *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. In this paper, the term 'k-subtree' refers to a subtree that consists of k number of nodes.

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) 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 does not create a forest in T.

Definition 2: A tree $T'(r', V', L', E')$ is an *embedded subtree* of a tree $T(r, V, L, E)$ if, and only if, (1) $V' \subseteq V$, (2) $L' \subseteq L$ and $L'(v)=L(v)$, (3) $\forall v' \in V'$, $\forall v \in V$, v' is not the root node, and v' has a parent in T', then *parent*(v')=*ancestor*(v) and the sets *ancestor*(v') $\cap$ *ancestor*(v) $\neq \phi$ (form a non-empty intersection). Examples of induced and embedded subtrees are given in Figure 1, where for each node, the label is shown inside the circle whereas its pre-order position is shown as an index at the left side of the circle.

Definition 3: 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* $\Delta(p,q)$ is defined as the length of the path between p and q, where $p \in V'$ and $q \in V'$, and p and q form an ancestor-descendant relationship. A *maximum level of embedding* (δ) is the limit on the level of embedding between any p and q. In other words, given a tree database T_{db} and δ, then any embedded subtree to be generated will have the maximum length of a path between any two ancestor-descendant nodes equal to 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.

Definition 4: The notation $t \prec k$, is used to denote an embedded subtree t which is supported by transaction $k \subseteq K$ in database of tree T_{db} 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 5: 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 T_{db}, the support of an embedded subtree t in T_{db} is defined as:

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

Transaction-based support has been used in [Chi et al. 2005; Wang et al. 2004; Zaki 2005]. However *occurrence-match* support has been less utilized and discussed. In this study we are in particular interested in exploring the application and the challenge of using *occurrence-match* support. 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. There has not been any general consensus which support definition is used for which application. However, it is intuitive to say that whenever order is important and repetition of items in each transaction is to be accounted for, *occurrence-match* support would be more applicable, i.e. when we are considering items as structured entities. Generally, *transaction-based* support is very applicable for relational data, since order and structure is generally not important in this case. To illustrate the importance of occurrence match support, consider the partial XML representation of protein data displayed in Figure 2. The original dataset describes a protein ontology instance store for Human Prion Proteins in XML format [Sidhu and Dillon 2005]. Protein Ontology (PO) provides a unified vocabulary for capturing declarative knowledge about the protein domain and classifies that knowledge to allow reasoning. Information captured by PO is classified in a rich hierarchy of concepts and their inter-relationships. Using the PO format, ATOMSequence labels can be compared easily across PO datasets for distinct protein families to determine sequence and structural similarity among them. Structured ATOMSequence labels, with repetition of Chain, Residue and Atom details can be used to compare a new unknown protein sequence and structure with existing proteins in the PO dataset, which helps users in drug discovery and design. In this case the repetition in the structure of the protein is of considerable importance.

Another scenario where *occurrence-match* support may be important is when performing specialized queries on a tree structured database. As an example, consider a library based application where author information may be separately stored in each transaction. A user may be interested in finding out information about the authors that have published at least X books with publisher Y.

```
. . .
<ATOMSequence>
    <ProteinOntologyID>PO0000000026</ProteinOntologyID>
    <_ATOM_Chain>C</_ATOM_Chain>
    <_ATOM_Residue>ALA</_ATOM_Residue>
    <AtomID>4011</AtomID>
    <Atom>N</Atom>
    <ATOMResSeqNum>196</ATOMResSeqNum>
    <X>14.052</X>
    <Y>77.339</Y>
    <Z>-2.999</Z>
    <Occupancy>4011</Occupancy>
    <TempratureFactor>4011</TempratureFactor>
    <Element>N</Element>
  </ATOMSequence>
  <ATOMSequence>
    <ProteinOntologyID>PO0000000026</ProteinOntologyID>
    <_ATOM_Chain>C</_ATOM_Chain>
    <_ATOM_Residue>ALA</_ATOM_Residue>
    <AtomID>4012</AtomID>
    <Atom>CA</Atom>
    <ATOMResSeqNum>196</ATOMResSeqNum>
    <X>13.5</X>
    <Y>76.085</Y>
    <Z>-3.481</Z>
    <Occupancy>4012</Occupancy>
    <TempratureFactor>4012</TempratureFactor>
    <Element>CA</Element>
  </ATOMSequence>
  . . .
```

Figure 2: snapshot of the representation of Human Prion Protein dataset in XML format

To satisfy this query, the repetition of author-book-publisher relation within a transaction will need to be considered. In these scenarios the repetition of items within a transaction is considered important and the knowledge of the number of repetitions provides useful information. Hence, for these purposes *occurrence-match* support would be more suitable than the *transaction-based* support.

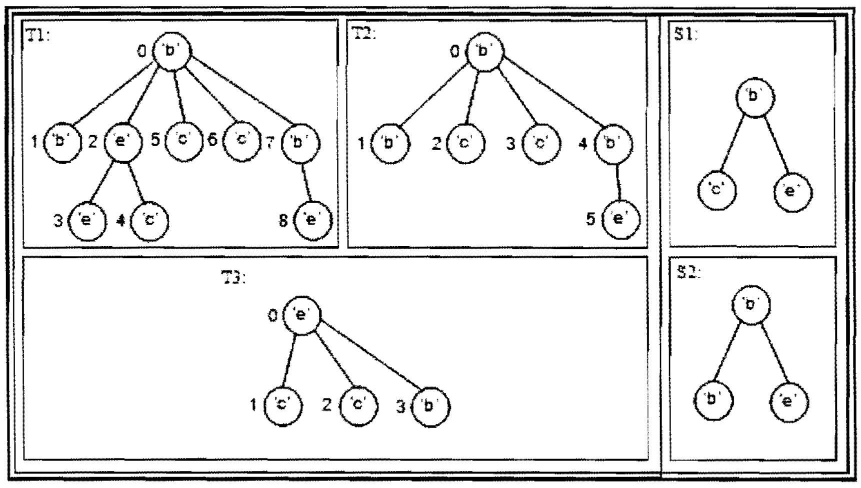


Figure 3: tree T1, T2, and T3 with subtree S1 and S2 to illustrate *transaction-based* and *occurrence-match* support definitions

The following example illustrates the effect of applying different support definitions described above. In Figure 3 there are three transactions, T1, T2, and T3. Suppose that *transaction-based* support is considered, the support of subtree S1 is equal to 2 as S1 is supported by T1 and T2 but not T3, i.e. S1 $\prec$ T1 and S1 $\prec$ T2. The support of subtree S2 is also equal to 2 as S2 is supported by T1 and T2 but not T3, i.e. S2 $\prec$ T1 and S2 $\prec$ T2. If *occurrence-match* support is considered, the support of subtree S1 is equal to the sum of its occurrences in T1, T2, and T3, i.e. *g*(S1,T1)+*g*(S1,T2)+*g*(S1,T3). It can be seen from Figure 3 that there are three occurrences of S1 in T1, two occurrences of S1 in T2 and none in T3, and hence the *occurrence-match* support of subtree S1 equals to 5. Counting the *occurrence-match* support of subtree S2 in the same way, gives us the *occurrence-match* support of 4 as there are three occurrences of S2 in T1 and one occurrence of S2 in T2.

String encoding (ϕ). We utilize the pre-ordering string encoding (ϕ) as described in [Tan et al. 2005b; Zaki 2005]. We denote the encoding of a subtree T as $\phi(T)$ and as an example from Figure 1, $\phi(T1)$:'*b c / b e / /*' and $\phi(T3)$:'*b e / c /*', respectively. 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 backtrack symbols after the last node, i.e. $\phi(T1)$:'*b c / b e*'. We refer to a group of subtrees with the same encoding *L* as *candidate subtrees* C_L. 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, as in the encoding used above the backtrack symbols need to be computed accordingly. For example, when appending a subtree '*b e*' *{0,3}* with '*c*' {4}, denoted as '*b e*' + '*c*', the resulting subtree is '*b e / c*' *{0,3,4}*. The operator '+' in this case appends '*b e*' with 'c' by inserting one backtrack symbol '/'. As mentioned earlier, the number of backtracks are determines by the number of times we have to move up a node in the tree during the pre-order traversal of the tree being represented by the encoding.

Mining (induced|embedded) frequent subtrees. Let T_{db} be a tree database consisting of N transactions of trees, K_N. The task of frequent (induced|embedded) subtree mining from T_{db} with given minimum support (σ), is to find all the candidate (induced|embedded) subtrees that occur at least σ times in T_{db}. Based on the downward-closure lemma [Agrawal and Srikant 1994], 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 as to whether the same principle applies to tree structured data when the *occurrence-match* support definition is used. To show that the same principle does not apply, we need to find a counter-example.

Definition 6. Given a tree database T_{db}, if there exist candidate subtrees C_L and $C_{L'}$, where C_L is a subset of $C_{L'}$ ($C_L \subseteq C_{L'}$), such that $C_{L'}$ is frequent and C_L is infrequent, we say that $C_{L'}$ is a *pseudo-frequent* candidate subtree. In the light of the downward closure lemma these candidate subtrees are infrequent because one or more of its subtrees are infrequent.

Lemma 1. The *anti-monotone* property of frequent patterns suggests that the frequency of a superpattern is less than or equal to the frequency of a subpattern. If *pseudo-frequent* candidate subtrees exist then the *anti-monotone* property does not hold for frequent subtree mining.

In the following example we will illustrate a pseudo-frequent subtree by drawing up instances as a case in point. First we will show an example of a frequent subtree and then an example of a pseudo-frequent subtree. We will use Figure 3 to draw examples. Suppose that the minimum support σ is set to 2. A candidate subtree C_L where *L*:'*b c / b*', is an example of a frequent subtree since there are three occurrences of embedded subtrees C_L that occur at position {(0, 4, 7), (0, 5, 7), (0, 6, 7)} and all of its (*k-1*)-subtrees 'b c' and 'b b' are also frequent. Similarly, when induced subtree is considered, C_L is also frequent as there are two occurrences of induced subtrees that occur at position {(0, 5, 7), (0, 6, 7)} and all of its (*k-1*)-subtrees 'b c' and 'b b' are also frequent. To show an example of a pseudo-frequent subtree we can now extend C_L with a node at position 8 so that we obtain a $C_{L'}$ where *L'*:*L*+'*e*' = '*b c / b e*'. In Figure 4, we show $C_{L'}$ with all its valid (*k-1*)-subtrees *S1*, *S2*, and *S3*. From Figure 4 we can see that subtree *S3* is infrequent since it occurs only once at position (0, 7, 8). Therefore, in the light of definition 6, $C_{L'}$ is a *pseudo-frequent* candidate subtree because one of its (*k-1*)-subtree is infrequent, i.e. subtree *S3*.

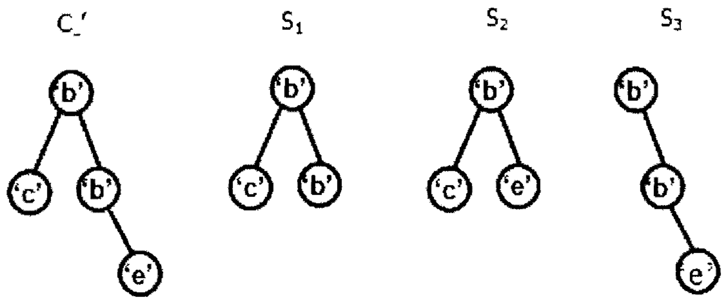


Figure 4: pseudo frequent subtree CL' (at support = 2) and all its *k-1* subtrees

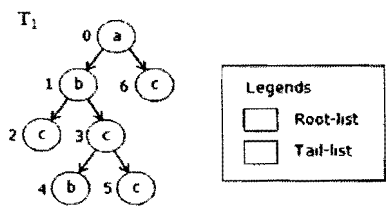
Subsequently, since *pseudo-frequent* candidate subtrees exist, according to Lemma 1, the *anti-monotone* property does not hold for frequent subtree mining when *occurrence-match* support is used. Hence, in the case where there exists a frequent subtree *s* with one or more of its subtrees infrequent, then *s* also needs to be considered infrequent for the *anti-monotone* property to hold. Tree structured data has a hierarchical structure where 1-to-many relationships can occur, as opposed to relational data where only 1-to-1 relationships exist between the items in each transaction. This multiplication between one node to its many children/descendants makes the *anti-monotone* property not hold for tree structured data. However, if *transaction-based* support is used no *pseudo-frequent* subtrees will be generated since the repetition of items is reported only once per transaction. This makes the 1-to-many relationship between a node to its children/descendants be treated as a set of items like in a relational database.

4. MB3-Miner Algorithm

This section provides an overview of the proposed approach for mining frequent embedded subtrees. We provide a short overview of the basic steps of the algorithm here and later in the section each step is explained in more detail. *Step 1*: As the input to the algorithm is a database of XML documents. For faster processing the database of XML documents is first transformed into a database of rooted integer-labeled ordered tree. *Step 2*: the tree database is traversed and a global sequence is created which stores each node in the pre-order traversal together with the necessary node information. The encountered node labels are hashed and the set of frequent (*k*)-subtrees is obtained. *Step 3*: the *embedding list* is created which for each node *n* in the *dictionary* stores *n*'s descendant nodes' hyperlinks in pre-order traversal ordering. At the same time the candidate subtrees' encodings are hashed which determines all the frequent *2*-subtrees. *Step 4*: *TMG* candidate generation using the *embedding list* generated in Step 3 takes place and for each $k>2$ the set of *k*-subtree candidates is hashed to the F_k hashtable. The coordinates of each *k*-subtree are stored in F_k and each *k*-subtree is extended one node at time, starting from the last node of its *right-most-path*, up to its root. This enumerates all embedded *k+1*-subtrees and the whole process repeats until all *k*-subtrees are enumerated and counted. At each *k* step we check that all the (*k-1*)-subtrees of the frequent *k*-subtree are also frequent in order to avoid the generation of *pseudo-frequent* subtrees. The details of how each step is performed and how the *maximum level of embedding* constraint is introduced are given in the sections that follow.

Step 1 - XML Data Pre-processing. Our previous algorithm, X3-Miner [Tan et al. 2005a], represents a database of XML documents as a database of rooted string-labeled ordered trees. When doing frequency counting using a hashtable, processing integer-labeled trees has a computational advantage over string-labeled trees, especially when the labels are long [Tan et al. 2005a, 2005b]. To expedite the frequency counting, the database of XML documents can be 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]. Each tag in an XML document can be encoded as an integer. Each integer will identify each tag uniquely. To encode a particular tree, these integers are used in the same way that string labels were used in the string encoding explained in Section 3. The only difference is that the backtrack ("/") symbol(s) is replaced by a negative integer indicating the number of backtrack symbols ("/") occurring at that place in the encoding. For example, if labels b, c and e are mapped to integers 1, 2 and 3, respectively, then from Figure 1, $\varphi(T1)$:'*1 2 -1 1 3* ' and $\varphi(T3)$:'*1 3 -1 2* ', and from Figure 3, $\varphi(T1)$:'*1 1 -1 3 3 -1 2 -2 2 -1 2 -1 1 1 3*'. For each XML tag, we consider tagname, attribute(s) and value(s). Hence, each unique system-generated integer corresponds to each unique tag combination. 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.

Step 2 - 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 *dictionary* consists of each node in T_{db} following the pre-order traversal indexing. For each node its *position*, *label*, *scope*, and *parent* position are stored. The scope of a node refers to the position of its *right-most-leaf* node or its own position if it is a leaf node itself. An item in the *dictionary D* at position *i* is referred as *D[i]*. The notion of position of an item refers to its index position in the *dictionary*. The purpose of the *dictionary* is to provide a shared global nodes' related information that allows for direct access and thereby avoid the space cost which would be caused if this information is copied (stored) locally for every occurrence of a node [in the embedded list] (see Figure 5 for an example). For iMB3, each node in *dictionary* contains additional level information of a node. When generating the *dictionary*, we compute all the frequent 1-subtrees. The set of all frequent 1-subtrees are denoted by F_1. After the *dictionary* is constructed our approach does not require further database scanning.

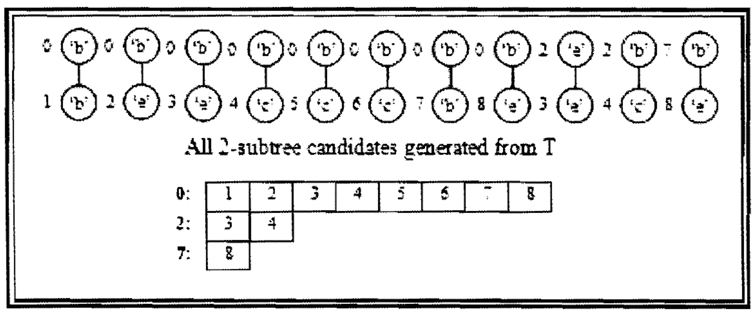


0	1	2	3	4	5	6
1	2	3	4	5		
2	φ					
3	4	5				
4	φ					
5	φ					
6	φ					

[0]	[1]	[2]	[3]	[4]	[5]	[6]
a,6,-1	b,5,0	c,2,1	c,5,1	b,4,3	c,5,3	c,6,0

Figure 5: *EL* and the *dictionary* (label, scope, parent position) of tree *T1*

Step 3 - Constructing Embedding List (*EL*). In this section we describe the process of constructing the *embedding list* which allows for an efficient implementation of the *TMG* candidate enumeration. For each frequent internal node in F_1, a list is generated which stores its descendant nodes' hyperlinks [Wang et al. 2004] 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 nodes 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* (Figure 6). Hereafter, we call this list an *embedding list* (*EL*).



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

Figure 6: the *EL* representation of T in Figure 1.

We use notation *i-EL* to refer to an embedded list of nodes at position *i*. The position of an item in *EL* is referred to as the *slot*. 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*. Figure 6 illustrates an example of the *EL* representation of tree T (Figure 1). In Figure 6, 0-*EL* for example refers to the list: 0:[1,2,3,4,5,6,7,8], 0-*EL*[0]=1 and 0-*EL*[6]=7. The pseudo code for *EL* construction is shown below.

```
Inputs: D (dictionary), σ (min. support), F_1 (frequent 1-subtrees)
Outputs: EL (embedded list), F_2 (frequent 2-subtrees)

ConstructEmbeddingList (F_1, D):
  for each frequent 1-subtree t_1 ∈ F_1
    vol-t_1 = GetVOL(t_1)  // returns a list of coordinates where t_1 occurs
    for each occurrence coordinate oc ∈ vol-t_1
      {oc[0]-EL,C_2} = Generate-EL (oc[0], D)
      EL = EL ∪ oc[0]-EL
  for each 2-subtree t_2∈C_2
    if( support(t_2)≥σ ) F_2 = F_2 ∪ t_2
return EL, F_2

Generate-EL (i, D):
line 1: i-scope = GetScope(i, D)          // get scope of i
line 2: for ( j = i+1 to i-scope )
line 3: i-EL = i-EL + j                   // add j to i-EL
line 4: C_2 = C_2 ∪ Enumerate-Candidate(i, j)
return i-EL,C_2
```

Figure 7: *EL* and F_2 construction pseudo-code

Line 3 of Generate-*EL* procedure constructs embedded list of node *i*. Line 4, generates 2-subtree candidates rooted at node *i*. 2-subtree candidates are computed while the *EL* is constructed. An example of an *embedding list* and the corresponding *dictionary* is shown in Figure 5.

Occurrence Coordinate (OC). A candidate subtree can occur at different positions in the database and OC is used to denote the node positions of that particular subtree so that it can be distinguished from other subtrees having the same encoding. 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* $r:[e_1,\dots e_{k-1}]$; r refers to k-subtree root position and $e_1,\dots,e_{k-1}$ 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 the OC of 3-subtree (T2) with encoding '*b b e*' is encoded as 0:[6,7]; 4-subtrees T1 with encoding '*b c / b e*' are encoded as 0:[5,6,7], and so on. Each OC of a subtree describes an instance of each occurrence of the subtree in T_{db}. Hence, each *candidate instance* has an OC associated with it.

The scope of extension of a node. We denote the range of nodes that can be appended to that node for the formation of new candidate subtrees as the *scope of extension* of a node (Figure 8). The *EL* representation preserves the ordering as well as the embedding relationships of nodes in a tree. *i-EL* defines the scope of extension of node i and it spans from i-EL[0] to i-EL[j] where j = |i-EL|-1. We refer to the first scope extension position as the *left-most scope* and the last as the *right-most scope.* Consequently, given a 4-subtree T with occurrence coordinate 1:[3,4,5], the *left-most* scope of T is defined by 1-EL[3] and the *right-most* scope of T is defined by 1-EL[5]. An occurrence coordinate of a valid candidate is defined by $r:[m,\dots n]$ where $m < n$. Thus, a valid candidate has an increasing scope ordering such that r-EL[m] < r-EL[n].

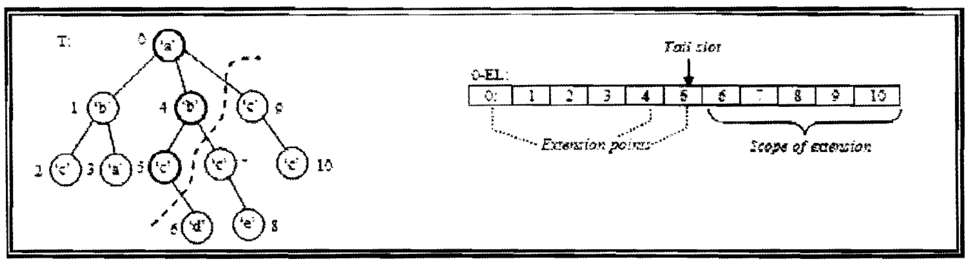


Figure 8: *TMG* enumeration: extending (k-1)-subtree t_{k-1} where $\varphi(t_{k-1})$: '*a b c*' occurs at position (*0, 4,5*) with nodes at positions 6, 7, 8, 9, and 10

<u>**Step 4 – Generating Candidate Subtree.**</u> 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, in the sense that it generates all possible candidate subtrees from

a given database of trees. We utilize the *TMG* [Tan et al. 2005a] candidate generation approach for an optimal, non-redundant [Chi et al. 2005; Tan et al. 2005b; Zaki 2005] candidate subtree enumeration. Our candidate generation approach makes efficient use of the novel *embedding list* representation.

***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) [Tan et al. 2005b; Zaki 2005]. To enumerate all embedded k-subtrees from a $(k-1)$-subtree, the *TMG* enumeration approach extends one node at a time to the *right-most-path* of $(k-1)$-subtree as illustrated in Figure 8. We refer to each node in the *right-most-path* as an extension point. One important property of *EL* is that the positions of nodes are stored in pre-order manner. Hence, given a $(k-1)$-subtree with known tail slot, the subsequent slots in *EL* will form the *scope of extension* from i to j. All embedded k-subtrees are generated by attaching a node at position i to j to the $(k-1)$-subtree. Suppose $l(i)$ 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, and occurrence coordinate $r:[m,\ldots,n]$, k-subtrees are generated by extending a subtree t_{k-1} with $j \in r\text{-}EL$ such that $t<j\leq|r\text{-}EL|-1$. Thus its occurrence coordinate becomes $r:[m,\ldots,n,j]$ and its encoding becomes $L':L+l(i)$ where $i=r\text{-}EL[j]$ and $m<n<j$.

Pruning. In this section we discuss the importance of pruning when *occurrence-match* support is considered. As previously discussed in Section 3, when using *occurrence-match* support there can be *pseudo-frequent* candidate subtrees generated when generating k-subtrees from $(k-1)$-subtrees. To make sure that all generated subtrees do not contain infrequent subtrees, full $(k-1)$ pruning must be performed. The rationale of this has been discussed in Tan et al. [2005b] and Zaki [2005]. From this point onward we refer to full $(k-1)$ pruning as *full pruning*. This implies that at most $(k-1)$ numbers of $(k-1)$-subtrees need to be generated from the currently expanding k-subtrees. An exception is made whenever the δ constraint is set to 1, i.e. mining induced subtree, as in this case we only need to generate l numbers of $(k-1)$-subtrees where $l < (k-1)$ and l is equal to the number of leaf nodes in k-subtrees. If the removal of the root node of the k-subtree does not generate a *forest* [Zaki 2005], 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 if 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. While *full pruning* is easily done in a BFS based method, it is a challenge for a Depth-First-Search (DFS) based approach such as VTreeMiner (VTM). When generating a k-subtree using the DFS traversal method information regarding the frequency of its (*k-1*)-subtrees may not be available at that time, whereas with the BFS method the frequency of all its (*k-1*)-subtrees has been determined. Therefore, *full pruning* can be done in a more complete way in the BFS approach than in the DFS approach. Because of this difficulty, a DFS approach such as VTM [Zaki 2005] is forced to employ an *opportunistic pruning* strategy that only prunes infrequent subtrees in an opportunistic way. On the other hand, the DFS method is a more space efficient approach compared to the BFS method. A DFS traversal 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 BFS method will need to store the occurrence coordinate of generated (*k*)-subtrees which is later used for generating (*k*+1)-subtrees from the same transaction.

Accelerating (*k-1*) pruning. 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 only 1 comparison is required.

Candidate subtree counting. In the candidate enumeration step, the process utilizes the notion of a coordinate. 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 with encoding *L* has a frequency *n* if there are *n* instances of subtrees with the same encoding *L*, 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 the subtree by its encoding. 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 $|VOL(L)|$. We can use VOL to count the *occurrence-match* support and *transaction-based* support. For *occurrence-match* support we suppress the notion of the transaction id (*tid*) that is associated with each occurrence coordinate. For *transaction-based* support the notion of *tid* of each occurrence coordinate is accounted for when determining the support. As an example when the *occurrence-match* support is used, the frequency of a subtree of tree *T* (Figure 1) with encoding '*b c / e*', denoted by |*VOL*('*b c / e*')| is equal to the size of the *VOL*, i.e. 3 (Figure 9a). When *transaction-based* support is used the |*VOL*('*b c / e*')| is equal to 1 because from the *transaction-based* support definition in Section 3, the support of a subtree *t* is equal to the numbers of transactions that support subtree *t*. From the example in Figure 9b there is only 1 transaction (*tid*:0) that supports subtree 'b c / e' of tree *T* (Figure 1).

0	6	8
0	5	8
0	4	8
'b c / e'		

(a) when *occurrence-match* support is used

0	0	6	8
0	0	5	8
0	0	4	8
'b c / e'			

(b) when *transaction-based* support is used

Figure 9: pictures of VOL('*b c / e*') of 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. Thus space compression is an important issue to be kept into a perspective, especially for a BFS method like ours. A strategy that allows the utilization of smaller size occurrence coordinates can further help improving the proposed technique. Our approach constructs and stores full coordinates of subtrees in memory to perform the *TMG* enumeration completely and help the pruning process. As described earlier in Section 3, the *TMG* enumeration is a specialization of the *right-most-path* extension method. Essentially the only information it requires for performing enumeration is the *right-most-path* coordinates of subtrees. However, due to the time and space limitation we will leave the investigation of storing only the *right-most-path* coordinates for enumeration and frequency counting in our future work. 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),Φ(max. level of embedding)
Outputs : $F_k$(Frequent subtrees), D(dictionary)
{D, $F_1$}  : DatabaseScanning ($T_{db}$)
{EL, $F_2$} : ConstructEmbeddedList ($F_1$,D,Φ)
k=3
while( $|F_k| \geq 0$ )
  $F_k$ = GenerateCandidateSubtrees($F_{k-1}$,Φ)
  k = k+1

GenerateCandidateSubtrees($F_{k-1}$,Φ):
for each frequent k-subtree $t_{k-1} \in F_{k-1}$
  $L_{k-1}$ = GetEncoding ($t_{k-1}$)
  VOL-$t_{k-1}$ = GetVOL($t_{k-1}$)
  for each occurrence coordinate $oc_{k-1}$ (r:[m,...n]) $\in$ 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( hashkey($L_k$),$oc_k$,$F_k$ )
        else
            If(k-1Pruning ($L_k$) = false) // if all k-1 patterns frequent
                 Insert( hashkey($L_k$),$oc_k$,$F_k$ )
return $F_k$
```

Figure 10: pseudo-code of iMB3-Miner

5. TMG Mathematical Model

In this section, we present the mathematical model of the *TMG* approach for enumerating embedded subtrees. The *TMG* enumeration approach belongs to the family of the horizontal enumeration approaches. It is an optimal enumeration strategy as it always generates unique subtrees (non-redundant) and it exhausts the search space completely. The unique subtrees generated by the *TMG* enumeration approach refer to instances of subtrees in the tree database. Unique instances of subtrees have unique occurrence coordinate, however, it may have the same encoding. Due to the fact that the *TMG* enumeration approach is optimal and all candidate subtrees enumerated by the *TMG* approach are valid, it is most likely that any other horizontal enumeration approach would need to enumerate at least as many candidates if not more. Throughout this chapter, we assume that all candidate subtrees generated are of embedded subtree type.

There is no simple way to parameterize a tree structure unless it is specified as a *uniform tree* (Section 3). The *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. Thus, the worst case complexity of enumerating embedded subtrees from any arbitrary trees is given by their *closed form* (Section 3). We denote a *uniform tree* as *T(n,r)* where *n* refers to its height and *r* refers to the degree of every node in the tree. The size of a *uniform tree T(n,r)* can be computed by counting the number of nodes at each depth. For a *uniform tree* with degree *r* there will be r^d numbers of nodes at depth *d*. Hence, there are $r^0 + r^1 + r^2 + \ldots + r^n$ numbers of nodes in a *uniform tree T(n,r)*. This can be computed using the geometrical series formula $(1-r^{n-1})/(1-r)$. When the root node is omitted the following formula is used, $r(r^n-1)/(r-1)$. If $r = 1$, the size of the *uniform tree* is equal to its height *n* and it becomes a sequence.

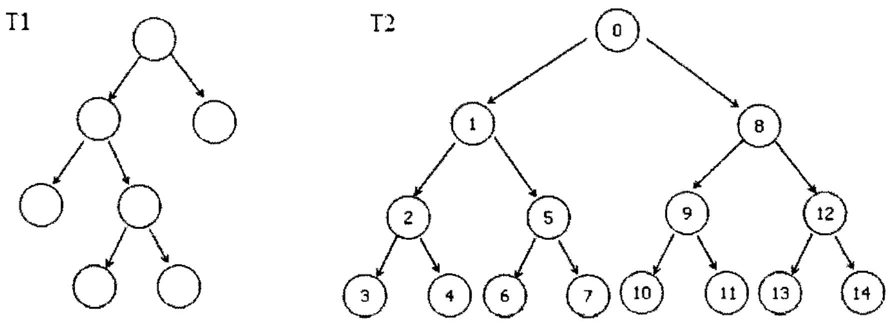


Figure 11: example of an arbitrary tree T1 and its closed form T2 (3,2)

The task of frequent subtree mining is to discover all candidate subtrees whose support is equal or greater than the user-specified minimum support σ. Since we are considering labeled trees, to discover such candidate subtrees, we have to count their support by counting the number of occurrences of the candidate subtrees that have the same string encoding. This means that for one candidate subtree with an encoding φ, there can be many instances of this subtree with the same encoding. Henceforth, we refer to an instance of a candidate subtree as a *candidate (subtree) instance*.

Given that the *TMG* enumeration approach uses the structural aspect of tree structures to guide the enumeration of subtrees, the enumeration complexity of the *TMG* enumeration approach is bounded by the height and degree of tree structures rather than by their label set. For the problem of mining frequent subtrees most of the time and space complexity comes from the candidate enumeration and counting phase. Thus, we define the cost of enumeration (complexity) as the number of candidate instances enumerated throughout the candidate generation process as opposed to the number of candidate subtrees generated.

The mathematical model of the *TMG* enumeration is formulated as follows. Given a *uniform tree* $T(n,r)$, the worst case complexity of candidate generation of $T(n,r)$ is expressed mathematically in terms of its height n and degree r.

Complexity of 1-subtree enumeration Since there are $|T|$ number of candidate 1-subtree instances that can be enumerated from a *uniform* tree $T(n,r)$, the complexity of 1-subtree enumeration, denoted as $||T_1||$, is equal to the size of the tree $|T(n,r)|$.

Complexity of 2-subtree enumeration. Earlier in the paper, we mentioned that the construction of the *EL* can be constructed by joining all the 2-subtree candidates that have common root node position and inserting each leaf node in the list with the same *root key*. In other words, all 2-subtrees with root key n are enumerated by constructing the *EL* with root key n and the size of the n-*EL* equates to the number of 2-subtrees generated with the root key n . Therefore, the total sum of the lists size in the *EL* reflects the total number of 2-subtree candidates and the complexity of generating the *EL* would be equal to the complexity of enumerating 2-subtrees.

Let s be a set with n objects. The 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). The combination ${}_sC_k$ formula is given by $s!/(s-k)!k!$. Thus, for 2-subtree enumeration, the following relation exists. Let an r-*EL* consist of l number of items; each item 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 a corollary, the complexity of 2-subtree enumeration, denoted as $||T_2||$, of tree T with size $|T(n,r)|$ is equal to the sum of all generated 2-subtree candidates given by exp. 1.

$$\sum_{r=0}^{|T(n,r)|-1} {}_{|r-EL|}C_1 \qquad \text{exp. 1}$$

From expression 1 since ${}_{|EL[r]|}C_1$ is equal to $|r\text{-}EL|$ and $|0\text{-}EL|=|T(n,r)|$, and we sum $|r\text{-}EL|$ for $r=0,\ldots,|T(n,r)|-1$. It appears that the complexity of 2-subtree enumeration is $O(|T(n,r)|^2)$. However, since the $|l\text{-}EL|$ is zero for $l \in L_T$ and L_T is a set of leaf nodes in T, the complexity of 2-subtree enumeration of a *uniform tree* $T(n,r)$ is $\leq O(|T(n,r)|^2-r^n)$ since there are r^n leaf nodes in a *uniform tree* $T(n,r)$. Furthermore, the size of each $|r\text{-}EL|$, for $r=0,\ldots,|T(n,r)|-1$ is not equal, i.e. $|0\text{-}EL|<|1\text{-}EL|$ and $|0\text{-}EL|<|2\text{-}EL|$, $|0\text{-}EL|<|(T(n,r)-1)\text{-}EL|$, and so on. Also, for the case of a *uniform tree* $T(n,r)$, given that ${}_d\Upsilon_p$:$\{c_1,\ldots,c_r\}$ is a set of the pre-order positions of the children nodes of node P at position p and all nodes in ${}_d\Upsilon_p$:$\{c_1,\ldots,c_r\}$ have depth d, the sum of $|c_1\text{-}EL|,\ldots, |c_r\text{-}EL|$ is equal to $|p\text{-}EL|-(r^1+\ldots+r^d)$ for

$d < n$-1. Therefore, the sum of $|r\text{-}EL|$ for r=0,...,$|T(n,r)|$-1 can be computed by $|0\text{-}EL|+|0\text{-}EL|-r^1+|0\text{-}EL|-(r^1+r^2)+|0\text{-}EL|-(r^1+r^2+r^3)+\ldots+|0\text{-}EL|-(r^1+r^2+\ldots+r^{n-1})$, where $|0\text{-}EL|=|T(n,r)|-1$. Suppose that our aim is to express the above expression in a form of $C.(A)$-M, and $r^1+\ldots+r^d$ can be computed by the geometrical series formula $r(r^d-1)/(r-1)$, the sum of $|r\text{-}EL|$ for r=0,...,$|T(n,r)|$-1 can be given by eq. 2, as follows:

$$\sum_{i=0}^{|T(n,r)|-1} |\, i - EL\,| = n.(|T(n,r)| - 1) - \sum_{i=1}^{n-1} \frac{r(r^i - 1)}{r-1} \qquad \text{eq. 2}$$

Since the complexity of generating 2-subtrees is equal to the sum of the size of $|r\text{-}EL|$ for r=0,...,$|T(n,r)|$-1, we can infer that the complexity of generating 2-subtrees is also equal to eq. 2.

Since $n.|T(n,r)| > n.(|T(n,r)|-1) \geq n.(|T(n,r)| - 1) - \sum_{i=1}^{n-1} \frac{r(r^i - 1)}{r-1}$, eq. 2 can be approximated by $n.|T(n,r)|$.

In contrary, the complexity of 2-subtree enumeration of the join approach is $O(|T(n,r)|.|T(n,r)|)$, assuming that each label of the *uniform tree* $T(n,r)$ is unique. Given that $n < |T(n,r)|$ and $r \neq 1$, the following relation is always true: $n.|T(n,r)| < |T(n,r)|.|T(n,r)|$. In the case when $r = 1$, $|T(n,1)| = n$, thus $n.|T(n,r)| = |T(n,r)|.|T(n,r)|$. However, from eq. 2 the complexity of our approach will still be less than the join approach due to the additional term $\sum_{i=1}^{n-1} \frac{r(r^i - 1)}{r-1}$ being subtracted.

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:[e_1,...,e_{k-1}]) rooted at *r* is equal to the number of combinations of *l* nodes from *i-EL* having (*k-1*) elements each. In Section 4, a valid occurrence coordinate of valid candidates has the property that $e_l < e_{k-1}$. 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:

$$\sum_{r=0}^{|T(n,r)|-1} {}_{|r-EL|}C_{k-1} \qquad \text{exp. 3}$$

In expression (exp) 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 the uniform tree T(n, r). ${}_r\delta^{n-d}$ denotes the size of embedded list of node *i* with depth *d* of a uniform tree *T*(*n*,*r*). By the definition in Section 3, ${}_r\delta^{n-d}$, is described by the 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 ${}_r\delta^{n-d}$, as given by exp. 4.

$$r^d {}_r\delta^{n-d}$$

Using the fact that for each level in *T*(*n*,*r*) there are r^d number of lists that have the same size ${}_r\delta^{n-d}$, exp. 3 can be expressed as shown below in exp. 5, summed over *n* levels.

$$r^0 {}_{{}_r\delta^n}C_{k-1} + r^1 {}_{{}_r\delta^{n-1}}C_{k-1} + \ldots + r^n {}_{{}_r\delta^0}C_{k-1} \qquad \text{exp. 5}$$

Further, exp. 5 can be written as follows.

$$\sum_{i=0}^{n-1} r^i {}_{{}_r\delta^{n-i}}C_{k-1}, \text{ for } {}_r\delta^{n-i} \geq (k-1) \qquad \text{exp. 6}$$

Substituting ${}_r\delta^{n-d}$ with $r(r^{(n-d)}-1)/(r-1)$ in exp. 6 gives us exp. 7.

$$\sum_{i=0}^{n-1} r^i {}_{\frac{r(r^{n-i}-1)}{r-1}}C_{k-1}, \text{ for } \frac{r(r^{n-i}-1)}{r-1} \geq (k-1) \qquad \text{exp. 7}$$

Please note that if the $|EL| < (k-1)$, no candidate subtrees would be generated, thus the constraint ${}_r\delta^{n-1} \geq (k-1)$ takes care of this condition. Hence, using the expressions developed, the complexity of total *k*-subtree candidates from a uniform tree *T*(*n*,*r*) for *k*=1,...,|*T*(*n*,*r*)| is given by eq. 8.

$$\sum_{k=1}^{|T(n,r)|} \|T(n,r)\|_k = \|T(n,r)\|_1 + \sum_{k=2}^{|T(n,r)|} \|T(n,r)\|_k \qquad \text{eq. 8}$$

From exp. 7, the second term of eq. 8 can be further expanded as follows in exp. 9.

$$\sum_{k=2}^{|T(n,r)|} \|T(n,r)\|_k = r^0 {}_{\frac{r(r^n-1)}{r-1}}C_{k-1} + \ldots + r^{n-1} {}_rC_{k-1}, \text{for } \frac{\mathrm{r(r^{n-i}-1)}}{\mathrm{(r-1)}} \geq \mathrm{k-1}$$

$$\sum_{k=2}^{|T(n,r)|} \|T(n,r)\|_k = \sum_{i=0}^{n-1} r^i \left\{ \sum_{k=2}^{\frac{1-r^{n-i+1}}{1-r}} {}_{\frac{r(r^{n-i}-1)}{r-1}}C_{k-1} \right\}$$

$$\sum_{k=2}^{|T(n,r)|} \|T(n,r)\|_k = \sum_{i=0}^{n-1} r^i \left\{ \sum_{k=2}^{A} {}_BC_{k-1} \right\}, A = \frac{1-r^{n-i+1}}{1-r}; B = \frac{r(r^{n-i}-1)}{r-1} \qquad \text{exp. 9}$$

Finally, eq. 8 can be restated as follows:

$$\sum_{k=1}^{|T(n,r)|} \|T(n,r)\|_k = \frac{(1-r^{n+1})}{(1-r)} + \sum_{i=0}^{n-1} r^i \left\{ \sum_{k=2}^{\frac{1-r^{n-i+1}}{1-r}} {}_{\frac{r(r^{n-i}-1)}{r-1}} C_{k-1} \right\} \qquad \text{eq. 10}$$

Thus, given an arbitrary tree *T* and its closed form *T'*(*n*,*r*), the worst case complexity of enumerating embedded subtrees using the *TMG* approach from *T* can be computed using eq. 10 where *n* is the height of *T'* and *r* is the degree of *T'*. Suppose you have a complete tree with degree 2 and height 3 denoted by *T*(*3,2*), using eq. 10, we could compute that the enumeration cost for generating all possible subtrees is 16,536, i.e. there are 16,536 subtrees enumerated. When the height of the tree is increased by 1, *T*(*4,2*), the enumeration cost for generating all possible subtrees is 1,073,774,896. Further, if we increase the degree by 1, *T*(*3,3*), the number of subtrees generated blows up to 549,755,826,275. Although *TMG* is an optimal enumeration approach the formula clearly demonstrates that the complexity of generating embedded subtrees from a complete tree structure can be intractable. It also suggests that the worst case complexity of enumerating all possible candidates from data in a tree structure form is mainly determined by the structure of the tree (*height* and *degree*).

Figure 12 shows the enumeration cost graph of a uniform tree *T*(*3,2*). The produced curve is not exactly symmetric, i.e. the left hand side of the curve (from the beginning to the middle of the curve) has slightly higher enumeration cost than the right hand side of the curve (from the middle to the end of the curve).

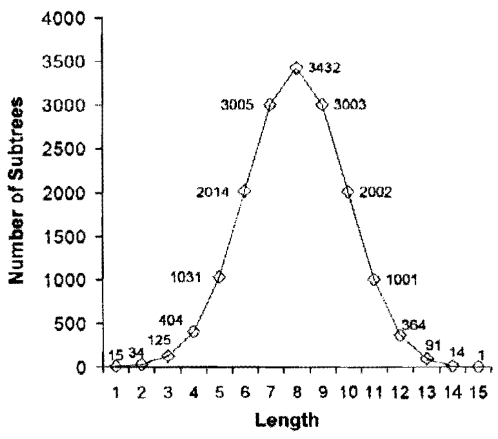


Figure 12: enumeration cost graph of uniform tree *T*(*3,2*)

It is interesting to analyze the curve produced by eq. 10 above for *T(3,2)*. To help us understand the curve we will use the *embedding list* representation of *T(3,2)* as shown in Figure 13. Please see Figure 11 for the topological structure of the uniform tree *T(3,2)*.

0:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1:	2	3	4	5	6	7								
2:	3	4												
5:	6	7												
8:	9	10	11	12	13	14								
9:	10	11												
12:	13	14												

Figure 13: *embedding list* (*EL*) of a uniform tree *T(3,2)*

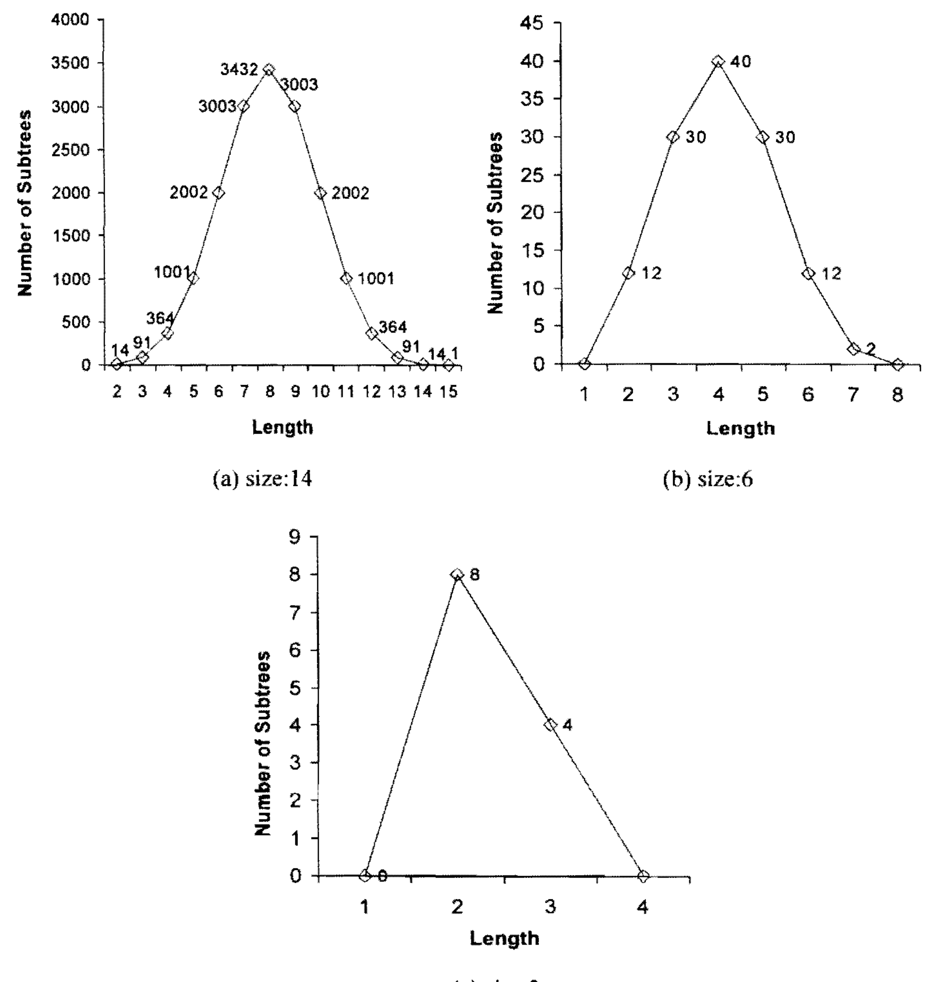
To illustrate how the enumeration cost of *T(3,2)* can be computed, using exp. 7, we show the computation for k=2,3, and 4 and leave out the computation for $k > 4$ as this can be easily obtained using the same procedure.

$$\|T(3,2)\|_2 = 2^0\,{}_{14}C_1 + 2^1\,{}_6C_1 + 2^2\,{}_2C_1 = 34$$

$$\|T(3,2)\|_3 = 2^0\,{}_{14}C_2 + 2^1\,{}_6C_2 + 2^2\,{}_2C_2 = 125$$

$$\|T(3,2)\|_4 = 2^0\,{}_{14}C_3 + 2^1\,{}_6C_3 = 404$$

The enumeration cost is governed by the size of each list in *EL*. Intuitively one can see that for k=4, 2-*EL*, 5-*EL*, 9-*EL*, 12-*EL* do not contribute anymore to the enumeration cost of *T(3,2)* since their size is less than (*k-1*) as explained above (exp. 6 and 7). For the same reason, 1-*EL* and 8-*EL* would cease to contribute to the enumeration cost of *T(3,2)* for $k > 7$. Then, for $7 < k \leq 15$ the enumeration cost simply comes from 0-*EL*. If the lists in *EL* (Figure 13) are grouped by their size, 3 different categories are obtained. The graphs in Figure 14 are obtained by plotting the enumeration cost for those 3 different categories separately.



(a) size:14 (b) size:6 (c) size:2

Figure 14: enumeration cost of $T(3,2)$ from *embedding list* in Figure 13

Exp. 1 suggests that the complexity of enumerating 2-subtrees from a tree T using the *TMG* approach is equal to the sum of each list size in *EL* of T. In other words, the complexity of enumerating 2-subtrees from a uniform tree $T(n,r)$ using the *TMG* approach, is bounded by $O(C.|T(n,r)|)$ where C is the height of the uniform tree $T(n,r)$ and $n < |T(n,r)|$. Using the join approach [Zaki 2005], the complexity of enumerating 2-subtrees is bounded by $O(|T(n,r)|*|T(n,r)|)$. Thus, the *TMG* approach using the tree model guided concept has a lower complexity for enumerating 2-subtrees. However, it is rather difficult to compare the *TMG* approach with the join approach for enumerating k-subtrees for $k > 2$, as we do not have mathematical model for the join approach. For this particular reason, we will compare the two approaches directly through the experiment, which will be described in Section 7. In addition, the mathematical model of *TMG* approach suggests that the size of the database of trees is not the main determining factor for the complexity of enumerating embedded subtrees. The structural property of a tree structure plays a more important role here. Moreover, eq. 10 helps us to understand that it is far more difficult to mine frequent embedded subtrees from a single transaction of a uniform tree with a large height *n* and degree *r* than thousands of transactions of a uniform tree with a small height *n* and degree *r*. Let's consider an example. The enumeration cost of *T(3,2)* is 16,536 and the enumeration cost of *T(3,3)* is 549,755,826,275. In other words, the enumeration cost of a database of trees that contain 1 transaction of *T(3,3)* is comparable to the enumeration cost of a database of trees with 33,245,998 transactions of *T(3,2)*. This indicates that mining frequent embedded subtrees is a more difficult problem to solve than mining frequent sequence or induced subtrees.

To conclude this section it is worth mentioning that even though we have shown here that the complexity of the task can easily become infeasible, in reality the developed algorithms are still well scalable even for quite large databases. To be able to obtain a mathematical formula for worst case analysis of *TMG* candidate enumeration we had to assume a uniform tree with support set to one. In real world, the tree databases would have varying depth and degree as well as support thresholds would be set higher. The number of candidate subtrees to be enumerated would reduce since many infrequent candidates would be pruned throughout the process. As the frequency distribution can generally not be known beforehand the support threshold could not be integrated into the *TMG* mathematical formula. At this stage, we aim at obtaining some insight into the worst case complexity of the task and in future we will strive to obtain an analysis on a more average case basis. Hence, despite the large complexity indicated by the formula, the developed tree mining algorithms are still well scalable for large databases of varying depth and degree, as is demonstrated in experiments provided in Section 7.

6. *iMB3-Miner - Mining Induced/embedded subtree*

In the previous section we discussed how the task of mining embedded subtrees can become infeasible, especially when large embeddings exist in the data. In this section we describe our approach to alleviate the problem by restricting the *maximum level of embedding*. In fact the distinctive characteristic between embedded subtrees and induced subtrees lies in the *level of embedding*. An induced subtree, as has been defined in Section 3 is an embedded subtree where the *maximum level of embedding* equals to 1. The mathematical model of *TMG* (eq. 10) can be used to determine the complexity of candidate generation. Whenever the complexity becomes intractable, we could restrict the *maximum level of embedding* of each subtree, which would produce a good estimate in a much shorter time. We demonstrate such a scenario in our experimental findings given in Section 7.

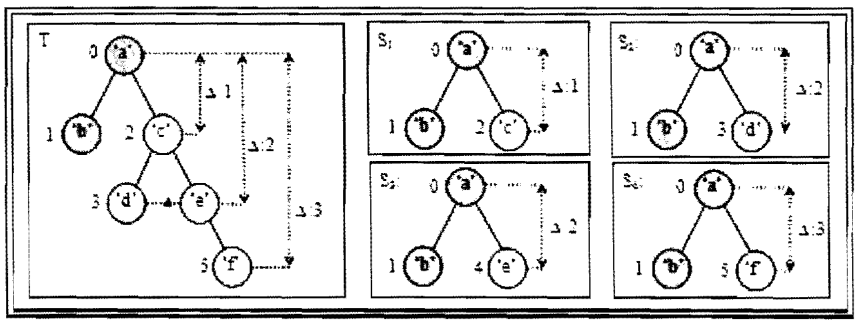


Figure 15: illustration of restricting the *maximum level of embedding* when generating *S1-4* subtrees from subtree '*a b*' with OC 0:[0,1] of tree *T*

Extensions to MB3. The originally developed MB3 algorithm only needs a slight adjustment when integrating the *maximum level of embedding* constraint. We simply avoid producing candidate subtrees where the *level of embedding* Δ between any two nodes is higher than the specified *maximum level of embedding* constraint threshold δ. To restrict the *level of embedding* of each node, at each extension a check is performed if the *level of embedding* is less than or equal to the specified *maximum level of embedding* constraint. Only when the *level of embedding* of a node to its extension point is less than the specified *maximum level of embedding* constraint, the extension is performed. From Figure 15 the *level of embedding* between nodes at position 0 with nodes with position 2, 3, 4 and 5, denoted by Δ(0,2), Δ(0,3), Δ(0,4), Δ(0,5) respectively, are indicated by the dotted line with arrow and marked with the *level of embedding* Δ specified on its right. For instance in Figure 15 the *level of embedding* between node at position 0 and node at position 5, denoted by Δ(0,5), in tree *T* is 3. Suppose that δ is set to 2, when we extend a subtree with OC 0:[0,1] with node with position 2, 3, and 4, the *level of embedding* between nodes at position 2, 3 and 4 to their extension point, Δ(0,2), Δ(0,3), Δ(0,4) respectively, are equal to 2 and it is less than or equals to the specified δ, and thus the candidate subtrees (*S1*, *S2*, and *S3*) are generated. However when it is extended with node at position 5 the *level of embedding* between node at position 5 to its extension point, Δ(0,5), is greater than 2 (δ), and hence the candidate subtree *S4* is not generated. According to our definition of induced and embedded subtree in Section 3, *S1* is an example of an induced subtree and *S2*, *S3*, and *S4* are examples of embedded subtrees. Here we show that by restricting the *maximum level of embedding* we can obtain different types of subtrees, induced and embedded subtrees.

7. Results and Discussions

We compare iMB3-Miner (iMB3), FREQT (FT) for mining induced subtrees and MB3-Miner (MB3), X3-Miner (X3), VTreeMiner (VTM) and PatternMatcher (PM) for mining embedded subtrees. We created an artificial database of trees with varying: max. size (s), max. height (h), max. fan-out (f), and number of transactions ($|T_r|$). Notation XXX–T, XXX-C, and XXX–F are used to denote execution time (including data preprocessing, variables declaration, etc), number of candidate subtrees |C|, and the number of frequent candidate subtrees |F| obtained from XXX approach respectively. Additionally, iMB3-(NP)-dx notation is used where x refers to the *maximum level of embedding* δ and (NP) is optionally used to indicate that *full pruning* is not performed. The minimum support σ is denoted as (sxx), where xx is the minimum frequency. *Occurrence-match* support was used for all algorithms unless it is indicated that the *transaction-based* support is used. In order to avoid redundant discussion, for each of the experiment we only provide a small discussion of the observed facts and a lengthier discussion is provided at the end of the section. Experiments were run on 3Ghz (Intel-CPU), 2Gb RAM, Mandrake 10.2 Linux machine and compilation were performed using GNU g++ (3.4.3) with –g and –O3 parameters. We use the Kudo's FT implementation [Kudo 2003] and disable the output mode so that it does not spend its execution time printing all the frequent subtrees when timing its performance.

Scalability (s:10,h:3,f:3). $|T_r|$ was varied to 100K, 500K and 1000K, with σ set to 25, 125 and 250, respectively. From Figure 16a we can see that all algorithms are scalable. MB3 outperforms VTM and PM for mining embedded subtrees and iMB3 outperforms FT for mining induced subtrees. Figure 16b shows the number of candidates generated by MB3, VTM and PM for $|T_r|$:1000K, σ:250. It can be seen that VTM and PM generate more candidates (VTM-C and PM-C) by using the join approach. The extra candidates are invalid, i.e. they do not conform to the tree model.

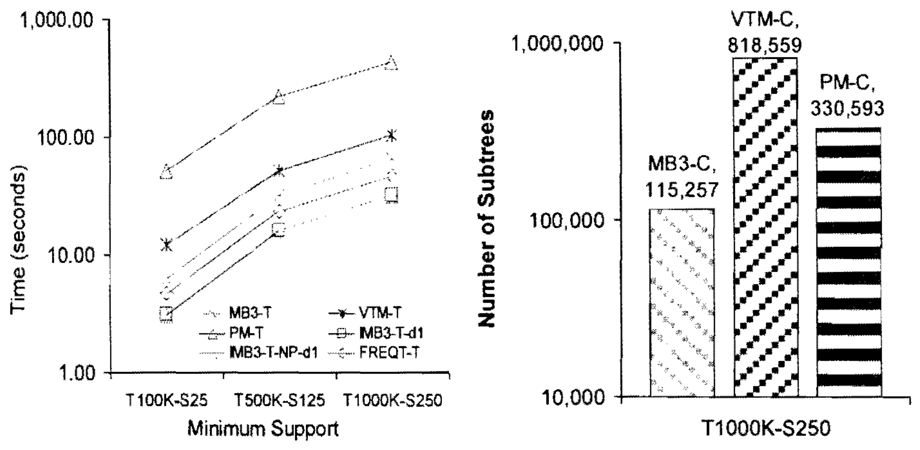


Figure 16: scalability test: (a) time performance (left) (b) number of subtrees |C| (right)

Pseudo-frequent (s:9,h:2,f:5,$|T_r|$:1). We created a datasets describing the tree shown in Figure 1 to illustrate the importance of *full pruning* when *occurrence-match* support is used. We set σ to 2 and compared the number of frequent subtrees generated by various algorithms. From Figure 17, we can see that the number of frequent subtrees detected by VTM (DFS) is larger when compared to PM, MB3 and X3 (BFS).

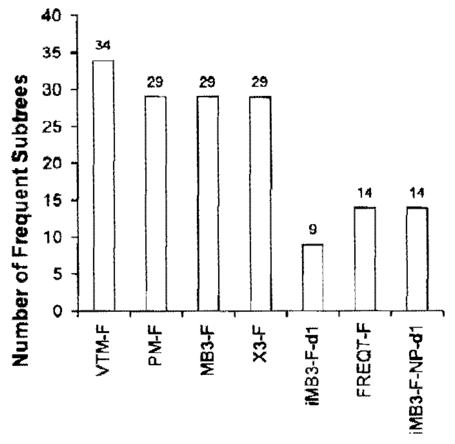
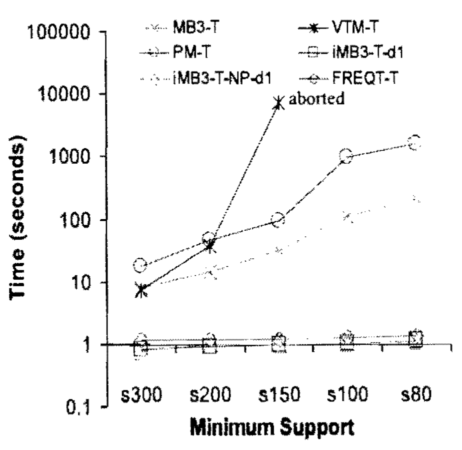


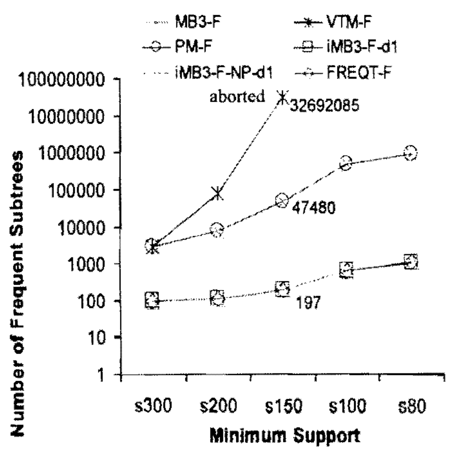
Figure 17: pseudo-frequent test: number of frequent subtrees |F|

The difference comes from the fact that the three BFS based algorithms perform *full pruning* whereas the DFS based approach such as VTM relies on opportunistic pruning which does not prune *pseudo-frequent* candidate subtrees. Figure 17 shows that FT and iMB3-NP generate more frequent induced subtrees in comparison to iMB3. This is because they do not perform *full pruning*, and as such generate extra *pseudo-frequent* subtrees.

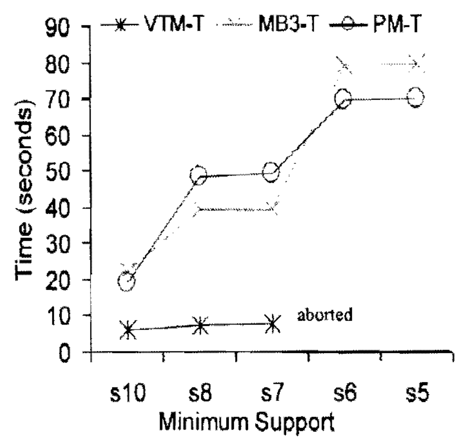
Deep Tree (s:28,h:17,f:3,$|T_r|$:10,000) vs Wide Tree (s:428,h:3,f:50,$|T_r|$:6,000). For deep tree (273,090 nodes), when comparing the algorithms for mining frequent embedded subtrees, MB3 has the best performance (Figure 18a). The reason for VTM aborting when $\sigma < 150$ can be seen in Figure 18b where the number of frequent subtrees increases significantly when σ is decreased. At σ:150, VTM generates a superfluous 688x more frequent subtrees compared to MB3 and PM. In regards to mining frequent induced subtrees, Figure 18a shows that iMB3 has a slightly better time performance than FT. At s80, FT starts to generate *pseudo-frequent* candidates.



(a) time performance



(b) number of frequent subtrees



(c) wide tree time performance

Figure 18: deep tree test

For wide tree (1,303,424 nodes), the DFS based approach like VTM outperforms MB3 as expected. However, VTM fails to finish the task when $\sigma < 7$, due to the extra number of *pseudo-frequent* subtrees generated throughout the process. In general, the DFS and BFS based approaches suffer from, deep and wide trees respectively. In Figure 18c we omit iMB3 and FT because the support threshold at which they produce interesting results is too low for embedded subtrees algorithms.

Prions (s:17,h:1,f:16,|T$_r$|:17551). This real world data describes a protein ontology database for Human Prion proteins in XML format [Sidhu and Dillon 2005]. For this dataset we map the XML tags to integer indexes similar to the format used in Zaki [2005]. The maximum height is 1. In this case all candidate subtrees generated by all algorithms would be induced subtrees. Figure 19a shows the time performance of different algorithms with varying σ. MB3 has the best time performance for this data.

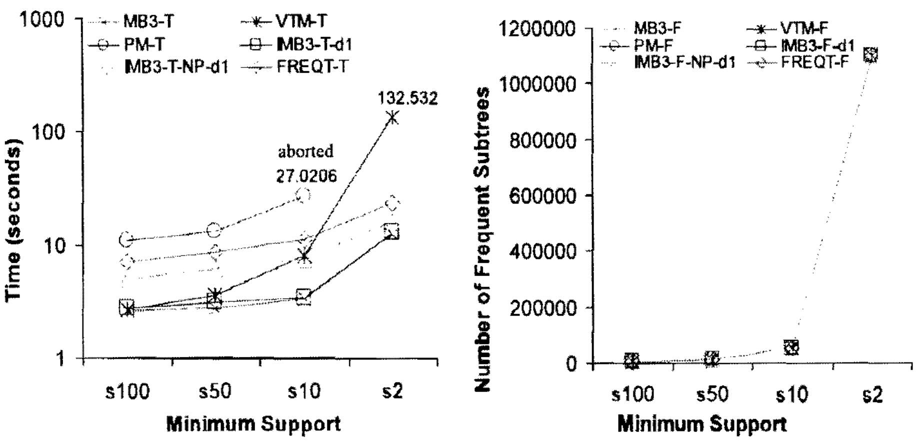
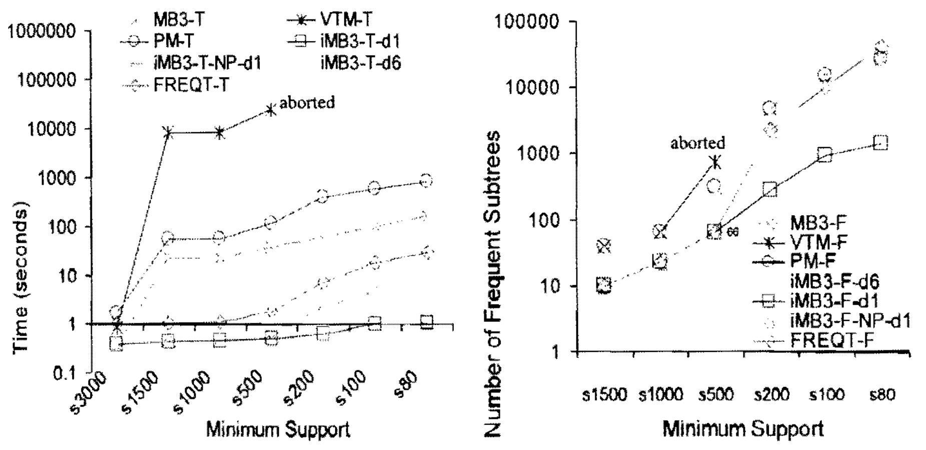


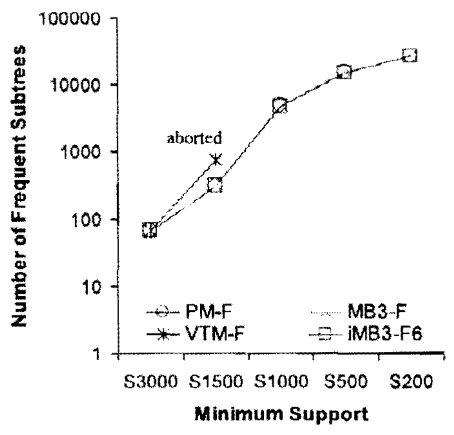
Figure 19: prions protein data: (a) time performance (left) (b) number of frequent subtrees (right)

Quite interestingly, with this dataset the number of frequent candidate subtrees generated is identical for all algorithms (Figure 19b). Another observation is that when σ < 10, PM aborts and VTM performs poorly. The rationale for this could be because the utilized join approach enumerates additional invalid subtrees. Note that original MB3 is faster than iMB3 due to additional checks performed to restrict the *maximum level of embedding*.

CSLogs (s:214,h:28,f:21). This dataset was previously used by Zaki [2005] to test VTM and PM using transactional support. When used for *occurrence-match* support, the tested algorithms had problems in returning frequent subtrees. This is a quite large dataset (|T$_r$|:59,691), and for *occurrence-match* support each of the transactions needs to be traversed to the full in order to count all occurrences of a subtree. Hence, during the process many more subtree occurrences have to be stored and processed which can cause memory problems. To handle such situations, one could make use of distributed parallel processing so that the tree database is split, and load balanced over a number of autonomous processing units. Merging of results would then take place to obtain a complete solution to the task. Alternatively, one can consider using secondary storage devices as additional storage resources. Assuming that the right mechanism is in place, with this approach, most of the processing can still be performed in the main memory and only when the space reaches a certain threshold, then some portions of the data in the main memory will be transferred to the secondary storage. We are certain that there are trade-offs with these solutions but we will not go into full details here and readers can use these ideas as possible solutions.



(a) time performance (b) number of frequent subtrees



(c) number of frequent subtrees for unconstrained vs constrained approach

Figure 20: test on 54% transactions of original CSLogs data (Zaki [2005])

The dataset was progressively reduced and at $|T_r|$:32,241 interesting results appeared. VTM aborts when $\sigma < 200$ due to numerous numbers of candidates generated. We demonstrate the usefulness of constraining the *maximum level of embedding* and provide results between the algorithms when σ is varied. From Figure 20b, we can see that the number of frequent subtrees generated by FT and iMB3-NP is identical. Both FT and iMB3-NP generate *pseudo-frequent* subtrees as they do not perform *full pruning.* Because of this, the number of frequent induced subtrees detected by FT and iMB3-NP can unexpectedly exceed the number of frequent embedded subtrees found by MB3 and PM (Figure 20b, s80). Figure 20a shows that both iMB3-NP and iMB3 outperform FT. A large time increase for FT and iMB3-NP is observed at s200 as a large number of *pseudo-frequent* subtrees are generated (Figure 20b). Secondly, we compare the results from VTM, PM and MB3 to the result obtained when the *maximum level of embedding* is restricted to 6 (iMB3-d6) (Figure 20c). By restricting the embedding level, we expect to decrease the execution time without missing many frequent subtrees. The complete set of frequent subtrees was detected at $\sigma \geq 200$, while only less than 2% were missed with $\sigma < 200$. Overall, MB3 and its variants have the best performance.

Mixed (deep and wide) dataset (s:428,h:17,f:50,|T$_r$|:76,000). Since the DFS approach and BFS approach suffer from, deep and wide trees respectively, it would be interesting to test the performance on a mixed dataset, which is both deep and wide. When comparing algorithms for mining embedded subtrees MB3 has the best performance as is shown in Figure 21.

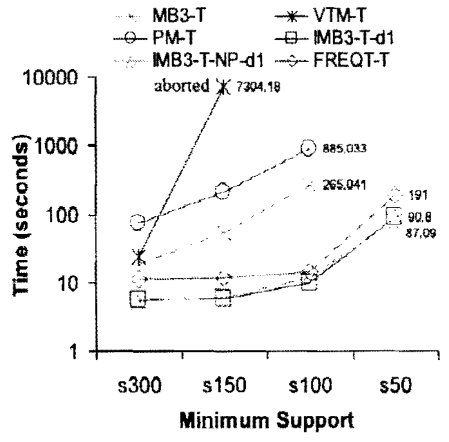
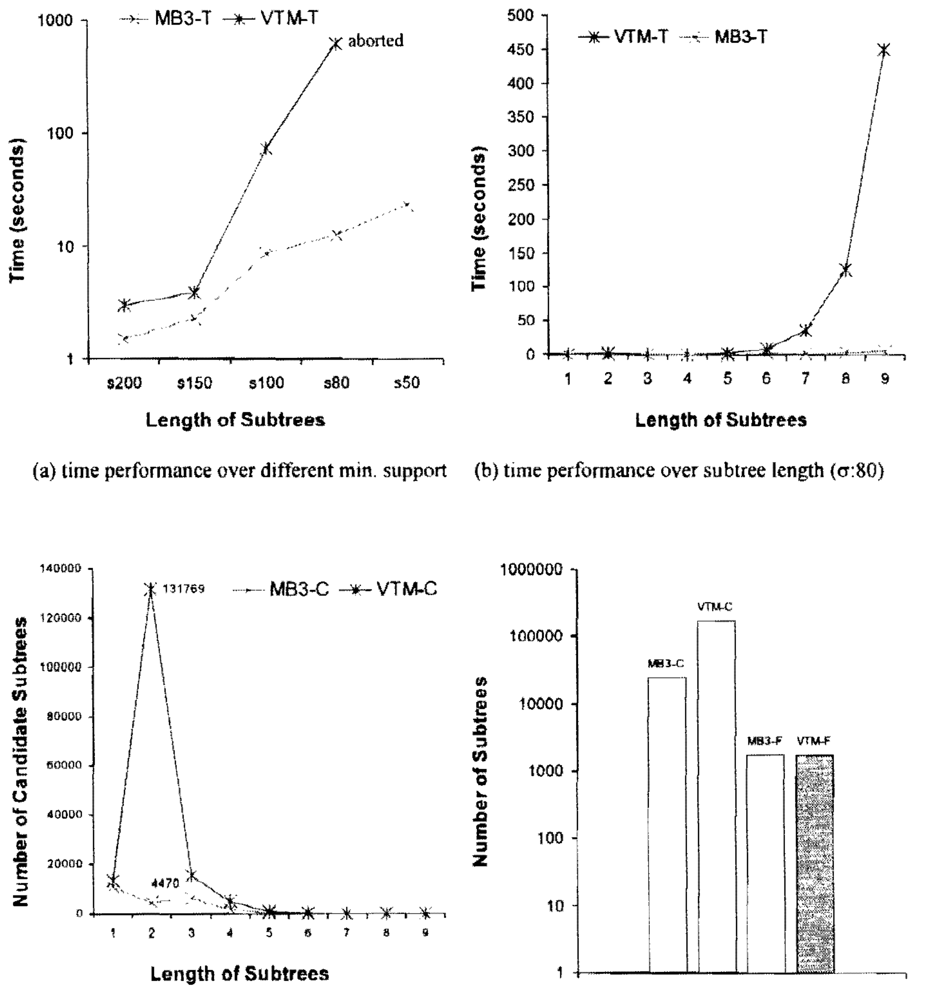


Figure 21: mixed dataset

VTM gets aborted when $\sigma < 150$, and the drawback of opportunistic pruning is even more noticeable. With regards to mining induced subtrees, iMB3 performs better than FT, the main reason being that FT does not perform full (*k-1*) subtrees and thereby enumerates additional subtrees as frequent.

Transaction-based support experiments. The paper focuses on the use of *occurrence-match* support. Previous experimental results show that in overall our approach performs better than other techniques when *occurrence-match* support is considered. In this experiment the comparison is made using the *transaction-based* support. As discussed earlier, our framework is flexible and generic enough to consider different support defintions. From Figure 22a it can be seen that MB3 performs better than VTM, and when σ is lowered to 50, VTM aborts. VTM performance degrades with the increase in subtree length, as is shown in Figure 22b. In Figure 22c we can see a spike of the total number of candidate 2-subtrees generated by the VTM. VTM generates 131,769 whereas MB3 only generates 4,470 candidate 2-subtrees.



(a) time performance over different min. support (b) time performance over subtree length (σ:80)

(c) number of candidate subtrees over subtree length (σ:80) (d) total number of generated subtrees (σ:80)

Figure 22: benchmarking the usage of *transaction-based* support for mining embedded subtrees

For generation of candidate 2-subtrees alone, VTM generates 29.47852 times more candidates in comparison to MB3. However, the total number of frequent subtrees produced by VTM and MB3 is identical, as evident from Figure 22d. The problem of generating *pseudo-frequent* subtrees, which was a major issue in our previous experiments, is eliminated here because the *transaction-based* support is considered. The flexibility inherent in our framework allows MB3 to swap from *occurrence-match* support to *transaction-based* support without a noticable performance penalty. The performance comparison for induced subtrees case can be seen from Figure 23. Overall iMB3 performs slightly better than FREQT.

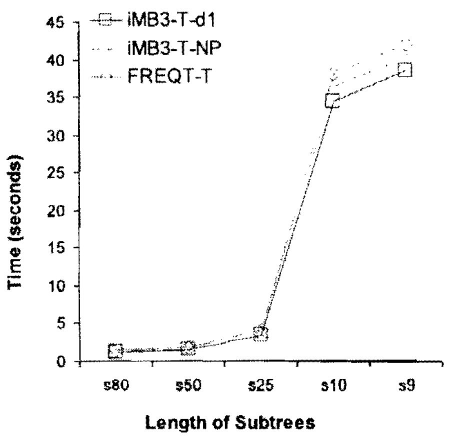


Figure 23: benchmarking the usage of *transaction-based* support for mining induced subtrees

Overall Discussion. MB3 and all its variants demonstrate high performance and scalability which comes from the efficient use of the *EL* representation and the optimal *TMG* approach that ensures only valid candidates are generated. The join approach utilized in VTM and PM could generate many invalid subtrees which degrades the performance. MB3 performs expensive *full pruning*, whereas VTM utilizes less expensive opportunistic pruning but suffers from the trade-off that it generates many *pseudo-frequent* candidate subtrees. This can cause memory blow-up and serious performance problems (Figure 18a, Figure 20 and Figure 21). This problem is evident in cases when VTM failed to finish for lower support thresholds. Some domains aim to acquire knowledge about exceptional events such as fraud, security attacks, terrorist detection, unusual responses to medical treatments and others. Often exceptional cases are one of the means by which the current common knowledge is extended in order to explain the irregularity expressed by the exception. In order to find the exceptional patterns the user needs to lower the support constraint as exceptional patterns are exceptional in the sense that they do not occur very often. It is therefore preferable that a frequent subtree mining algorithm is well scalable with respect to varying support.

In the context of association mining, regardless of which approach is used, for a given dataset with minimum support σ, the discovered frequent patterns should be identical and consistent. Assuming *pseudo-frequent* subtrees are infrequent, techniques that do not perform *full pruning* would have limited applicability to association rule mining. When representing subtrees, FT [Abe et al. 2002] uses string labels. VTM, PM, and MB3 (and its variants) use integer labels. As mentioned earlier, when a hashtable is used for candidate frequency counting, hashing integer labels is faster than hashing string labels especially for long patterns. As we can see, iMB3 and iMB3-NP always outperform FT. When experimenting with the *maximum level of embedding* constraint (Figure 20c), we have found that restricting the *maximum level of embedding* at a particular level leads to speed increases at the low cost of missing a very small percentage of frequent subtrees. This indicates that when dealing with very complex tree structures where it would be infeasible to generate all the embedded subtrees, a good estimate could be found by restricting the *maximum level of embedding*.

The flexibility inherent in our framework allows the MB3 and iMB3 algorithms to consider the *transaction-based* support with only a slight change to the way subtree occurrences are counted. Despite the fact that the implementation was not tailored for *transaction-based* support, our algorithms still exhibit a good performance when compared to other algorithms (Figure 22 and Figure 23),

8. Concluding Remarks

In this study we present technique to efficiently mining frequent embedded and induced subtrees from a database of XML documents. We proposed an efficient approach to tackle the complexity of mining embedded subtrees by utilizing a novel *embedding list* representation, *Tree Model Guided* (TMG) enumeration, and introducing the *maximum level of embedding* constraint. Representing a tree structure using *embedding list* has simplified the implementation of the *TMG* enumeration approach. Despite the fact that it requires more memory storage, we have shown that the *embedding list* can be very useful in the amount of information it provides. It allows us to easily formulate a precise mathematical model of the *TMG* candidate generation approach. Moreover, an additional benefit of such a representation is that it reduces the dimension complexity inherent in hierarchical tree structures. Throughout our experiments the *TMG* approach has proven to be more efficient than the join approach. Furthermore, using the developed mathematical model, one can predict the worst case scenario of mining frequent embedded subtrees from certain datasets in advance. When it is too costly to mine all frequent embedded subtrees, one can decrease the *maximum level of embedding* constraint gradually up to 1, from which all the obtained frequent subtrees are induced subtrees. Hence, it helps us to avoid difficult situations. The implications of using *occurrence-match* support instead of the simpler *transaction-based* support were investigated. High performance and scalability of the proposed approach was demonstrated in our experiments by contrasting it with the state of the art algorithms TreeMiner and FREQT. We use both synthetic and real datasets in the experimental studies. The results show the flexibility and efficiency of our approach when either support-definition was used.

Acknowledgements

A special thanks to Prof. M. J. Zaki [Zaki 2005] for providing us the TreeMiner source code and discussing the results obtained from it with us. We are also indebted to Mr. Amandeep S. Sidhu [Sidhu and Dillon 2005] for providing us with prions dataset. We would also like to thank the anonymous referees for providing many useful comments, which helped us improve the quality and clarity of this paper.

References

AGRAWAL, R. AND SRIKANT, R. 1994. Fast algorithm for mining association rules. In *Proceedings of the 20th Very Large Data Bases* (*VLDB 1994*), Santiago de Chile, Chile, 487-499.

AGRAWAL, R., AND IMIELINSKI, T., SWAMI, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (*SIGMOD 1993*), Washington, DC, USA, 207-216.

AGRAWAL, R., MANNILA, H., SRIKANT, R., TOIVONEN, H., VERKAMO, A.I. 1996. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, Ramasamy Uthurusamy, Eds. American Association for Artificial Intelligence, CA, USA, 307-328.

AGRAWAL, R. AND SRIKANT, R. 1995. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering* (*ICDE 1995*), Taipei, Taiwan, 3-14.

ABE, K., KAWASOE, S., ASAI, T., ARIMURA, H., ARIKAWA, S. 2002. Optimized substructure discovery for semistructured data. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery* (*PKDD 2002*), Helsinki, Finland, 1-14

BAYARDO, R. J. 1998. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD Conference on Management of Data* (*SIGMOD 1998*), Seattle, WA, USA, 85-93.

BODON, F. 2003. A fast apriori implementation. Informatics Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences.

BRIN, S., MOTWANI, R., ULLMAN, J. D., TSUR, S. 1997. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD Conference on Management of Data* (*SIGMOD 1997*), Arizona, USA, 255-264.

CHI, Y., YANG, Y., MUNTZ, R.R. 2004. HybridTreeMiner: An efficient algorihtm for mining frequent rooted trees and free trees using canonical forms. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, Santorini Island, Greece, 11-20.

CHI, Y., NIJSSEN, S., MUNTZ, R.R., KOK. J.N. 2005. Frequent subtree mining an overview. *Fundamenta Informaticae, Special Issue on Graph and Tree Mining*, vol. 65, no. 1-2, 161-198.

FENG, L., DILLON, T.S., WEIGAND, H., CHANG, E. 2003. An XML-Enabled association rule framework. In *Proceedings of the 14th Database and Expert Systems Applications* (*DEXA 2003*), Prague, Czech Republic, 88-97.

FENG, L., AND DILLON, T.S. 2004. Mining XML-Enabled association rule with templates. In Proceedings of the *3rd International Workshop on Knowledge Discovery in Inductive Databases* (*KDID 2004*), Pisa, Italy, 66-88.

FENG, L., AND DILLON, T.S. 2005. An XML-Enabled data mining query language XML-DMQL (invited paper). *International Journal of Business Intelligence and Data Mining*, vol. 1, no. 1, 22-41.

GHOTING, A., BUEHRER, G., PARTHASARATHY, S., KIM, D., NGUYEN, A., CHEN, Y-K., & DUBEY, P. 2005. Cache-conscious Frequent Pattern Mining on a Modern Processor, In *Proceedings of the 31st International Conference on Very Large Database (VLDB)*, Trondheim, Norway, 577-588.

HAN, J., PEI, J., YIN, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD Conference Management of Data*, Dallas, Texas, USA, 1-12.

JENKINS, B. 1997. Hash Functions. Dr. Dobb's Journal, September 1997.

KUDO, T. 2003. An implementation of FREQT, http://www.chasen.org/~taku/software/freqt/. (*Last accessed 1 Jan 2006*).

KURAMOCHI, AND M., KARYPIS, G. 2004. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions Knowledge and Data Engineering*, vol. 16, no. 9, 1038-1051.

LUK, R.W., LEONG, H., DILLON, T.S., CHAN, A.T., CROFT, W.B., ALLEN, J. 2002. A Survey in Indexing and Searching XML Documents. *Journal of the American Society for Information Science and Technology*, vol. 53, no. 6, 415-438.

NIJSSEN, S., AND KOK, J.N. 2003. Efficient discovery of frequent unordered trees. In *Proceedings of the 1st International Workshop Mining Graphs, Trees, and Sequences (MGTS-2003)*, Dubrovnik, Croatia, 55-64.

PAPAKONSTANTINOU Y., AND VIANU, V. 2000. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'00)*, Dallas, Texas, USA, 35-46.

PARK, J. S., CHEN, M.-S., YU, P.S. 1997. Using a Hash-based method with transaction trimming for mining association rules. *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 5, 813-825.

RUCKERT, U., AND KRAMER, S. 2004. Frequent free tree discovery in graph data. In *Proceedings of the 2004 ACM symposium on Applied computing*, Nicosia, Cyprus, 564-570.

SIDHU, A. S., AND DILLON T. S., CHANG, E., SIDHU, B. S. 2005. Protein ontology: vocabulary for protein data. In *Proceedings of the 3rd IEEE International Conference on Information Technology and Applications* (*ICITA 2005*), Sydney, Australia, 465-469.

SUCIU, D. 2000. Semistructured data and XML. *Information Organization and Databases: Foundations of Data Organization*, K. Tanaka, S. Ghandeharizadeh, and Y. Kambayashi, Eds. Kluwer International Series In Engineering And Computer Science Series, vol. 579. Kluwer Academic Publishers, Norwell, MA, 9-30.

TAN, H., DILLON, T.S., FENG, L., CHANG, E., HADZIC, F. 2005a. X3-Miner: mining patterns from XML database. In *Proceedings of the 6th International Data Mining 2005*, Skiathos, Greece, 287-297.

TAN, H., DILLON, T.S., HADZIC, F., FENG, L., CHANG, E. 2005b. MB3-Miner: mining eMBedded subTREEs using tree model guided candidate generation. In *Proceedings of the 1st International Workshop on Mining Complex Data 2005 in conjunction with ICDM 2005*, Houston, Texas, USA, 103-110.

TAN, H., DILLON, T.S., HADZIC, F., FENG, L., CHANG, E. 2006a. iMB3-Miner: Mining induced/embedded subtrees by constraining the level of embedding. In *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining* (*PAKDD 2006*), Singapore, 450-461.

TAN, H., DILLON, T.S., HADZIC, F., FENG, L., CHANG, E. 2006b. SEQUEST: mining frequent subsequences using DMA Strips. In *Proceedings of Data Mining and Information Engineering '06*, 11-13 July, Prague, Czech Republic, 315-328.

TAN, H. 2008. Tree Model Guided (TMG) enumeration as the basis for mining frequent patterns from XML documents. PhD Thesis, University of Technology Sydney (UTS).

TATIKONDA, S., PARTHASARATHY, S., and KURC, T. 2006. TRIPS and TIDES: new algorithms for tree mining, In *Proceedings of the 15th ACM international Conference on information and Knowledge Management* (*CIKM '06*), Arlington, Virginia, USA, 455-464.

TERMIER, A., ROUSSET, M-C., SEBAG, M. 2002. Treefinder: A first step towards XML data mining. In *Proceedings of the 2nd IEEE International Conference on Data Mining* (*ICDM 2002*), Maebashi City, Japan, 450-458.

WAN, J.W. AND DOBBIE, G. 2003. Extracting association rules from XML documents using XQuery. In *Proceedings of the 5th ACM international Workshop on Web information and Data Managemen (WIDM '03)*. New Orleans, Louisiana, USA, 94-97.

WANG, C., HONG, M., PEI, J., ZHOU, H., WANG, W., SHI, B. 2004. Efficient Pattern-Growth methods for frequent tree pattern mining. In *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining* (*PAKDD 2004*), Sydney, Australia, 441-451.

WANG, K., AND LIU, H. 1998. Discovering typical structures of documents: a road map approach. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia, 146-154.

YAN, X., AND HAN, J. 2002. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2nd IEEE International Conference on Data Mining* (*ICDM 2002*), Maebashi City, Japan, 721-724.

YANG, L.H., LEE, M.L., HSU, W. 2003. Efficient mining of XML query patterns for caching. In *Proceedings of the 29th International Very Large Data Bases* (*VLDB*) *Conference*, Berlin, Germany, 69-80.

XIAO, Y., YAO, J.-F., LI, Z., DUNHAM, M.H. 2003. Efficient data mining for maximal frequent subtrees. In *Proceedings of the 3rd IEEE International Conference on Data Mining* (*ICDM 2003*), Melbourne, Florida, USA, 379-386.

ZHANG, J., LING, T. W., BRUCKNER, R.M., TJOA, A.M., LIU, H. 2004. On efficient and effective association rule mining from XML data. In *Proceedings of the 15th International Conference Database and Expert Systems Applications* (*DEXA 2004*), Zaragoza, Spain, 497-507.

ZHANG, S., ZHANG, J., LIU, H., WANG, W. 2005. XAR-miner: efficient association rules mining for XML data. In *Proceedings of the Fourteenth International World Wide Web Conference* (*Special interest tracks and posters*), Chiba, Japan, 894-895.

ZAKI, M.J. 2003. Fast Vertical Mining Using Diffsets. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, USA, August 2003.

ZAKI, M.J. 2005. Efficiently mining frequent trees in a forest: algorithms and applications. *IEEE Transaction on Knowledge and Data Engineering*, vol. 17, no. 8, 1021-1035.

APPENDIX 1: Implementation issues

Accelerating Object Oriented (OO) approach. OO development approach is one of the common practices today in software development and it has been claimed to result in a more manageable, extensible and easy to understand code. We have developed our algorithms in C++ making use of some of the OO features of the language such as class and function overloading. There are a few important things worth noting when implementing time critical systems using the OO approach. Constructor calls on objects have overheads and one should keep them as low as possible. Inheritance between objects should be avoided whenever possible. We found that performing equivalent computations through primitive objects such as array of integers instead of thick objects such as vector objects or link list objects could improve the performance, especially when we need to construct a heap of objects in memory. Another reason not to use thick objects is that they may implement inheritance. An additional way to avoid expensive constructor calls is by only storing a hyperlink [Wang et al. 2004] or a kind of pointer to the existing object and using that pointer to access information from the same object. Another well known way to increase the performance is by passing an object through function by reference instead of by value. Moreover, when copying a block of memory from one location to another location, performing the operation through the use of a memory block copying routine such as memcpy in C library instead of using the for loop could also be the next step in performance tweaking. Last but not least, one can consider writing inline functions when they are called very frequently. However it is not always a good practice to create all functions in inline mode.

Hash Functions. In the context of mining frequent patterns, one of the most common approaches to do frequency counting is to use a hashtable. When using a hashtable choosing a good hash function is a very important task. Unfortunately, choosing a hash function can be more than a trivial task [Jenkins 1997]. The following are several known hash functions that were compared: Rotation Hash (RH), Additive Hash (AH), Bernstein Hash (BH), Zobrist Hash (ZH), and One-At-A-Time Hash (OH). Detailed descriptions of each hash function and a few others hash functions can be found in Jenkins [1997]. To see the effect of using different hash functions we ran experiments on reduced CSLogs [Zaki 2005] dataset that consists of 32,241 transactions of trees with max. size: 214, max. height:28, and max. fan-out:21.

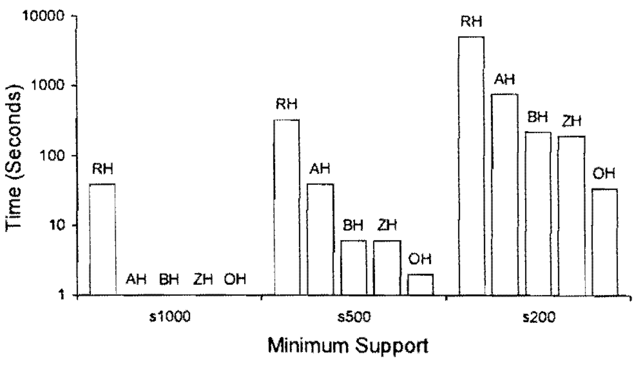


Figure 24: number of collisions plotted over different minimum support

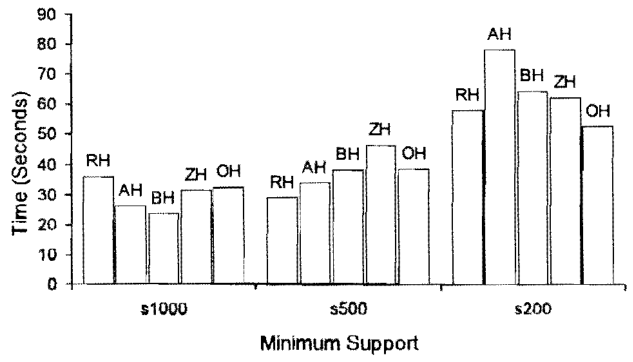


Figure 25: time performance plotted over different minimum support

From Figure 24 and Figure 25 it is not quite obvious which function is truly the winner. When minimum support set to 1000, BH is the fastest. At minimum support 500 and 200, RH and OH are the fastest respectively. In terms of number of collisions produced, RH is consistently the worst for different minimum support and OH seems to be the best of all. One interesting point to note is that at minimum support 500, RH has a faster execution time than RH at minimum support 1000, whereas the other functions seem to have increasing execution time when the minimum support threshold is decreased. There is no direct explanation for this but we can see an indication that certain functions can have varying performances at different minimum support threshold. In general simple functions like AH and RH are not recommended as they do not handle collisions very well [Jenkins 1997]. Our experiment supports this view. AH and RH are the two functions that produce the worst number of collisions (Figure 24). From Figure 24 and Figure 25, we can infer that BH and OH are the two top performers. For all the experiments used in Section 7 we use BH. BH can produce fewer collisions than a hash that gives a more truly random distribution if all 32 bits in the keys are used [Jenkins

[1997]. However, if you do not use all the 32 bits, this function has detected flaws. There is a possibility that if a better hash function is used further optimization can be attained.

Hashing Int vs String In our algorithm, we transform and map the tree structure data into integer indexes as opposed to consuming string labels directly [Tan et al. 2005a, 2005b; Zaki 2005]. Representing label as integer opposed to string label has considerable performance and space advantages. When a hashtable is used for candidate frequency counting, hashing integer over string label can have significant impact on the overall candidates counting performance. In an experiment we discovered how the time taken to hash a string versus and integer can differ by more than 10x when the dataset is large and patterns become relatively long.

Label sensitivity. Let's consider a very large database of integer-labeled trees with large labels set. In this case the labels can be a very big integer value. We performed label sensitivity test and created 4 synthetic datasets by varying the maximum integer label values: 24; 24,000; 24,000,000; 240,000,000. It is important to see that the algorithms can handle databases of small and big integer-labeled trees.

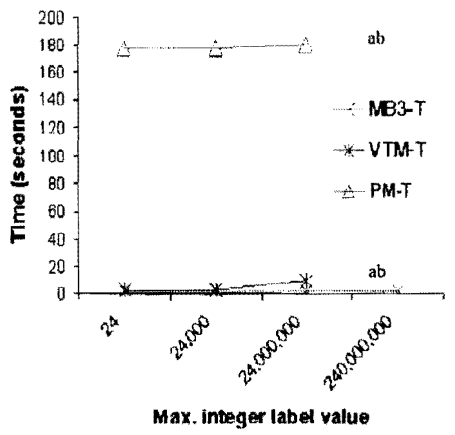


Figure 26: label sensitivity test

As we can see from Figure 26, MB3 can handle both small and big integer-labeled trees very well. The performance of MB3 remains the same for all 4 different datasets. On the contrary, we find that both TreeMiner algorithms VTreeMiner (VTM) and PatternMatcher (PM) suffer performance degradation whenever the maximum value of the integer-labeled trees is increased. Surprisingly, for the last dataset with maximum integer-labeled value equal to 240,000,000 both VTM and PM get aborted. We observe that the implementation of the TreeMiner for generating 1 and 2-subtrees employs a perfect hashing scenario using array objects and using the label as the key for each cell in the array. What essentially happens with this implementation is that it is performance optimized but space inefficient. In the last scenario where the maximum label can go up to 240,000,000, VTM and PM will unnecessarily allocate an array with 240,000,001 cells even when there is only 1 node with label equal to 240,000,000 in the tree database. Using a hashtable as opposed to using an array based implementation would solve this problem.