X3-Miner: mining patterns from an XML database

H. Tan[1], T. S. Dillon[1], L. Feng[2], E. Chang[3] & F. Hadzic[1]
[1]*Faculty of Information Technology, University of Technology Sydney, Australia*
[2]*University of Twente, The Netherlands*
[3]*Curtin University, Australia*

Abstract

An XML enabled framework for the representation of association rules in databases was first presented in [4]. In Frequent Structure Mining (FSM), one of the popular approaches is to use *graph matching* that use data structures such as the adjacency matrix [7] or adjacency list [8]. Another approach represents semi-structured tree-like structures using a string representation, which is more space efficient and relatively easy for manipulation [10]. However, with XML, mining association rules are faced with more challenges due to the inherent flexibilities in both structure and semantics, such as: 1) more complicated hierarchical data structure; 2) ordered data context; and 3) much bigger data size. To tackle these challenges, 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 by taking into account the semantics embedded in the tree-like structure in an XML database and obtain only valid candidates out of the XML database; (2) minimising I/O overhead by intersecting XML database with the frequent 1-itemset. This results in a frequent 1-itemset XML tree. 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 [10] to *xstring* for describing an XML document 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: association mining, tree, algorithm, semantic relationships, XML.

WIT Transactions on Information and Communication Technologies, Vol 35, © 2005 WIT Press
www.witpress.com, ISSN 1743-3517 (on-line)

1 Introduction

In several years we have seen tremendous works in the area of mining graph, sequence and tree data for patterns of interest [4, 5, 7, 8, 10, 11]. Association mining has been so successful in discovering useful associations between data [1, 3, 4, 5, 9, 10]. Most of the works done in association mining [1, 6, 9] were tailored for structured data but only few for semi-structured data [2, 4, 5, 10, 11]; especially where the order is important and schema is not fixed. While some approaches have focused on mining for patterns in databases containing general graphs [7, 8], 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 [2, 4, 5, 10, 11]. Our research takes a step on developing an algorithm that is well suited to the characteristics of the domain described in [4, 5] that is characterized by: 1) a more complicated hierarchical data structure; 2) an ordered data context; and 3) a much bigger data size. It extends the notion of associated items to XML fragments to present associations among trees. Despite the strong foundation established in [4], however, an efficient way to implement the framework had not been discussed. Ling et al. continue their work with a template model for mining XML-enabled association rules [5]. Zaki presented TreeMiner [10], an algorithm to discover all frequent sub-trees in a forest using a data structure called the scope-list. The developed algorithm was 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. In DHP [9], the 2-itemset candidate reduction is performed to overcome the performance bottleneck in Apriori. Recently, [11] 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.

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 its 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 [6], Apriori based approaches [1, 3, 7, 8, 9, 10] suffer greatly from an inherent problem in generating C_2. 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 have been generated with little knowledge of the actual data model. In Apriori [1], candidate generation, C_k, is done by joining (operator *) frequent (k-1)-itemsets, L_{k-1}, with itself, $L_{k-1} * L_{k-1}$. If the order is important and $|L_1|$ (size of L_1) equals to n, the complexity of generating 2-itemsets in Apriori turns out to be in $O(n^2)$ complexity class. Many candidates generated are useless, invalid or redundant. A systematic guided candidate generation according to model validating rule is what we aim in this research. Able to do so, generation of invalid patterns can be minimized. Subsequently, 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.

Section 2 examines our three phases candidate generation in detail and presents our algorithm. Section 3 gives the problem decomposition. We empirically evaluate the performance of these algorithms and study their scale-up properties in section 4 and the last section concludes the paper.

2 Our approach

2.1 The three phases

To do candidate generation, a data structure to represent the XML data in space efficient way and easy to manipulate need to be constructed. Our approach transforms the XML data into a string like presentation called *xstring*. Our string representation differ with TreeMiner in two ways, firstly we incorporate a dpp into our string representation. Secondly, we utilize the *xstring* to generate candidate through guided model approach. For each of the element read from the XML document the triplet, as introduced earlier, [pos, $scope_{max}$](dpp), is constructed. The next phase is to generate 1-tsis. Generating 1-tsis is known as generating C_1. 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. The 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. 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. 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.

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 candidates through join operations. Most of the constructed 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 as our approach takes advantage of the coordinates in the *xstring*. For each of the *tsi* constructed from the database the coordinate introduced conceptually adds visioning ability. 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 early intersection of frequent 1-itemset with the databases. We construct a hashing approach to suit our solution for applying association mining to XML database using *xstring* as the hash key. One of our novel approaches in attempt to bring the complexity of C_2 into a manageable *O(kn)* complexity, where for the worst case scenario *k* represents the depth of the tree and *n* the tree

size. The formal prove would be left explained in our future work. Throughout the candidate generation process, the algorithm trims the string that contains infrequent substring so that it adheres with the downward closure lemma [1].

2.2 Problem statement

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*. XML data is a rooted well-formed tree [4]. 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. XML document frequently modelled using a labelled ordered tree. Because of the space limitation, we leave readers to consult [4, 10] for some definition in regards to tree structure includes ancestor-descendant relationship between tags. Embedded subtrees are a generalization of induced subtrees [7]. They allow not only direct parent-child edges, but also ancestor-descendant edges. Via embedded subtrees, hidden pattern deep within large trees can be detected. In this paper, we consider embedded subtree and induced subtree. The difference between embedded subtree and induced subtree can be seen in [4, 10].

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 $n \in N$ 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 (dpp) refer to its parent node position. Each ***node position*** has an integer number, i, start from 0 following its position (coordinate) in depth-first (or pre-order) traversal of the tree. The **scope** is defined as the descendant's node position. The scope of node is defined such as such as $\{(n,m) \mid n \leq m, n, m \text{ is positive integer}\}$. All nodes within the scope of node n would be its children or descendants, except itself. The second integer, m (**scope$_{max}$**), in scope pair describes the position of its right most descendant according to pre-order traversal. For more details explanation of how scope pair works reader can consult [10]. Also, to directly reference a parent node of any particular node, a ***direct parent pointer*** (dpp) notion is introduced. 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.

XML document mining problem. The challenge of association mining is in fact to have efficient and scalable process in discovering large itemsets [6, 9]. After all large itemsets discovered the generation of association rules can be derived more straightforwardly. The paper concentrates on developing discovering large itemsets and apply the technique to XML data (semi-structured data). The problem of finding frequent patterns from XML data deals with tree-like structure and to present associations among trees rather than simple-structured items of atomic values. Consequently, 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 (database of trees). Let $S \therefore T$ (read: T embed S) 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 $S \therefore T$. Zaki [10] proposed 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.

```
/* 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 1: Frequent sets generation pseudo-code.

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.

```
/* 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 2: Candidate generation pseudo-code.

```
/* 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 3: Update backtracks.

4 Results and 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 [10] that VTreeMiner execute the mining task at about 4 to 20 times faster than HTreeMiner. However, apart from reported result 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 below:

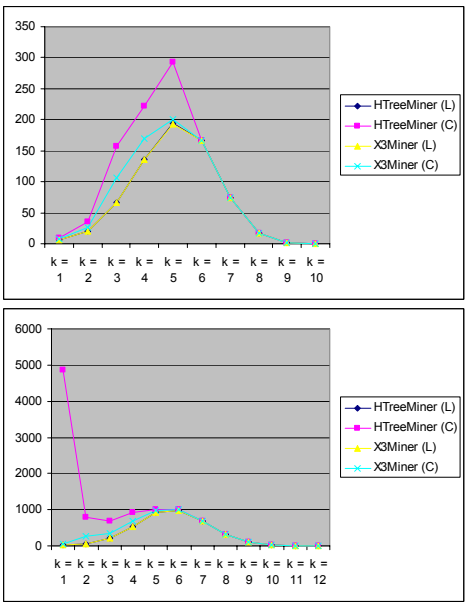

Figure 4: Graph of HTreeMiner & X3-Miner candidates generation. Vertical & horizontal line representing the number of candidates & iteration respectively.

From the fig. 4 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.

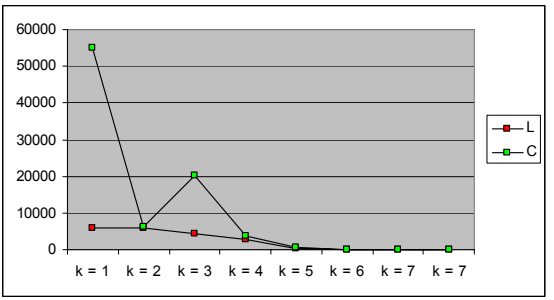

Figure 5: Graph of X3-Miner running dblp data (nodes = 123012).

```
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 6: Frequent items of dblp (nodes = 123012).

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 fewer 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. Fig. 6, 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 patterns to interfere with the learning mechanism.

References

[1] Agrawal, R., Mannila, H., et al., "Fast Discovery of Association Rules." *Advances in Knowledge Discovery and Data Mining*, AAAI Press: 307-328, 1996.

[2] Asai, T., Abe, K., et al., Efficient Substructure Discovery from Large Semi-Structured Data. Fukuoka, Japan, Department of Informatics, Kyushu University, 2001.

[3] Bayardo, R. J., Efficiently Mining Long Patterns from Databases. *SIGMOD' 98*, Seattle, WA, USA, ACM, 1998.

[4] Feng, L., Dillon, T. S., Weigand, H., Chang, E., An XML-Enabled Association Rule Framework. *In Proceedings of DEXA 2003*, pp 88-97, Prague, Czech Republic, 2003.

[5] Feng, L. & Dillon, T. S., Mining XML-Enabled Association Rule with Templates. *In Proceedings of KDID 04*, 2004.
[6] Han, J., Pei, J. & Yin, Y., Mining Frequent Patterns without Candidate Generation. *In ACM SIGMOD Conf. Management of Data*, May 2000.
[7] Inokuchi, A., Washio, T., Nishimura, Y., & Motoda, H., General framework for mining frequent patterns in structures. *In Proceedings of the ICDM-2002 workshop on Active Mining (AM-2002)*, pages 23–30, 2002.
[8] Kuramochi, M. & Karypis, G., Frequent Subgraph Discovery. IEEE International Conference on Data Mining (ICDM), Mineapolis, Department of Computer Science/Army HPC Research Center University of Minnesota, 2001.
[9] Park, J. S., Chen, M.-S., et al., Using a Hash-Based Method with Transaction Trimming for Mining Association Rules. *IEEE Transactions on Knowledge and Data Engineering* 9(5): 813-825, 1997.
[10] Zaki, M. J., Efficient Mining of Trees in the Forest. *SIGKDD '02*, Edmonton, Alberta, Canada, ACM, 2002.
[11] Zhang, J., Ling, T. W., Bruckner, R. M., Tjoa, A. M., Liu, H., On Efficient and Effective Association Rule Mining from XML Data. *In Proceedings of DEXA 2004*, LNCS 3180, pp. 497 - 507, Zaragosa, Spain, 2004.