X3-Miner: Mining Patterns from XML Database

Henry Tan[1], Tharam S. Dillon[1], Feng Ling[2], Elizabeth Chang[3] and Fedja Hadzic[1]

[1] Faculty of Information Technology, University of Technology Sydney
{henryws, tharamd, fhadzic}@it.uts.edu.edu.au

[2] University of Twente, The Netherlands
ling@cs.utwente.nl

[3] Curtin University, Australia
ChangE@cbs.curtin.edu.au

Abstract

An XML enabled framework for representation of association rules in databases was first presented in **[Feng03].** In Frequent Structure Mining (FSM), there are techniques proposed to mine frequent patterns from complex trees and graphs databases. One of the popular approaches is to use *graph matching*. Graph matching algorithms use data structures such as the adjacency matrix **[Inokuchi00]** or adjacency list **[FSG01]**. Another approach represents semi-structured tree-like structures using a string representation, which is more space efficient and relatively easy for manipulation **[Zaki02]**. However, in the XML Era, mining association rules is faced with more challenges due to the inherent flexibilities of XML in both structure and semantics. The primary challenges include 1) a more complicated hierarchical data structure with tags and attributes; 2) an ordered data context; and 3) a much bigger data size. To tackle these challenges, in this paper, we propose an approach – X3-Miner that efficiently extracts patterns from a large XML data set, and overcomes the challenges by: (1) exploring the use of a model validating approach in deducing the number of candidates generated. The basic idea is that by taking into account of the semantics embedded in the tree-like structure in an XML database while generating candidates directly from the XML tree, we can obtain only valid (i.e., possibly existing) candidates out of the XML database; (2) minimising I/O overhead by first trimming the infrequent 1-itemset in the XML database. The XML database is intersected with the frequent 1-itemset resulting in a smaller XML tree that contains only the frequent 1-itemset. The algorithm also progressively trims infrequent k-itemsets that contain infrequent (k-1)-itemsets. (3) extending the notion of *string* representation of a tree structure proposed in **[Zaki02]** to *xstring* for describing an XML document in a flat format without loss of both structure and semantics. Such an extension enables an easier traversal of the tree-structured XML data during our model-validating candidate generation.

Our experiments with both synthetic and real-life data sets demonstrate the effectiveness of the proposed model-validating approach in mining XML data.

Keywords: Data Mining, Algorithm, Association Mining, Semantic Relationships, XML.

1 Introduction

In several years we have seen tremendous works in the area of mining graph, sequence and tree data for patterns of interest **[Agr95, Inokuchi00, FSG01, Washio02, TreeFinder02, Abe02, Zaki02, Inokuchi02a, Inokuchi02b, COFI03, Feng03, XRules03, Feng04, Zhang04]**. While some approaches have focused on mining for patterns in databases containing general graphs **[FSG01, Washio02, Inokuchi00, Inokuchi02a, Inokuchi02b],** the rising of XML data and the need for mining semi-structured data has sparked a lot of interest in finding frequent rooted trees in forests **[Asai01, TreeFinder02, Abe02, Zaki02, Feng03, COFI03, XRules03, Feng04, Zhang04]**. The work described in this paper falls into the latter category: we are mining for frequent substructures in tree-like structure database and in particular data represented in XML format. Our research takes a step on developing an algorithm that is well suited to the characteristics of the domain described in **[Feng03, Feng04]** which is characterized by: 1) Ordered data context 2) More complex hierarchy relations 3) Larger in size.

Modelling objects using graphs allows us to represent arbitrary relations among entities. The standard ways of representing graph data is via an adjacency matrix **[Inokuchi00]** or adjacency list **[FSG01]**. Representing graph data in those forms requires graph traversal that can be very expensive in terms of CPU processing and memory for large datasets. Tree like structure as a more specialized form of graph can be easily converted to a string representation or pattern set representation **[Zaki02, Bayardo98]**. **[Zaki02]** shown that the string representation is relatively easy to manipulate and less expensive in comparison to both adjacency matrix and adjacency list representation. His work concentrated on mining frequent tree structures within a forest. This was one of the first attempts in mining tree-like structures as opposed to mining relational data. The developed algorithm was one of the most efficient current approaches to tree mining. Furthermore it was noted that the algorithm could be extended for the purpose of mining frequent tree structures in XML documents. Recently, **[Zhang04]** has proposed a hybrid approach transforming XML documents into IX-Tree and Multi-DB depending on the size of the XML documents. The reported approach showed that XAR-Miner is more efficient in performing a large number of AR mining tasks from XML documents than the MINE RULE operator introduced in **[Meo96]** and extended in **[Meo98]**.

In this paper we propose a novel approach to efficiently extract frequent patterns from XML database. We deviate developing our algorithm from XQuery-based approaches as done in **[Braga02]** as XQuery-based implementation suffers greatly from a slow performance. The proposed algorithm employs a unique strategy to efficiently generate candidates that improves significantly over the classic apriori based approaches. As has been pointed out in **[Han01]** Apriori based approaches **[DHP97, Inokuchi00, FSG01, Zaki02]** suffer greatly from an inherent problem in generating C_2. In fact a performance gain can be attained by having a smaller C_2 in the early iteration **[DHP97]**. Motivated by such limitation, a novel way of generating C_2 has been carefully redefined by taking advantage of the semantic notion attached to XML format.

In most of the previous works proposed the database provides little clue of how candidates can be generated out of records or transactions. Very often majority of candidates in C_2 have been generated with little knowledge of the actual data model. Hence, many candidates generated are useless, invalid or redundant. It consequently contributes greatly to combinatorial explosion

problem. XML data on the other hand provides more clues to generate C_2 efficiently via its hierarchical-structures and relationships between tags.

We understand that a growth in biological term is somehow controlled by some internal and external factors. We can observe that biological tree grows naturally from a small seed to become a big tree with certain growth pattern. The growth pattern is somehow controlled in such a way that it follows certain model. For example, biological tree members of class A will grow uniquely according to a tree model of class A and we call such tree as an instance of tree from class A. Generalizing the above direction, a candidate generation technique should then be derived accordingly to our XML data. A systematic guided candidate generation according to a generation model is what we should really aim in this research. Able to do so, generation of invalid patterns can be minimized.

Following the above deduction, A substructure of size (k+1) should then be able to be generated from a substructure of size k from any particular XML data following the data model embedded in the XML document. The data model in XML is described by the inherent hierarchical relationship between the elements/attributes contained in it. In other words, the XML data gives a clue on how a (k+1) substructure should be generated from specific k substructure exist in the XML data. Any (k+1) substructure generated violates the embedded data model will result in invalid substructure. In fact, one of our main objectives in generating candidates is to alleviate generating invalid substructures by taking advantage of semantic notion attached to the XML data. Thus, it saves us from extra processing incur from having to generate and eliminate those invalid substructures.

The first attempt to achieve such objective is to transform the XML document into an easy to manipulate string representation. In the proposed algorithms the XML document is converted into a modified version of a scope-list **[Zaki02]**, called *xstring*. Through xstring the order and the semantic of the original XML database preserved. For fast candidate filtering and counting, our approach employs a generalization of the hash based approach proposed by **[DHP97]** for unstructured simple data. As *xstring* not only preserves the order but also relationships between elements in XML document, two strings with the same value but with different relationships between each item in the string would be considered as two unique patterns. Two different structures with the same label, hence, will be hashed into different hash bucket.

As the result, our algorithm can overcome the most expensive candidate generation at C_2 from $O(n^2)$ down to O(n) complexity, where n is the size of frequent 1-itemset, |L1|. Furthermore, our approach generates only valid candidates. To our definition all valid candidates should at least have a frequency of 1 in the database. Hence, generation of invalid and redundant candidates is eliminated. The resulting algorithm lightens candidate reduction process.

1.1 Problem Statements

A *tree* is an acyclic connected graph. XML data can be easily represented in a tree-like structure. A rooted tree is a tree in which one of the vertices is distinguished from others, and called the *root*. All well-formed tree has only one unique root node. XML data is a rooted well-formed tree **[Feng03]**. We refer to a vertex of a rooted tree as a node of the tree. In case of XML data, node refers to tag. An ordered tree is a rooted tree in which the order of the children of each node is important. Hence,

if we have k nodes as children of say node A, a node A at the left most position would be at position 0 and the nodes at its right will have incrementing position number up to the (k-1)th child. A labelled tree is a tree where each node of the tree is associated with a label. There can be two or more nodes with the same label. XML document possesses a hierarchical document structure, where an XML element may contain further embedded elements, and can be attached with a number of attributes. It is therefore frequently modelled using a labelled ordered tree. In this paper, we consider XML data which basically ordered, labelled, and rooted trees.

Ancestor and Descendants. Consider a node x in a rooted tree T with root R. Any nodes y on the unique path from R to x is called x's ancestor. We denote node $y \leq_{\prime} x$. Oppositely, if y is x's ancestor, then x is descendant of y. On the other hand, if $y \leq_{\prime} x$, then y is called the parent of x and x is the child of y. We called nodes x and y are *siblings* if they share the same parent.

Subtrees and embedded subtrees. Any node in tree, T, is identified by its position and its label. In case of XML, the label could be of a simple type or complex type. A simple type label would be represented as just the XML element name. A complex type label in XML would be represented as a combination of of element name, attribute pairs and values. Each edge, $e = (n_x,n_y) \in E$, is an ordered pair of nodes, where n_x is the parent of n_y. A tree $S = (N_s, E_s)$ is an embedded subtree of $T = (N_t, E_t)$, denoted as $S \therefore T$, provided: $N_s \subseteq N_t$, and $e = (n_x, n_y) \in E_s$ if and only if $n_x \leq_{\prime} n_y$ (n_x is the ancestor of n_y in T). This is equivalent by saying an edge appears in S if and only if the two vertices are on the same path from the root to the node in T. $S \therefore T$ can also be said that T contains S. Embedded subtrees are a generalization of induced subtrees **[Inokuchi00]**. They allow not only direct parent-child edges, but also ancestor-descendant edges. Via embedded subtrees as described in **[Zaki02]**, hidden pattern deep within large trees can be detected. The following figure illustrate the difference between induce subtrees and the embedded subtrees.

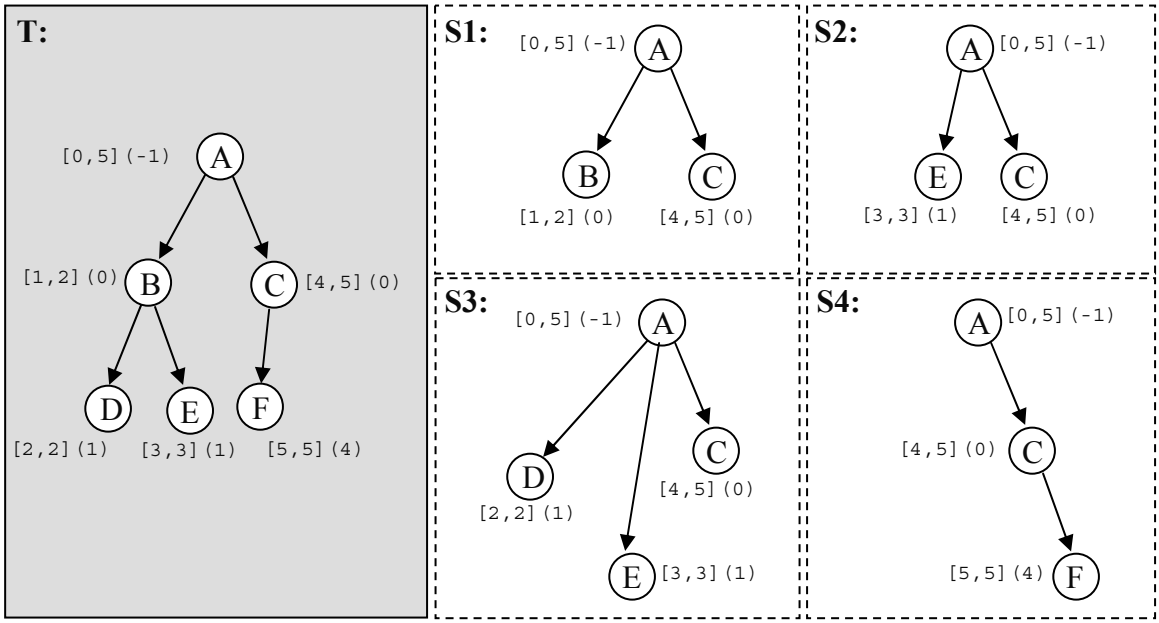


Figure 1 An illustration of induced subtree S1, S4 and embedded subtree S2, S3 of T

Node position, scope and direct parent pointer. We denote a tree as T as collection of nodes (N) and edges (E). The size of T, denoted as |T|, is the number of nodes in T. Each node n ∈ N will contain a triplet with notation: [**pos**, **scope$_{max}$**] (**dpp**), where pos is node position, scope$_{max}$ is the right-most descendant's node position, and direct parent pointer which will be described in more details in the following passage.

Each ***node position*** has an integer number, i, start from 0 following its position in depth-first (or pre-order) traversal of the tree as can be seen from figure 1. We also called that the node position as the coordinate of the node within the database. It tells us in spatial term where it is uniquely located in the database. Our candidate generation utilizes this spatial information to uniquely identify each node in the tree.

The **scope** is defined as the last child's or last descendant's node position. If T(n_o) refers to a subtree rooted at node n_o, and n_n be the right-most leaf node in T(n_o). The scope of node n_x is given as the interval [o, n]. Hence, a node at position i with k-nodes associated as its children then the scope of that node will be an integer pair start from n to (n+k), denoted [n, n+k]. We say a node position at position n has a scope from n, (n+1), …, (n+k). Also, all nodes within the scope of node n would be its children or descendants, except itself. For each node, therefore, a scope pair is defined such as {(n,m) | n ≤ m, n, m is positive integer}. Thus, the scope information consists of pair of integers, (n, m), where m=n+k for node having k-descendant. The first integer, n, describes a position of a node in the string and the second integer, m, describes the position of its right most descendant according to pre-order traversal. The scope pair is very powerful in encoding node positions, parent-child, and ancestors-descendants relationships between nodes. It tells you for each node if whether it is a leaf node or not. Plus, for each parent nodes the number of descendent nodes associated with it can be obtained easily.

The example below will illustrate how the scope information works.

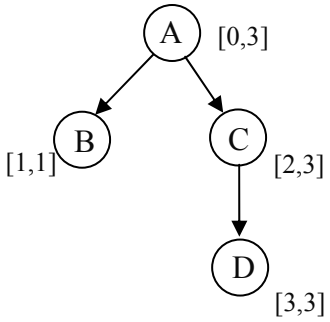


Figure 2 Scope Pair Illustration

In the figure above, A is the first node in the string. As the position of node is encoded in the first integer of the scope pair, with a zero-based indexing, node A's position is encoded at position 0 followed by nodes B's, C's and D's position respectively at position 1, 2 and 3. The scope of node A, on the other hand, tells us that node A has descendants and the right-most descendant's position is at position 3 in the string. The other way to say this is that node A's scope is from 0 to 3, i.e. all nodes within A's scope are its descendants, except itself. The number of A's descendents can be calculated by subtracting the scope by the node position, i.e. 3-0=3.

On the contrary, node B doesn't have descendants. It can be seen from its scope that its position and its $scope_{max}$ are equal. This tells us that node B is a leaf node. The same reasoning applies to node D. The example above also tells us that D is descendant of C, as D's position, 3, is greater than C position but less than C's scope or within C's scope. Similarly, we can easily find out that B, C, and D are also the descendants of node A with the same reasoning.

From the example above, it has been shown that the introduction of scope pair into the string representation is so powerful in preserving the semantic notion that is in XML documents. Through this scope pairs, the hierarchical tree structure in XML documents can be encoded in an efficient manner. Plus, through the use of array indexing, quick data retrieval and storing can be realized. The scope pair, on the contrary, excellently helps us representing XML documents in a space efficient format as it compresses the size of XML documents in some ways. First, you don't need to encode a '<' and '>' for each node. Secondly, you don't need to store an opening and closing tag pairs. The scope pair will take care of this. Hence, all of these features of scope pair enable us to represent XML documents in a flat string representation that has a 1-to-1 mapping to its corresponding XML document.

A ***direct parent pointer*** is also stored in each of node in T so that it can directly reference its parent node. Via this direct pointer an efficient candidate generation process can then be achieved. It eliminates the needs to search for direct parent of each node in T. We trade a space for performance by adding a direct parent pointer to each of node in T. The direct parent pointer is generated in transformation phase of our algorithm, i.e. transforming XML data to our string representation *xstring* in pre-order traversal way.

Back to figure 1, having the triplet of *node position*, $scope_{max}$ and *direct parent pointer* described we can now interpret the triplet notation attached to each node. For example node A has a triplet [0,5](-1), i.e. it is at position 0, with $scope_{max} = 5$ and have no parent, -1. Node D has a triplet [2, 2](1), i.e. it is at position 2, with $scope_{max} = 2$ and its parent is located at position 1. B is D's parent; B is at position 1.

XML document mining problem.

Knowledge discovery is the process of detecting new and useful patterns from data. Data mining is a step in knowledge discovery concerns with applying programs that are capable of finding patterns from data and that can learn and generalize from the presented information **[Han01]**. Thus, the notion of semantic is important. However, most of the works done in the association mining area have been focused on improving the performance but less has been emphasized on the semantic part **[Zaki98, Han04]**. This research tries to take these two areas of development to receive an equal attention. The algorithm proposed would be consequently aimed at constructing an efficient algorithm that intelligently discovers more useful semantic concepts.

The problem of mining association rules can be decomposed into the following steps:

1. Discover the large itemsets that have support above a predetermined minimum support s
2. Use the large itemsets obtained from step 1 to generate association rules

The challenge of association mining is in fact to have efficient and scalable process in discovering large itemsets **[DHP97, Han01]**. After all large itemsets discovered the generation of association rules can be derived more straightforwardly. The paper concentrate on developing step one of the above steps and apply the technique to XML data (semi-structured data) rather than relational data (structured data). Hence, generalization and extension of previous approaches done in the literature is necessary.

As has been introduced in **[Feng03, Feng04]** the problem of finding frequent patterns from XML data deals with tree-like structure rather than atomic item as in relational data. It extends the notion of associated items to XML fragments to present associations among trees rather than simple-structured items of atomic values. The adaptation of association mining to the XML document results in a more flexible and powerful representation of both simple and complex structured association relationships inherent in XML data. Hence, in this paper, a tree like structure and a collection of tree-like structures with size k will be called a k tree-structured-item (k-tsi) and a k tree-structured-itemset (k-tsi set).

Let D denote a database of XML document which literally can be perceived as database of trees. Let $S \therefore T$ for some $T \in D$. Each occurrence of S can be identified by its *matched string*, which is given as the unique set of sequence of string (in T) for string in S. The string is constructed from the combination of XML elements, attributes and values. More formally, let $\{t_1, t_2, \ldots, t_n\}$ be the XML elements in T, with $|T| = n$, and let $\{s_1, s_2, \ldots, s_m\}$ be the XML elements in S, with $|S| = m$. Then S has a match string $\{ti_1, ti_2, \ldots, ti_m\}$, if and only if: 1) $str(s_k) = str(t_{ik})$ for all $k = 1, \ldots, m$, and 2) edge $e(s_j, s_k)$ in S iff ti_j, is ancestor of ti_k in T. The first condition specifies that all nodes in S have a match in T, while the second condition indicates that the tree structure or topology of the matching nodes in T is the same as in S. Let $\delta_T(S)$ defined as the number of occurrences of the subtree $S \therefore T$. [Zaki02] defines two type of supports, weighted and non-weighted support. Let $\sigma(S)$ be the *support* of subtree $S \therefore T$. Our definition of support of $S \therefore T$ is defined as $\sigma(S) = \Sigma_{T \in D} \delta_T(S)$, i.e. total number of occurrences of S over all trees in T. This definition of support is equivalent to the weighted support definition. A subtree S is frequent if its support is more than or equal to a user-specified minimum support (σ_{min}). Our goal in mining XML document is to enumerate all frequent tsis in D given a user specified σ_{min} value.

D:

```
<ORDER>
    <PERSON Profession="teacher" >
        <Name>Martin Louis</Name>
        <Age>Young</Age>
        <Gender>Male</Gender>
        <Address>Wattle Bank</Address>
    </PERSON>
    <ITEM>
        <CD>
            <Title>Pop Music</Title>
            <Title>StarWar Game</Title>
        </CD>
        <BOOK>
            <Title>StarWar I</Title>
            <Title>StarWar II</Title>
            <Title>Lost Space</Title>
        </BOOK>
    </ITEM>
```

```
<PERSON Profession="student">
        <Name>George Sorensen</Name>
        <Age>Young</Age>
        <Gender>Male</Gender>
        <Address>York City</Address>
</PERSON>
<ITEM>
        <CD>
                <Title>Jazz Music</Title>
                <Title>StarWar Game</Title>
        </CD>
        <BOOK>
                <Title>StarWar I</Title>
                <Title>StarWar II</Title>
                <Title>Lost Space</Title>
        </BOOK>
</ITEM>
</ORDER>
```

Figure 3 An XML document example [Adapted from Feng03]

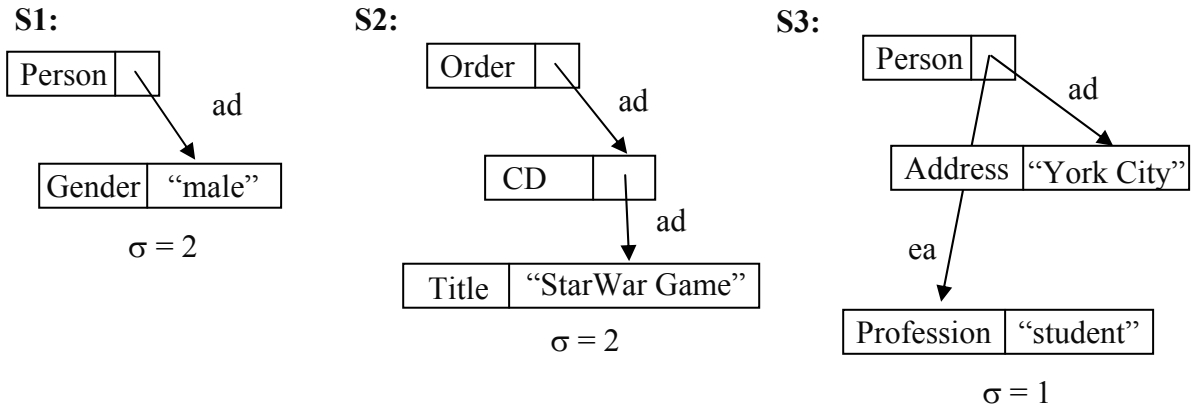


Figure 4 Example subtrees of XML document in Figure 3.

Consider Figure 4, which shows an example subtrees S1, S2, and S3 of the XML document , D. S1 is a subtree of D, it has a support, σ, of 2. We can see from figure 2 that there are two occurrences of subtree that with matched label and topology in D. S2 is also a subtree of D with support, σ, of 2. We can see that there are two CDs with title "StarWar Game" have been Ordered. In the last two examples, S1 and S2 exhibit the possible subtree formed as a combination of elements/values relationship. However in the last subtree, S3, a possibility that the subtree in XML can be formed as a combination of elements/attributes/value relationship is demonstrated. ea and ad stands for element-attributes and ancestor-descendant relationship respectively. In S3, Person with Profession: "student" is connected by an edge of ea constraint **[Feng03]** and Person at the Address: "York City" is connected by an edge of ad constraint. We are aiming at extracting such patterns in our mining algorithm. S3 has σ = 1.

1.2 Related Works

XML document has increasingly gained popularity and adopted as a platform independent data exchange medium. On the other hand, association mining has been so successful in discovering useful associations between data **[Agr93, Agr94, Agr96, DHP97, Bayardo98, Zaki02, Feng03, and Feng04]**. To date, most of the works done in association mining **[Agr93, Agr94, Agr96,**

CHARM02, COFI03, Brin97, Han00, Pan03, Pei00, Toivonen96, DHP97, Pasquier99, Robert98, Hidber98, Zaki02, Zaki03, and TOPK02] were tailored for structured data but only few for semi-structured data **[Feng03, Feng04, Asai01, XRules03, Wang 97, Wang98, Wang00, Cong02]**; especially where the order is important and schema is not fixed. Also, much research has been emphasized on performance but less has been emphasized on semantic.

[Wang97, Wang98, Wang00] works was to discover similar structures among a collection of semi-structured objects. A modified version of frequent set mining algorithm for finding structural associations inherent in semi-structured objects was done in **[Cong02]**. Recently, **[Feng03]** has initiated an XML-enabled association rule framework. It extends the notion of associated items to XML fragments to present associations among trees rather than simple-structured items of atomic values. The adaptation of association mining to the XML document as is shown in **[Feng03]** results in a more flexible and powerful representation of both simple and complex structured association relationships inherent in XML data. **[Feng04]** then continue their work with a template model for mining XML-enabled association rules, which can deal with associations among both contents and structures, with possibly a set of constraints related to context positions including hierarchical levels. The work reported using template-guided mining of association rules from large XML data.

In the next passage, we will briefly describe some algorithms developed for data mining within the context of the research of this paper.

The apriori algorithm firstly introduced in **[Agr96]** works very efficient if the data to be analysed is sparse and there are few frequent itemsets containing large numbers of items. However, Apriori is less efficient when data is dense **[Bayardo97]**. For dense data the number of frequent itemsets can greatly outnumber the number of transactions, requiring a lot of computation for the management of the candidate frequent itemsets and discovered frequent itemsets.

In Apriori, the 2-itemsets candidate generation, C_2, is done by joining (operator *) frequent 1-itemsets, L_1, with itself, $\mathbf{L_1 * L_1}$. It means, if the order is important and $|L_1|$ (read: size of L_1) equals to n, apriori will generate n^2 candidates. For example, with $|L_1|$ equal to 3, apriori based approach will easily generate $3^2 = 9$ candidates for 2-itemsets. Hence, the complexity of generating 2-itemsets in Apriori turns out to be in $O(n^2)$ complexity class. Secondly, Apriori needs to scan the full database, compare and count the frequency of n^2 candidates generated with the database. In a worst case scenario, where the number of existing 2-itemsets pattern in the database is minimum, apriori generates the maximum combination of 2-itemsets. Therefore, there are n, where n = (maximum-minimum) number of 2-itemsets candidates generated wastefully which add significantly to I/O overhead. Let's take an example of $|L_1| = 10^4$, when order is important, apriori will generate 10^8 2-itemsets candidates. Assuming there are 10^2 existing 2-itemsets in the database, there are $10^8 - 10^2 = 99{,}999{,}900$ number of candidates generated that add up to huge I/O overhead wastefully as a result of comparison cost. It is costly to handle 2-itemsets candidate generation in a huge number of candidate sets in Apriori-like algorithms **[Han00, Bayardo97]**. One approach to reducing these overheads is to reduce the number of frequent itemsets that must be processed. Frequent closed itemsets approaches record only frequent itemsets that are closed **[Pasquier99, Pei00]**.

On the contrary, Zaki formulated the problem of mining (embedded) subtrees in a forest of rooted, labelled, and ordered trees. Zaki presented TreeMiner **[Zaki02]**, an algorithm to discover all

frequent sub-trees in a forest using a new data structure called scope-list. For efficient subtree counting and manipulation TreeMiner adopted a string representation of a tree. He developed a framework for non-redundant candidate subtree generation. He also described a method to compute the frequency of candidate trees by joining the scope-lists of its subtrees in a forest. However, as the matter of fact that TreeMiner is also a variant of Apriori, the complexity of TreeMiner for 2-Itemsets turns out to be in $O(n^2)$ complexity class as well. So for, 2-itemsets candidate generation it has the same computational cost problem.

In Direct Hashing and Pruning (DHP) **[DHP97]**, the 2-itemset candidate reduction is performed to overcome the performance bottleneck in Apriori. Generation of smaller sets enable us to effectively trim the transaction database size at a much earlier stage of iterations by using direct hashing approach, thereby reducing the computational cost for later iterations significantly. The DHP algorithm reduces the amount of disk I/O required. DHP algorithm has three main features: (1) Efficient generation of 2-itemsets, (2) Effective reduction on transaction database size, and (3) The option of reducing the number of database scans required. DHP is very efficient for generation of large candidate sets, in particular for large 2-itemsets, where the number of candidate sets generated by DHP is in orders of magnitude smaller than that generated by previous methods, thus greatly improving the performance bottleneck of the whole process.

On the other hand, the proposed XML-enabled association rule framework **[Feng03]** is flexible and powerful enough to represent: (1) simple and complex structured associations and (2) An ordered and labelled data context. The notion of associated item is extended to an XML tree fragment (tree-structured item) and associations are built among trees rather than simple-structured items of atomic values. A tree-structured item consists of five components: (1) A set of nodes, representing XML elements or attributes. (2) A set of directed edges representing ancestor-descendant or element-attribute relationships between the nodes. (3) A set of labels, denoting different types of relationships between nodes. (4) A set of constraints defined over the nodes and edges. (5) A unique root node.

1.3 Organization of the Paper

We solved the problem of finding all frequent substructures in 2 main phases: i) Transformation phase, ii) Probing phase, and iii) Candidate Generation phase. Section 2 gives the problem decomposition. Section 3 examines our candidate generation phase in detail and presents our algorithm. We empirically evaluate the performance of these algorithms and study their scale-up properties in section 4. In the last section we then conclude the paper with a summary and future works.

2 Our Approach

2.1 Transformation Phase

Before we do the candidate generation, a data structure to represent the XML data in space efficient

way and easy to manipulate need to be constructed. The constructed data structure will also need to handle the semantic notion inherent in XML data. String representation is one of the space efficient ways in representing XML tree. Our approach transforms the XML data into a string like presentation called *xstring*. Previously, **[Zaki02]** introduced a scope-list to represent tree-like structure. His approach uses the scope notion. We are extending the scope-list model that suits our candidate generation method. Our formulation of *xstring* aligns with the formulation of the XML-enabled framework proposed in **[Feng03]**.

Our string representation differ with TreeMiner in two ways, firstly we incorporate a direct parent pointer into our string representation in addition to the scope information. As a result, less computational cost in searching for direct parent in generating candidates dynamically. Secondly, we utilize the string to generate candidate differently than most of the apriori through join approach. Our approach generates fewer candidates by eliminating generation of invalid candidates in each iteration.

The *xstring* can be constructed directly from XML document by parsing the XML Tree in pre-order manner. The position, scope information and the direct parent pointer is then constructed for each of the element in XML document. Hence, for each of the element read from the XML document the triplet, as introduced earlier in section 1.1, [pos, $scope_{max}$](dpp), is calculated. For instance the following fragment example of the XML document as shown in Figure 3 information can be converted to an *xstring* as follows:

```
<ORDER>
      <PERSON Profession="teacher" >
            <Name>Martin Louis</Name>
            <Age>Young</Age>
            <Gender>Male</Gender>
            <Address>Wattle Bank</Address>
      </PERSON>
      <ITEM>
            <CD>
                  <Title>Pop Music</Title>
                  <Title>StarWar Game</Title>
            </CD>
            <BOOK>
                  <Title>StarWar I</Title>
                  <Title>StarWar II</Title>
                  <Title>Lost Space</Title>
            </BOOK>
      </ITEM>
</ORDER>
```

pos	$scope_{max}$	dpp	Element-name:"element-value";{Attribute-name:"attribute-value"}
0	13	-1	ORDER
1	5	0	PERSON;{Profession="teacher"}
2	2	1	Name:"Martin Louis"
3	3	1	Age:"Young"
4	4	1	Gender:"Male"
5	5	1	Address="Wattle Bank"
6	13	0	ITEM
7	9	6	CD
8	8	7	Title:"Pop Music"
9	9	7	Title:"StarWar Game"
10	13	6	BOOK
11	11	10	Title:"StarWar I"
12	12	10	Title:"StarWar II"
13	13	10	Title:"Lost Space"

Figure 5: xstring Transformation Example

2.2 Probing Phase

After the transformation phase has been completed, the next phase is to generate 1-tsis. In apriori generating 1-tsis is known as generating C_1, i.e. set of candidates with size of 1. We call this approach differently. In our approach we are not only generating 1-tsis and count its frequency but also probing the coordinates of each unique element contained in the actual XML database. As has been described earlier we define the pre-order traversal position of the elements in XML tree as their spatial information or coordinate.

We use a hash-based approach for frequency counting. Our approach extends **[DHP97]** hash-based approach to work with the semi-structured data type like XML. When we do frequency counting, two or more of the same patterns encountered with different coordinates will be hashed to the same bucket. At the same time we also keep track of its coordinates. As we represent our xml tree with string, we use the string as the key to the hash bucket. The trade-off of using string as the key to the hash bucket is that the longer the string the more costly the hash key calculation is.

This probing phase memorizes coordinates of each unique element located in the XML database. One-to-many relationships occur between any particular patterns and their coordinates. The ultimate reason of having to memorize each of unique coordinates of elements contained in the XML database is to speed up the Candidate Generation process. Our Candidate Generation method utilizes the tree semantic model inherent in the XML database which is represented in spatial form into our xstring. Thus, having the spatial representation constructed in systematic way optimizes our Candidate Generation method. In this research, we show that the optimization can be achieved via generating candidates through the coordinates.

Having to probe coordinates of patterns with increasing size from the database is an expensive process. Fortunately, we only need to do the coordinates probing once. Although, we still require to construct the coordinates when generating C_2, …, C_k, however, the way they are constructed is different from C_1 coordinates construction (probing). In fact, they require less computational cost. The main reason why they are less expensive in computation cost is due to the inherent semantic

embedded in our *xstring* representation. Thus, having represented our data in *xstring* format enables us to do the coordinates construction for C_2, …, C_k process more efficiently through level-wise coordinates expansion. Thus, the term coordinates probing in our case is really mean for coordinates construction when constructing C_1. More discussion will be provided in the next section, 2.3.

2.3 Candidate Generation Phase

As we all understand that with Apriori-based approach, the candidate generation strategy produces candidates that logically and physically never exist in the database. Their approach let the algorithm generates 2-itemset candidates through join operations. Most of the constructed 2-itemset candidates will then be discarded at the next stage as the algorithm locate that they have no existence in the database. Our expansion strategy eliminates the need to generate redundant, meaningless, and inexistence candidates. It takes advantage of the coordinates introduced in the *xstring* representation. For each of the tsi constructed from the database the coordinate introduced conceptually adds visioning ability. For each *tsi* in constructed it knows other *tsis* within its vicinity. With this visioning ability the expansion can be done in a systematic and natural way. It grows the tsis according to the XML tree model.

In addition, our approach also considers I/O optimization by minimizing the number of scan of the database. As in DHP, we generalize the hashing approach to suit our solution for applying association mining to XML database. One of our novel approach in attempt to bring the complexity of C_2 generation to O(n), we do not use $\mathbf{L_1} * \mathbf{L_1}$. Faster 2-itemset generation with reduced I/O overhead can then easily be achieved. Throughout the candidate generation process, the algorithm trims the string that contain infrequent substring. It adheres to the downward closure lemma that if a pattern is said to be frequent then none of its subpattern is infrequent.

Our approach develops fast string manipulation to encode 2-tree-structured itemset candidates directly from the *xstring*. It applies our tree expansion strategy adding one node at a time to k-tree-structured itemset candidates and consequently constructs (k+1)-tree structured itemset candidates. Our expansion strategy is unique and efficient in generating candidates. We grow frequent tree-structured itemsets efficiently and effectively from 1-tsi to k-tsi , where k=2, …, n and n is the longest possible pattern that can be constructed. The detailed of our algorithm will be discussed in the next section.

3 X3-Miner Algorithm

In this section, we present the pseudo-code of our algorithm in details. We present our C_2 generation approach and its complexity calculation. Continued with our C_k generation approach and followed by updating backtrack approach. Finally we present our k-trimming approach that validates each k-tsi against downward closure lemma.

```
/* C1 Generation pseudo-code */
D = Transform_XMLtoXString(xml-filename);
min_support = s;
k = 1;

for each tag x in D
{
        1-xstring   = xstring(x);
        x-coordinate = getCoordinate(x);
        string-key   = generateStringKey(x);
        hitem        = hashitem(1-xstring, string-key, x-coordinate);
        insert(C1, hitem);
}


/* L1 Generation */
for each hashitem hitem in C1
{
  if(count(hitem) >= min_support)
    insert(L1, hitem)
  else
    trimming(D, hitem) /* Trimming: D - h */
}

k++;

/* Ck Generation */
while(Lk-1.size() > 0)
{
  for each hashitem hitem in Lk-1
  {
        for each coordinate c in hitem
        {
              base-xstring          = getBase(hitem, c);
              string-key            = getStringKey(hitem)
              generateCandidates(base-string, string-key, Ck, Lk-1, D);
        }
  }

  /* Generate k-Frequent Itemsets */
  for each hashitem hitem in Ck
  {
        if(count(hitem) >= min_support)
          insert(Lk, hitem);
  }

  k++;
}
```

Figure 6. Pseudo-code of X3-Miner Frequent Sets Generation

```
/* generateCandidates function */

generateCandidates(base-xstring, key-string, Ck, Lk-1, D)
{
  base-coordinates = getCoordinates(base-xstring);
  coord-size       = length(base-xstring);
  parent-pointer   = coord-size – 1;
  head             = getFirstCoordinate(base-xstring);
  tail             = getLastCoordinate(base-xstring);

  slot = tail;
  next-child-coordinate = (tail + 1);
  while(slot >= head)
  {
        end-child-coordinate = getScopeMax(D, slot);

        for(j = next-child-coordinate; j <= end-child-coordinate; j++)
        {
          newnode = getNode(D, j);
          if (isFrequent(newnode))
          {
                prefix-string   = append(key-string, backtracks-string);
                newnode-string  = generateStringKey(newnode);

                // Check if (k-1)substring including newnode-string is frequent
                if(IsFrequent(prefix-string, newnode-string, coord-size, Lk-1))
                {
                  k-xstring = append(base-xstring, node, parent-pointer);
                  newcoord  = append(base-coordinates, getCoordinate(node));
                  new-key-string = append(prefix-string, newnode-string);

                  hitem = hashitem(k-xstring, new-key-string, newcoord);
                  Insert(Ck, hitem);
                }
      }
    }

    next-child-coordinate = end-child-coordinate + 1;

        slot = findNextParent(slot);
        updateBacktracks(backtracks-string, parent-pointer);
  }
}
```

Figure 7. Pseudo-code for Candidate Generation

3.1 C2 Generation

Our C2 generation is straight-forward and efficient and fall within O(n) complexity class where n = $|L_1|$ [read: size of frequent tsis-1]. The generation is done by appending each descendant of frequent tsi in L_1. For example the generation of 2-tsis from a node labelled A as shown in the figure below produces 2-tsis: AC, AD, AE, and AB.

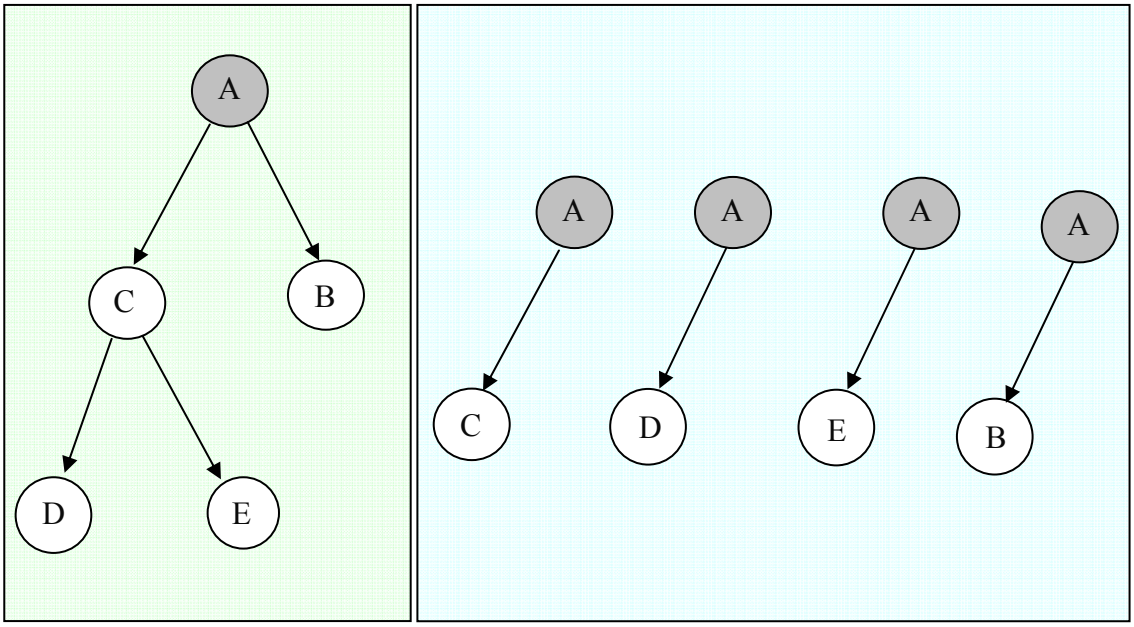


Figure 8. C2 Generation Illustration

3.2 C_k Generation

As our aim is to be able to generate only valid candidates, C_k generation is done through coordinates of nodes constructed in **transformation phase**. To generate (k+1)-tsi from k-tsi our approach utilize the coordinates and the tree-semantic model. Start from the last node in the k-tsi we climb the k-tsi up until it reaches its root. For example from Figure 8 above, say we want to expand 2-tsi AD. we start from D, expanding D with its descendants until no more descendants found. Then, backtrack to C, as C is D's direct parent. Repeat the expansion process, i.e. expanding node C with its descendants, E. We are not expanding node C with its descendant D as we always expand with all C's descendants with the pre-order traversal position greater than C's child (which is D) $scope_{max}$. $Scope_{max}$ has been defined as the position of the right-most descendants of any particular node. We continue the process by backtracking one more time until it reaches the root of the expanded tsi. In our case it is node A. Continue by expanding node A with its descendants that its pre-order traversal position greater than C's $scope_{max}$ as we come from C before backtrack to A. All the 3-tsis generated from 2-tsi AD are shown in the Figure 9 below.

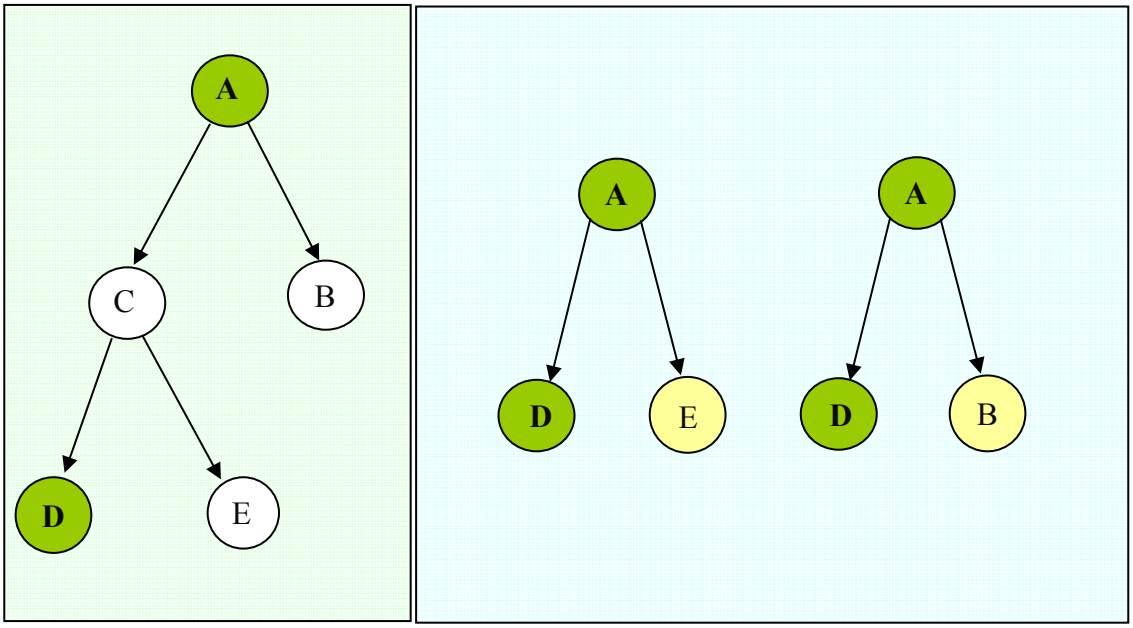


Figure 9. 3-tsis Generated from 2-tsi AD

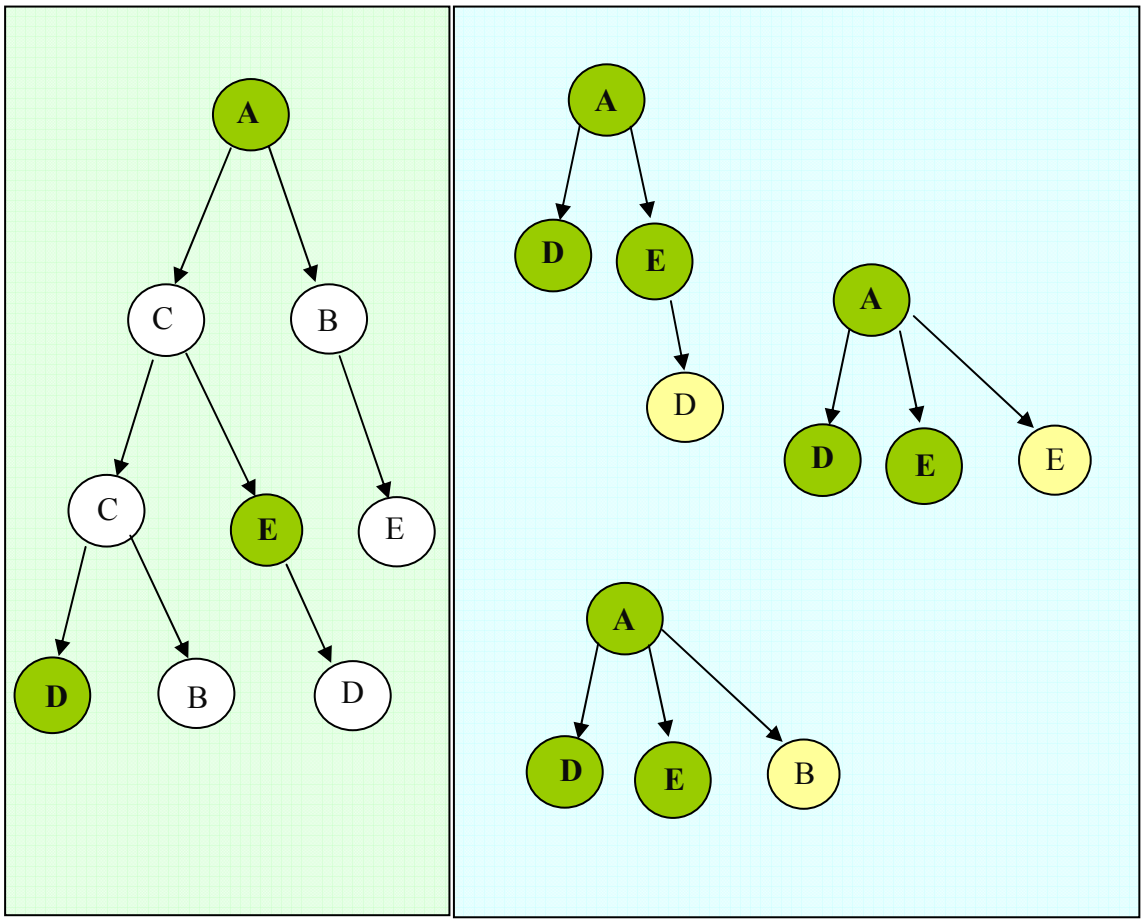


Figure 10. 4-tsis Generated from 3-tsi AD/E

In the Figure 10 above, 3-tsi AD/E as the base-string expanded into 3 4-tsis AD/ED, AD/E/E, and AD/E/B with the same procedure as we generate 3-tsis in Figure 9 earlier. The following explanation applies to Figure 10. One important rule of our candidate generation is that we always start from the last node in the string representation following pre-order traversal, i.e. E. Expand node E with its descendant. As node D is the only descendant of E then we have AD/ED for E expansion. We then backtrack to E's direct parent, which is node C. However no expansion can be made with C as it doesn't have descendants with position greater than the previous child, E's $scope_{max}$. Continue backtrack if it is not the root of the tsi to be expanded. Continue, we expand A with B and E as B and E satisfy the condition, i.e. they are A's descendants with position greater than the $scope_{max}$ of the previous direct descendant or child of A, which is node C. Thus, we finished the candidate generation for the base-string AD/E.

The following is the C_k generation rule:

D := ***XMLDB***
i := last-node-coordinate of ***k-tsi-base-string*** in D
end := root-node-coordinate of ***k-tsi-base-string*** in D

1. ***i*** = ***last-node-coordinate*** in the ***k-tsi-base-string***.
2. ***Expand*** with its descendants. Calculate the $\mathbf{scope_{max}}$ of ***node*(i)** in ***D***.
3. ***Backtrack,*** update ***i*** with its direct parent position, and ***Expand*** with its descendants, ***n***, such that $\{n > i \textbf{ AND } n > scope_{max}(node(i)) \mid n, i < size(D)\}$
4. Repeat step 2, until (***i*** < **end)**.
5. Finish C_{k+1} generation

3.3 Updating Backtracks

This section discusses the updateBacktracks(back-track-string, parent-pointer) method whose implementation details were omitted in the pseudo code from Figure 6 for clarity purposes.
We make use of a direct parent pointer (dpp) in our xstring representation of candidate sub-patterns in order to easily traverse the tree and preserve the structure. Furthermore we use the string representation (SR) of the sub-pattern to easier distinguish different structures. The SR contains string representation of nodes (and values) and back slashes ("/") to indicate when we backtrack on a previous node.
The purpose of this method is to update the parent pointer for the new node added (to form k+1 sub-pattern) and to determine the number of slashes that the SR of the new sub-pattern will have before new node is added. Let: 'D' be the xstring of the original database, 'SD' the xstring of the sub-pattern to be expanded, 'BSR' the SR of the sub-pattern to be expanded, 'NSR' the SR of the new candidate sub-pattern generated and 'n' be the last node (tag) in the SD. First all the frequent descendants (on right) of n are added to SD with the dpp set to n and their SR is appended to the BSR to form NSR for the new candidate sub-pattern. After the descendents of n have been processed we backtrack in the D and add one backslash to the BSR. The same process as above is repeated except that the dpp of the new nodes to be added (later referred to as current dpp) now points to the dpp of the n in the SD. Every time we backtrack in D we check whether the node we are currently at corresponds to the current dpp in SD (we call this a 'hit'). If it does we process the descendents of that node in same way as above and then add a "/" to the BSR and set the current

dpp to be the dpp of the node at current dpp's position in SD. So every time it is a hit we update BSR and current dpp and when it is not a hit it stays the same and descendents are added of the current node we are backtracked to in D. The process repeats until we finish processing the right descendents of the root in SD.
To illustrate the process consider the C_3 generation in Figure2 below. We are starting BSR being "A D" and SD being |A|D|. The current dpp points to |D| and so if there were any descendents they would have |D| as their dpp and they would be appended to the BSR. As there is none we backtrack and so the current dpp now points to |A| and the BSR is now "A D /". The new sub-pattern is created with string representation "A D / E" and xstring |A|D|E| where E's dpp points to |A|. Similarly, new sub-pattern with string representation "A D / B" and xstring |A|D|B| where B's dpp points to |A|. As we have already hit the root in SD the process terminates.
The pseudo code of the process is shown in Figure 7. The same notion is used as above. Note that this pseudo code only indicates the updating of direct parent pointers and the number of backtracks for the string representation as the setting is already previously described in pseudo code from Figure 7.

```
/* pseudo code for updating backtracks and direct parent pointer */
n           = D[n]->GetDirectParentPointer();
current-dpp = SD[previous-dpp]->GetDirectParentPointer();
short hit;
if(current-dpp <= -1)
            hit = 1;
else
            hit = SD[current-dpp]->Equal(D[n]);

if (hit)
{
            previous-dpp = SD[previous-dpp]->GetDirectParentPointer();
            number-of-backtracks++;
            for(int i=0; i < number-of-backtracks; i++)
            {
                        backtracks-string += "/ ";
            }
            previous-backtracks-string = backtracks-string;
}
else
            backtracks-string = previous-backtracks_string + "/ ";
```

Figure 11. Update Backtracks

3.4 k-Trimming

In Apriori-based approach a k+1 candidate is frequent if and only if all k subsets of the k+1 candidate are frequent as well. Because of the tree structured item-sets present in XML, a more specialized approach was needed in our implementation for checking whether all the k subsets from a k+1 sub-pattern are frequent. This problem occurs because in a tree structured item-set all nodes may be frequent but the actual structure of those nodes is infrequent. The approach taken was to generate all valid k-1 sub-patterns from the currently expanding k sub-pattern. The new node to be added was then added to all the k-1 sub-patterns and only if each one was frequent the new candidate k+1 sub-pattern would be generated.
For clarity purposes in the rest of this section when term node is used it is referred to the string representation of the node. The valid sub-patterns were generated by removing each of the nodes in

the string representation and adding the new node. The string representation had to be adjusted to preserve the structure after a removal of a node
Let s denote the size of the k sub-pattern, then the number of valid k-1 sub-patterns will be either s-1 or s depending on whether the removal of the root cause a forest or not, respectively. To preserve the structure of the new sub-pattern it had to be detected whether a removal of a node had to be accompanied with the removal of a backslash in the string representation. The following logic took care of these constraints:

- We remove node at a certain position in the string representation;
- We traverse the rest of the string and every time a node is encountered we increment the counter (initially 0), and when a backslash is encountered we decrement the counter;
- If the counter ever reaches -1 we remove the back-slash that caused it to go to -1.

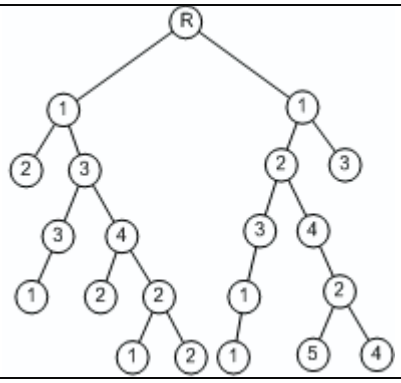
Same logic was applied to determine whether the removal of the root causes a forest. This occurred if the counter reached 0 and at this time we would not create the sub-pattern with the root removed as it is invalid. Progressively each node was removed and the produced sub-pattern was checked for frequency. Only if all the generated sub-patterns were frequent the candidate was generated with the new node. The same logic applied at the different levels of expansion of the tree as the string representation was changed to include the extra backslashes.
As an illustrative example consider the string representation of "A B C / / D" and we want to append node "E" at D. First we detect that the root A cannot be removed as the counter becomes 0 after reading the second "\". Then we remove "B" and we remove the second "/" as at that time the counter will be -1 and the sub-pattern to be checked for frequency becomes "A C / D E". In same manner the sub-patterns "A B / D E" and "A B C / / E" are generated. If we backtrack in the original tree and we are at "A" node, the new string representation becomes "A B C / / D /" and the new sub-patterns stay the same except that there is an extra "/" before the new node that is added.

4 Results & Discussion

We tested our algorithm by using the same data type that is used by the TreeMiner for the purpose of comparison. TreeMiner defines weighted and non-weighted support. For the purpose of mining semi-structured type of data weighted support is considered to be more appropriate.

Zaki developed two variants of TreeMiner: VTreeMiner and HTreeMiner. He reported in **[Zaki02]** that VTreeMiner execute the mining task at about 4 to 20 times faster than HTreeMiner. However, apart from reported result in **[Zaki02]** our experiment result shows that VTreeMiner reports all embedding subtree while HTreeMiner reports all embedding subtree that contains only frequent subtree. In other words, VTreeMiner doesn't trim k-tsis that contain infrequent {(k-1), …, 0}-tsis (called sub-tsis). Consequently, HTreeMiner does more processing to trim k-tsis that contain infrequent sub-tsis. As the result, VTreeMiner not only faster than HTreeMiner but also reports far more candidates than HTreeMiner. On the other hand, our approach checks and trims all tsis that contain infrequent sub-tsis. The experimental result shown below confirms our result with HTreeMiner.

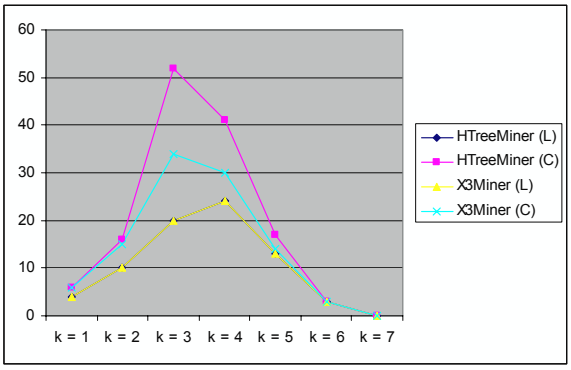
We also run the benchmarking on parallel Xeon processor PIII 1 GHz machine with 1024 MB RAM. Some test results are presented in the table below:



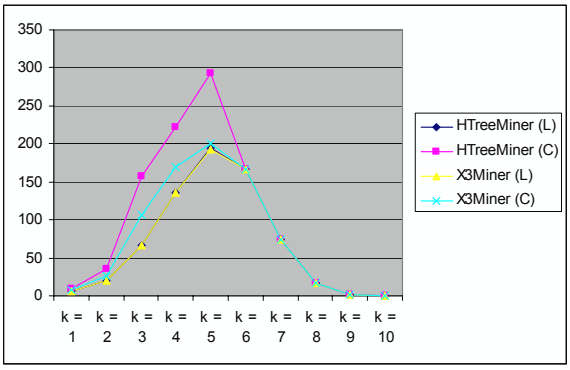
VTreeMiner	HTreeMiner	X3-Miner
k = 1		
2 – 3	1 - 6	1 - 6
3 - 4	2 - 6	2 - 6
4 - 6	3 - 4	3 - 4
5 – 6	4 – 3	4 - 3
k = 2		
4 2 – 4	1 1 - 5	1 1 - 5
3 1 - 5	1 2 - 6	1 2 - 6
3 2 – 3	1 3 - 4	1 3 - 4
1 4 - 3	1 4 - 3	1 4 - 3
1 3 - 4	2 1 - 3	2 1 - 3
1 1 - 5	2 2 - 2	2 2 - 2
1 2 – 6	2 4 - 3	2 4 - 3
2 4 - 3	3 1 - 5	3 1 - 5
2 1 - 3	3 2 - 3	3 2 - 3
2 2 - 2	4 2 – 4	4 2 - 4
k = 3		
1 1 / 2 – 6	1 1 / 2 - 6	1 1 / 2 - 6
1 1 / 3 - 2	1 1 / 3 - 2	1 1 / 3 - 2
1 1 / 4 - 5	1 1 / 4 - 5	1 1 / 4 - 5
1 2 / 1 - 3	1 2 / 1 - 3	1 2 / 1 - 3
1 2 / 2 - 5	1 2 / 2 - 5	1 2 / 2 - 5
1 2 1 - 3	1 2 1 - 3	1 2 1 - 3
1 2 2 – 2	1 2 2 - 2	1 2 2 - 2
1 2 / 3 - 4	1 2 / 3 - 4	1 2 / 3 - 4
1 2 4 - 3	1 2 4 - 3	1 2 4 - 3
1 3 1 - 5	1 3 1 - 5	1 3 1 - 5
1 3 / 2 - 4	1 3 / 2 - 4	1 3 / 2 - 4
1 3 2 – 3	1 3 2 - 3	1 3 2 - 3
1 3 / 4 - 3	1 3 / 4 - 3	1 3 / 4 - 3
1 4 2 - 4	1 4 2 - 4	1 4 2 - 4
1 4 / 3 - 2	1 4 / 3 - 2	1 4 / 3 - 2
2 1 / 2 - 3	2 1 / 2 - 3	2 1 / 2 - 3
2 1 / 4 – 4	2 1 / 4 - 4	2 1 / 4 - 4
3 1 / 2 – 4	3 1 / 2 - 4	3 1 / 2 - 4
3 2 / 2 – 2	3 2 / 2 - 2	3 2 / 2 - 2
4 2 / 2 – 2	4 2 / 2 - 2	4 2 / 2 - 2
k = 4		
1 1 / 2 / 2 – 2	1 1 / 2 / 2 - 2	1 1 / 2 / 2 - 2
1 1 / 2 / 3 - 2	1 1 / 2 / 3 - 2	1 1 / 2 / 3 - 2
1 1 / 2 4 - 2	1 1 / 2 4 - 2	1 1 / 2 4 - 2
1 1 / 4 2 – 5	1 1 / 4 2 - 5	1 1 / 4 2 - 5
1 1 / 4 / 3 - 4	1 1 / 4 / 3 - 4	1 1 / 4 / 3 - 4
1 2 / 1 / 2 – 5	1 2 / 1 / 2 - 5	1 2 / 1 / 2 - 5
1 2 1 / 2 - 3	1 2 1 / 2 - 3	1 2 1 / 2 - 3
1 2 1 / / 3 – 2	1 2 1 / / 3 - 2	1 2 1 / / 3 - 2
1 2 1 / 4 - 4	1 2 1 / 4 - 4	1 2 1 / 4 - 4
1 2 / 2 1 - 2	1 2 / 2 1 - 2	1 2 / 2 1 - 2
1 2 / 2 / 2 - 2	1 2 / 2 / 2 - 2	1 2 / 2 / 2 - 2
1 2 / 2 2 – 2	1 2 / 2 2 - 2	1 2 / 2 2 - 2
1 2 / 3 1 - 3	1 2 / 3 1 - 3	1 2 / 3 1 - 3
1 2 / 3 / 2 - 3	1 2 / 3 / 2 - 3	1 2 / 3 / 2 - 3
1 2 / 3 2 – 3	1 2 / 3 2 - 3	1 2 / 3 2 - 3
1 2 4 / / 3 – 3	1 2 4 / / 3 - 3	1 2 4 / / 3 - 3
1 3 / 2 / 2 – 2	1 3 1 / / 2 - 5	1 3 1 / / 2 - 5
1 3 1 / / 4 - 5	1 3 1 / 2 - 4	1 3 1 / 2 - 4
1 3 1 / 2 - 4	1 3 1 / / 4 - 5	1 3 1 / / 4 - 5
1 3 2 / 2 – 2	1 3 / 2 / 2 - 2	1 3 / 2 / 2 - 2
1 3 1 / / 2 – 5	1 3 2 / 2 - 2	1 3 2 / 2 - 2
1 3 / 4 2 – 4	1 3 / 4 2 - 4	1 3 / 4 2 - 4
1 4 2 / 2 – 2	1 4 2 / 2 - 2	1 4 2 / 2 - 2

3 1 / 2 / 2 – 2 **2 1 / 2 4 – 2** **2 1 / 4 2 - 2** **2 1 / 4 4 – 2** **1 1 / 4 4 - 2**	3 1 / 2 / 2 - 2	3 1 / 2 / 2 - 2
k = 5		
1 1 / 4 4 / / 3 – 2 1 1 / 4 2 / 2 – 2 1 2 / 1 / 2 / 2 – 2 1 2 1 / 4 / / 3 - 4 1 2 / 2 1 / 2 – 2 1 2 / 3 1 / / 2 - 3 1 2 / 3 1 / 2 – 4 1 2 / 3 / 2 / 2 – 2 1 2 / 3 2 / 2 – 2 1 3 1 / 2 / 2 – 2 1 3 1 / / 2 / 2 – 2 1 3 1 / / 4 2 – 5 1 3 / 4 2 / 2 – 2 **1 3 1 / / 2 4 - 2** **1 3 1 / / 4 4 - 2** **1 2 1 / 2 / / 3 - 2** **1 2 1 / 2 4 – 2** **1 2 1 / 4 2 - 2** **1 2 1 / 4 4 – 2** **2 1 / 4 2 4 – 2** **1 1 / 2 4 / / 3 – 2** **1 1 / 4 2 4 - 2** **1 1 / 4 2 / / 3 – 2** **1 3 1 / / 4 4 - 2** **1 3 1 / / 4 2 – 5**	1 1 / 2 4 / / 3 - 2 1 1 / 4 2 / 2 - 2 1 2 / 1 / 2 / 2 - 2 1 2 1 / 4 / / 3 - 4 1 2 / 2 1 / 2 - 2 1 2 / 3 1 / / 2 - 3 1 2 / 3 1 / 2 - 4 1 2 / 3 / 2 / 2 - 2 1 2 / 3 2 / 2 - 2 1 3 1 / / 2 / 2 - 2 1 3 1 / 2 / 2 - 2 1 3 1 / / 4 2 - 5 1 3 / 4 2 / 2 - 2	1 1 / 2 4 / / 3 - 2 1 1 / 4 2 / 2 - 2 1 2 / 1 / 2 / 2 - 2 1 2 1 / 4 / / 3 - 4 1 2 / 2 1 / 2 - 2 1 2 / 3 1 / / 2 - 3 1 2 / 3 1 / 2 - 4 1 2 / 3 / 2 / 2 - 2 1 2 / 3 2 / 2 - 2 1 3 1 / / 2 / 2 - 2 1 3 1 / 2 / 2 - 2 1 3 1 / / 4 2 - 5 1 3 / 4 2 / 2 - 2
k = 6		
1 2 / 3 1 / / 2 / 2 – 2 1 2 / 3 1 / 2 / 2 - 2 1 3 1 / / 4 2 / 2 – 2 **1 3 1 / / 4 2 4 - 2** **1 1 / 4 2 4 / / / 3 – 2** **1 2 1 / 2 4 / / / 3 – 2** **1 2 1 / 4 2 4 - 2** **1 2 1 / 4 2 / / / 3 – 2** **1 2 1 / 4 4 / / / 3 – 2**	1 2 / 3 1 / / 2 / 2 - 2 1 2 / 3 1 / 2 / 2 - 2 1 3 1 / / 4 2 / 2 - 2	1 2 / 3 1 / / 2 / 2 - 2 1 2 / 3 1 / 2 / 2 - 2 1 3 1 / / 4 2 / 2 - 2
k = 7		
1 2 1 / 4 2 4 / / / / 3 – 2		

Table 1. Algorithms results & comparison on artificial data



tree3.asc



tree4.asc

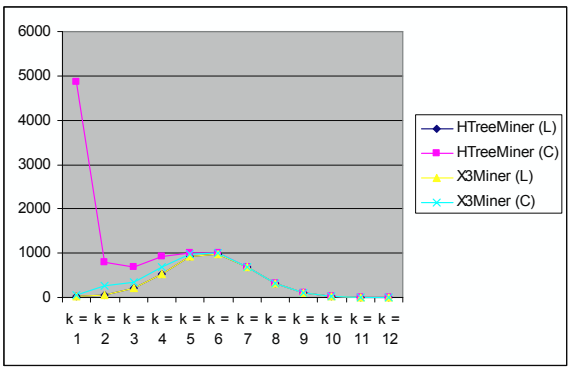


Figure 12. Graph of HTreeMiner and X3Miner Candidate Generations

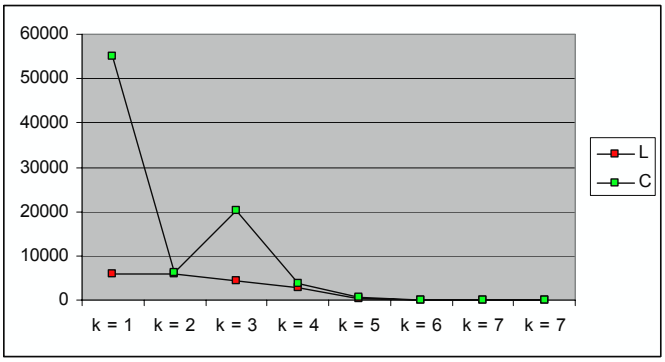


Figure 13. Graph plotted of X3-Miner on dblp data with number of nodes = 123012

```
article year[1984] / journal[J. Comb. Theory, Ser. A]  - 71
article year[1993] / volume[11] / journal[Image Vision Comput.]  - 66
article year[2003] / volume[19] / journal[Future Generation Comp. Syst.]  - 120
```

Figure 14. Some Frequent Items from dblp data with number of nodes = 123012

From the Figure 12 above it is shown that X3-Miner in overall generates less candidates for the same number of frequent itemsets as TreeMiner. It can be seen from the graph above that the difference between the candidates curve and the frequent itemsets curve is smaller than the TreeMiner's.

On the other hand, the TreeMiner is a very efficient algorithm in mining frequent tree structures within a forest. We compared our algorithm with TreeMiner and the same frequent itemsets were detected. The difference lies in the fact that our algorithm generated less candidates due to the model-validating approach applied in comparison to generating candidates using the join operation as done in most of the apriori based approach such as the TreeMiner. However TreeMiner performed the task in less time.

This gain in efficiency could be due to the assumed file structure in TreeMiner where the tree is represented as a list of integers rather than an XML document. Processing and hashing integers is

more efficient than processing strings. As our approach actually takes in XML documents as input and takes care of the structure and the values of the attributes the implementation had to be done in the way where strings were processed and hashed. Our observation indicates that this is where the extra computational cost comes from. Theoretically our algorithm should execute the task faster as it generates less candidates at each step. The larger the number of the infrequent candidates generated the more time needed for processing.

With the XML data set we use a cut-down version of dblp.xml database with 123012 number of nodes. Figure 14, shows some frequent tree structured items represented in string representation. The result tells us that there are a significant number of article published in year 2003 with volume 9 in journal of Future Generation Comp. Syst. in the dblp file. So is with the number of article published in Journal of J. Comb. Theory, Ser. A in 1984.

In reality the two algorithms are incompatible for efficiency comparisons due to different type of data being processed. This caused different implementation issues as mentioned above implementations differ because of different constraints imposed on the type of data that is processed. Optimization will be the focus of our future works.

5 Conclusions

The main strength of the proposed approach is that the candidate generation is model-validating and so there is no time wasted in generating invalid candidates that are discarded at a later stage. The algorithm can process an XML document directly taking into account the values of the nodes present in the XML tree. The frequent item-sets generated will contain node names and values in comparison to the TreeMiner approach which only generates frequent tree structures and does not process an XML document in the form that is mostly present. The frequent item-sets generated by our approach are in the XML format that can be interpreted as association rules supported by the minimum support provided by the user.

The current way the string representation of a sub-pattern is used as the key in the hash multimap may be the cause for extra computational cost as we include the values and names of each node and so when hashcode is calculated it is more expensive. The algorithm is still performing candidate generation efficiently on large XML documents but as frequent sub-patterns grow there will be a large increase in the computational cost. Some of the immediate extensions to the algorithm will be to investigate the bottleneck of performance and find a more efficient way for storing and retrieving the formed candidate sub-patterns. In the current work we have shown that candidate generation for XML documents does not have to follow the traditional approach of performing joins on frequent item-sets and in a sense generating all the possible candidates blindly without consulting the document model. Moreover, from the theoretical perspective it is always preferred to have a model based upon which reasoning is validated.

The major extensions to the current work will be to perform rule extraction on the extracted interesting patterns from XML document. This will improve the task as there will be no irrelevant and/or uninteresting patterns to interfere with the learning mechanism.

6 REFERENCES

1. **[Abe02]** K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized substructure discovery for semi-structured data. In Proceedings of the 6th European Conference on Principles and Practice of Data Mining and Knowledge Discovery (PKDD-2002), pages 1–14, 2002.
2. **[Agr93]** Agrawal, R., T. Imielinski, et al. (1993). Mining Association Rules between Sets of Items in Large Databases. Proceedings of the 1993 ACM SIGMOD Conference, Washington DC, USA.
3. **[Agr94]** Agrawal, R. and R. Srikant (1994). Fast Algorithms for Mining Association Rules. 20th Int. Conf. Very Large Databases, VLDB, Santiago de Chile, Chile.
4. **[Agr95]** R. Agrawal & Ramakrishnan Srikant. Mining Sequential Patterns. In 11th Intl. Conf. on Data Engg., 1995.
5. **[Agr96]** Agrawal, R., H. Mannila, et al. (1996). "Fast Discovery of Association Rules." Advances in Knowledge Discovery and Data Mining, AAAI Press: 307-328.
6. **[AKA94]** Sestito S and Dillon T. *Automated Knowledge Acquisition*. Prentice Hall. 1994.
7. **[Asai01]** Asai, T., K. Abe, et al. (2001). Efficient Substructure Discovery from Large Semi-Structured Data. Fukuoka, Japan, Department of Informatics, Kyushu University.
8. **[Bayardo1998]** Jr., R. J. B. (1998). Efficiently Mining Long Patterns from Databases. SIGMOD' 98, Seattle, WA, USA, ACM.
9. **[Braga02]** Braga, D., A. Campi, et al. (2002). "A Tool for Extracting XML Association Rules." 14th IEEE International Conf. on Tools with Artificial Intelligence (ICTAI'02).
10. **[Brin97]** Brin, S., R. Motwani, et al. (1997). Dynamic Itemset Counting and Implication Rules for Market Basket Data. ACM SIGMOD Conf. Management of Data, Tucson, Arizona, USA.
11. **[CHARM02]** Zaki, M. J. and C.-J. Hsiao (2002). CHARM: An Efficient Algorithm for Closed Itemsets Mining. New York, USA, Computer Science Department, Rensselaer Polytechnic Institute Troy.
12. **[COFI03]** El-Haji, M. and O. R. Zaiane (2003). COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation. FIMI, Melbourne, Florida, USA.
13. **[Cong02]** G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In Proc. of the 2nd SIAM Intl. Conf. on Data Mining, Virginia, USA, April 2002.
14. **[DHP97]** Park, J. S., M.-S. Chen, et al. (1997). "Using a Hash-Based Method with Transaction Trimming for Mining Association Rules." IEEE Transactions on Knowledge and Data Engineering 9(5): 813-825.
15. **[Feng03]** L. Feng, T. S. Dillon, H. Weigand, E. Chang. An XML-Enabled Association Rule Framework. In Proceedings of DEXA 2003, pp 88-97, Prague, Czech Republic, 2003.
16. **[Feng04]** Feng, L. & T. Dillon (2004). Mining XML-Enabled Association Rule with Templates. In Proceedings of KDID 04.
17. **[FSG01]** Kuramochi, M. and G. Karypis (2001). Frequent Subgraph Discovery. IEEE International Conference on Data Mining (ICDM), Mineapolis, Department of Computer Science/Army HPC Research Center University of Minnesota.
18. **[Han01]** Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

19. **[Han00]** J. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. In ACM SIGMOD Conf. Management of Data, May 2000.
20. **[Han04]** J. Han., Personal Communication, PAKDD Sydney 2004.
21. **[Hidber98]** Hidber, C. (1998). Online Association Rule Mining. Berkeley, International Computer Science Institute.
22. **[Inokuchi00]** A. Inokuchi, T.Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In Proceedings of the 4th European Conference on Principles and Practice of Data Mining and Knowledge Discovery (PKDD-2000), pages 13–23, 2000.
23. **[Inokuchi02a]** Inokuchi, A., T. Washio, et al. (2002). "Complete Mining of Frequent Patterns from Graphs." Kluwer Academic Publishers, Netherlands.
24. **[Inokuchi02b]** A. Inokuchi, T.Washio, Y. Nishimura, and H. Motoda. General framework for mining frequent patterns in structures. In Proceedings of the ICDM-2002 workshop on Active Mining (AM-2002), pages 23–30, 2002. Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM press, 2002.
25. **[Leung03]** H. Leung, F. Chung , & S. C. Chan. A New Sequential Mining Approach to XML Document Similarity Computation. Technical Report. Department of Computing Hong Kong Polytechnic University, Hunghom, Kowloon, Hong Kong.
26. **[Meo96]** R. Meo, G. Psaila and S. Ceri. A New Operator for Mining Association Rules, in Proceeding of VLDB'96, pp. 122-133, Bombay, India, Sept. 1996.
27. **[Meo98]** R. Meo, G. Psaila and S. Ceri. A Tightly-coupled Architecture for Data Mining, in Proceedings of ICDE'98, pp. 316-323, Orlando, FL, USA, Feb. 1998.
28. **[Pasquier99]** Pasquier, N., Y. Bastide, et al. (1999). Discovering Frequent Closed Itemsets for Association Rules. ICDT '99, Jerusalem, Israel.
29. **[Pei00]** Pei, J., J. Han, et al. (2000). CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery, Dallas, Texas, USA.
30. **[Toivonen96]** Toivonen, H. (1996). Sampling Large Databases for Association Rules. 22nd VLDB Conference, Mumbai (Bombay), India.
31. **[TOPK02]** Han, J., J. Wang, et al. (2002). Mining Top-K Frequent Closed Patterns without Minimum Support. ICDM '02, Illinois, USA, University of Illinois at Urbana-Champaign.
32. **[TreeFinder02]** A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), pages 450–457, 2002.
33. **[XRules03]** Zaki, M. J. and C. C. Aggarwal (2003). XRules: An Effective Structural Classifier for XML Data. SIGKDD '03, Washington DC, USA.
34. **[Wang97]** K. Wang and H. Liu. Schema Discovery for semi-structured data. In Proc. of the 3rd. International Conference on Knowledge Discovery and Data Mining, pages 271-274, California, USA, August 1997.
35. **[Wang98]** K. Wang and H. Liu. Discovering typical structures of documents: a road map approach. In Proc. of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pages 146-154, Melbourne, Australia, August 1998.
36. **[Wang00]** K. Wang and H. Liu. Discovering structural association of semistructured data. IEEE Transactions on Knowledge and Data Engineering, 12(2):353-371, 2000.
37. **[Washio02]** Washio, T., A. Inokuchi, et al. (2002). A General Framework for Mining Patterns in Structures. Yomato, Kanagawa, Japan, Tokyo Research Laboratory, IBM Japan.

38. **[Zaki98]** Zaki, M. J. and M. Ogihara (1998). Theoritical Foundations of Association Rules. DMKD, Seattle, WA, USA.
39. **[Zaki02]** Zaki, M. J. (2002). Efficient Mining of Trees in the Forest. SIGKDD '02, Edmonton, Alberta, Canada, ACM.
40. **[Zaki03]** Zaki, M. J. and K. Gouda (2003). Fast Vertical Mining using Diffsets. SIGKDD 2003, Washington DC, USA.
41. **[Zhang04]** J. Zhang, T. W. Ling, R. M. Bruckner, A. M. Tjoa, H. Liu. On Efficient and Effective Association Rule Mining from XML Data. In Proceedings of DEXA 2004, LNCS 3180, pp. 497 - 507, Zaragosa, Spain, 2004.