Discovering Minimal Infrequent Structures from XML Documents

Wang Lian Nikos Mamoulis David W. Cheung S. M. Yiu

Department of Computer Science and Information Systems,
The University of Hong Kong, Pokfulam, Hong Kong.
{*wlian, nikos, dcheung, smyiu*}*@csis.hku.hk*

Abstract. More and more data (documents) are wrapped in XML format. Mining these documents involves mining the corresponding XML structures. However, the semi-structured (tree structured) XML makes it somewhat difficult for traditional data mining algorithms to work properly. Recently, several new algorithms were proposed to mine XML documents. These algorithms mainly focus on mining frequent tree structures from XML documents. However, none of them was designed for mining infrequent structures which are also important in many applications, such as query processing and identification of exceptional cases. In this paper, we consider the problem of identifying infrequent tree structures from XML documents. Intuitively, if a tree structure is infrequent, all tree structures that contain this subtree is also infrequent. So, we propose to consider the minimal infrequent structure (MIS), which is an infrequent structure while all proper subtrees of it are frequent. We also derive a level-wise mining algorithm that makes use of the SG-tree (signature tree) and some effective pruning techniques to efficiently discover all MIS. We validate the efficiency and feasibility of our methods through experiments on both synthetic and real data.

1 Introduction

The standardized, simple, self describing nature of XML makes it a good choice for data exchange and storage on the World Wide Web. More and more documents are wrapped in XML format nowadays. To process large amount of XML documents more efficiently, we can rely on data mining techniques to get more insight into the characteristics of the XML data, so as to design a good database schema; to construct an efficient storage strategy; and especially for enhancing query performance. Unlike traditional structured data, XML document is classified as semi-structured data. Besides data, its tree structure always embeds significant sementic meaning. Therefore, efficiently mining useful tree structures from XML documents now attract more and more attention.

In recent years, several algorithms that can efficiently discover frequent tree structures are available. However, discovering infrequent tree structures is also a very important subject. In fact, infrequent tree structures carry more information than frequent structures from the information theory point of view. There are many applications that can make use of the infrequent tree structures such as query optimization, intrusion detection and identification of abnormal cases. Unfortunately, as far as we know, there are no papers that formally address this problem. In this paper, we present our work

in mining infrequent structures. Because any superstructure of an infrequent structure must be infrequent, identifying all infrequent structures is impractical as the number of them will be huge. On the other hand, we propose to discover a special kind of infrequent structure: the *Minimal Infrequent Structure* (MIS), which is itself infrequent but all its substructures being frequent. The role of MIS is similar to that of negative border on itemsets [8]. Based on MIS, we can easily identify all infrequent structures by constructing superstructures from MIS.

Several methods have been proposed for the problem of mining frequent tree structures in semi-structure data [2][6] [10][12]. Most of them are based on the classical Apriori technique [1], however, these methods are rather slow when applying to find MIS (the number of MIS is rather small, whereas the number of frequent structures is very large, a lot of time is spent on counting the support of frequent structures). In order to accelerate the discovery of MIS, we propose the following three-phase data mining algorithm. In phase one, we scan the data to derive the edge-based summary (signature) of each XML document. In phase two, we run Apriori on the summaries to quickly generate a set of frequent structures of size k. In phase three, we remove the frequent structures so as to identify the minimal infrequent structures. Experimental results demonstrate that this three-phase data mining algorithm indeed achieves a significant speed improvement.

The contribution of this paper is two-fold:

- We introduce a useful structure: Minimal Infrequent Structure as a base for representing all infrequent structures in XML documents.
- We propose an efficient level-wise data mining algorithm that discovers Minimal Infrequent Structures in XML data, which can also be used in other mining applications.

The rest of the paper is organized as follows. Section 2 provides background on existing work. In Section 3, we present the data mining algorithm that finds all MIS. In Section 4, we evaluate our proposed methods on both synthetic and real data. Finally, Section 5 concludes the paper with a discussion about future work.

2 Related Work

According to our knowledge, there are no papers that have discussed the problem of mining infrequent tree structures in XML documents. On the other hand, there are several algorithms for mining frequent tree structures in trees [2][6] [10][12] based on Apriori [1]. Starting from frequent vertices, the occurrences of more complex structures are counted by adding an edge to the frequent structures of the previous level. The major difference among these algorithms is on the candidate generation and the counting processes.

In [2], a mining technique that enumerates subtrees in semi-structured data efficiently, and a candidate generation procedure that ensures no misses, was proposed. The tree enumeration works as follows: for each frequent structure s, the next-level candidate subtrees with one more edge are generated by adding frequent edges to its rightmost path. That is, we first locate the right most leaf r, traverse back to the root,

and extend each node visited during the backward traversal. This technique enumerates the occurrences of trees relatively quickly, but fails to prune candidates early, since the final goal is to discover frequent structures. A similar candidate generation technique is applied in TREEMINER [12]. This method also fails to prune candidates at an early stage, although the counting efficiency for each candidate is improved with the help of a special encoding schema.

In [6], simple algorithms for canonical labeling and graph isomorphism are used, but they do not scale well and cannot be applied to large graphs. In [10], complicated pruning techniques are incorporated to reduce the size of candidates, however the method discovers only collections of paths in ordered trees, rather than arbitrary frequent trees. Our work is also related to FastXMiner [11], discovers frequently asked query patterns and their results are intelligently arranged in the system cache in order to improve future query performance. This method is also Apriori-like, however, it only discovers frequent query patterns rooted at the root of DTD, whereas our mining algorithm does not have this limitation.

Among the studies on association rule discovery, the Maxminer in [3] is the most related work to this paper. Maxminer is equipped with an efficient enumeration technique based on set-enumeration tree to overcome the inefficiency of lattice-based itemset enumeration. On the other hand, our data mining method uses the enumeration method in [2] to generate the candidates level-by-level, but we apply more effective pruning techniques to reduce the number of candidates; a generated candidate is pruned if any of its substructures are not in the set of frequent structures generated in previous level. Furthermore, we use a novel approach which performs counting on a SG–tree [7], a tree of signatures, to quickly identify possible large frequent structures of a particular size. The mining algorithm on the exact structures then counts only candidates which are not substructures of the frequent structures already discovered. This greatly reduces the number of candidates that need to be counted and speeds up the mining process significantly. The details of the new data mining method will be described in the next section.

Other related work includes [8] and [5]. In [8], the concept of negative border is introduced, which is a set of infrequent itemsets, where all subsets of each infrequent itemset are frequent. Conceptually, MIS is similar to negative border. In [5], WARMR is developed on first-order models and graph structures. This algorithm can be applied for frequent tree discovery, since tree is a special case of graph structures. However its candidate generation function is over-powerful, it produces many duplicated candidates.

3 Discovery of MIS

3.1 Problem Definition

Before defining our data mining problem, we introduce some concepts that are related to the problem formulation.

Definition 1. *Let L be the set of labels found in an XML database. A **structure** is a node-labelled tree, where nodes are labels from L. Given two structures, s_1 and s_2, if s_1 can be derived by removing recursively $l \geq 0$ nodes (which are either leaf nodes*

*or root nodes) from s_2 then s_1 is a **substructure** of s_2. In this case we also say that s_2 **contains** s_1, or that s_2 is a **superstructure** of s_1. Finally, the $size$ of a structure s $size(s)$ is defined by the number of edges in it.*

If a structure contains only one element, we assume the size of it is zero. Assuming that $L = \{a, b, c, d, e\}$, two potential structures with respect to L are $s_1 = (a(b)(c(a)))^{\star}$ and $s_2 = (a(c))$; s_2 is a substructure of s_1, or s_1 contains s_2.

Definition 2. *Given a set D of structures, the **support** $sup(s)$ of a structure s in D is defined as the number of structures in D, which contain s. Given a user input threshold ρ, if $sup(s) \geq \rho \times |D|$, then s is **frequent** in D, otherwise it is **infrequent**.*
*(1) If $size(s) \geq 1$ and $sup(s) < \rho \times |D|$, and for each substructure s_x of s, $sup(s_x) \geq \rho \times |D|$, then s is a **Minimal Infrequent Structure**(MIS)**.*
(2) If $size(s) = 0$ and $sup(s) < \rho \times |D|$, then s is a MIS.

In practice, some MIS could be arbitrarily large and potentially not very useful, so we restrict our focus to structures that contain up to a maximum number of k edges. Note that the set D in Definition 2 can be regarded as a set of documents. Now, we are ready to define our problem formally as follows:

Definition 3. (problem definition)*: Given a document set D, and two user input parameters ρ and k, find the set S of all MIS with respect to D, such that for each $s \in S$, $size(s) \leq k$.*

The following theorem shows the relationship between MIS and an infrequent structure.

Theorem 1. Let D be a document set, and S be the set of MIS in D with respect to ρ and k. If an infrequent structure t contains at most k edges then it contains at least one MIS.

Proof In the cases where $size(t) = 0$ or all substructures of t are frequent, t itself is an MIS (of size at most k), thus it should be contained in the set of MIS. Now let us examine the case, where t has at least one infrequent proper substructure t'. If t' is MIS, then we have proven our claim. Otherwise, we can find a proper substructure of t' which is infrequent and apply the same test recursively, until we find a MIS (recall that a single node that is infrequent is also an MIS of size 0). □

Based on the above theorem, we can easily: (i)construct infrequent structures by generating superstructures from MIS. (ii)verify whether a strcture is infrequent by checking whether it contains a MIS or not.

Finally, in order to accelerate the data mining process we make use of a document abstraction called *signature*, which is defined as follow:

* We use brackets to represent parent/child relationship.
** From now on, we use MIS to represent both Minimal Infrequent Structure and Minimal Infrequent Structures.

Definition 4. *Assume that the total number of distinct edges in D is E, and consider an arbitrary order on them from 1 to E. Let $order(e)$ be the position of edge e in this order. For each $d \in D$, we define an E-length bitmap, $sig(d)$, called* ***signature****; $sig(d)$ has 1 in position $order(e)$ if and only if e is present in d. Similarly, the signature of a structure s is defined by an E-length bitmap $sig(s)$, which has 1 in position $order(e)$ if and only if e is present in s.*

The definition above applies not only for documents, but also for structures. Observe that if s_1 is a substructure of s_2, then $sig(s_1) \subseteq sig(s_2)$. Thus, signature can be used as a fast check on whether or not a structure can be contained in a document. On the other hand, it is possible that $sig(s_1) \subseteq sig(s_2)$ and s_1 is not a substructure of s_2 (e.g., $s_1 = (a(b(c))(b(d)))$, $s_2 = (a(b(c)(d)))$. Thus, we can only use the signature to find an *upper bound* of a structure's support in the database.

3.2 Mining MIS

We consider the problem of mining MIS as a three-phase process. The first phase is *preprocessing*. The document in D is scanned once for two purposes. (1) Find out all infrequent elements, infrequent edges and all frequent edges. The set of all frequent edges are stored in FE. We will use M to store the discovered MIS. (2) Compute signatures of all the documents and store them in an array SG. Besides the signature, SG also store the $support$ of the signature in D, which is the number of documents whose signatures match the signature. Since many documents will have the same signature, SG in general can fit into the memory; otherwise, the approach in MaxMiner[3] could be used to store SG on disk.

The second phase of the mining process is called *signature-based counting*, and is described by the pseudocode in Figure 1. In this phase, given a fixed k, we will compute the set of size-k frequent structures in D. These frequent structures will be stored in F_k. In the third phase, F_k will be used to narrow the search domain of MIS because no subset of a frequent structure in F_k can be an MIS. Note that k is a user input parameter which is the same as the maximum size of MIS that we want to mine.

```
/* Input  k, the maximum number of edges in a MIS*/
/* Output  F_k, all size k frequent structures*/
1).   F_prev = FE
2).   for i=2 to k
3).       candidates = genCandidate(F_prev, FE)
4).       if candidates == ∅ then break; F_prev = ∅
5).       for each c in candidates
6).           if sigsup(c) ≥ ρ × |D| w.r.t SG then move c to F_prev
7).   scan documents to count the support of each c ∈ F_prev
8).   for each c in F_prev
9).       if sup(c) ≥ ρ × |D| then insert c into F_k
10).  return F_k
```

Fig. 1. The Mining Algorithm – Phase 2

The algorithm in Figure 1 computes the frequent structures in an Apriori-like fashion. The procedure $genCandiates$ uses the method proposed in [2] to generate the

candidates. Frequent structures of the previous level (F_{prev}) and the edges in FE are used to generate candidate structures for the current level (line 3). Note that if the signature of a structure does not have the enough support, then it cannot be a frequent structure. (The reverse is not necessary true.) Therefore, we can use the information to remove many inadmissable candidates during the generation. Using SG to prune the candidates in F_{prev} will not wrongly remove a true frequent structure but may have false hits remaining in F_{prev}. Therefore, a final counting is required to confirm the candidates reminds in F_{prev} (line 9).

The third phase is *structure counting*. We will find out all MIS in D and this step is described in Figure 2. The key steps are in lines 5–7. We again build up the candidates from FE by calling $genCandidates$ iteratively (line 3). The function $prune()$ removes a candidate if anyone of its substructure is found to be infrequent by checking it against the frequent structures from the previous iteration stored in F_{prev} (line 3). Subsequently, for a candidate c, we check if it is a subset of a frequent structure in F_k. This will remove a large number of frequent structures (line 7). For the reminding candidates, we scan D to find their exact support (line 9). If it is not frequent, then it must be an MIS and it will be stored in M (line 13).

```
/* Input  int k, the maximum number of edges in a MIS*/
/* Input  F_k, all size k frequent structures*/
/* Output  M, the set of MIS up to size k*/
1).   F_prev = FE; M = ∅
2).   for i=2 to k
3).       candidates = prune(genCandidate(F_prev, FE))
4).       if candidates == ∅ then break; F_prev = ∅
5).       for each c in candidates
6).           if c is a substructure of an structure in F_k then
7).               candidates.remove(c); insert c into F_prev
8).       for each c in candidates
9).           update sup(c) w.r.t D
10).              if sup(c) ≥ ρ × |D| then
11).                  candidates.remove(c); insert c into F_prev
12).      for each c in candidates
13).          if sup(c) > 0 then insert c into M
14).  return M
```

Fig. 2. The Mining Algorithm – Phase 3

3.3 Optimizations

In this section we describe several optimizations that can improve the mining efficiency.

First Strategy: Use distinct children-sets During the first scan of the dataset, for each element we record every distinct set of children elements found in the database. For example, consider the dataset of Figure 3, consisting of two document trees. Observe that element a has in total three distinct children sets; $((a)(b))$, $((c)(d))$, and $((d)(f))$. The children sets $((a)(b))$, $((c)(d))$ are found in doc1, and the children sets $((a)(b))$, $((d)(f))$ are found in doc2. When an element a having d as child is extended during

candidate generation, we consider only f as a new child for it. This technique greatly reduces the number of candidates, since generation is based on extending the frequent structures by adding edges to their rightmost path.

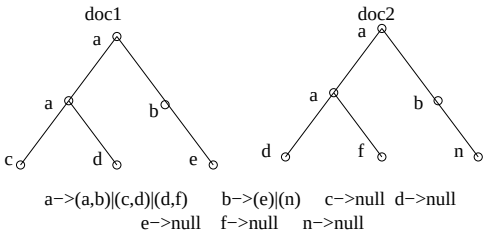

Fig. 3. Summary of Children-sets

Second Strategy: Stop counting early Notice that we are searching for MIS rather than frequent structures, thus we are not interested in the exact support of frequent structures. During the counting process, when the support of a candidate is greater than the threshold, the structure is already frequent. Thus, we do not need to count it anymore; it is immediately removed from $candidates$ and inserted to F_{prev}. This heuristic is implemented in Lines 8–9 in Figure 1 and Lines 10–11 in Figure 2.

Third Strategy: Counting multiple levels of candidates After candidate pruning, if the number of remaining ones is small, we can directly use them to generate the next level candidates and count two levels of candidates with a single scan of the documents. This can reduce the I/O cost at the last phases of the data mining process.

Fourth Strategy: Using the SG–tree in phase-two counting In Figure 1, we obtain the supports of candidates by sequentially comparing their signatures to those of all documents. This operation is the bottleneck of the second phase in our mining algorithm. Instead of comparing each candidate with all document signatures, we can employ an index for document signatures, to efficiently select only those that contain a candidate.

The SG–tree (or *signature* tree) [7] is a dynamic balanced tree similar to R–tree for signatures. Each node of the tree corresponds to a disk page and contains entries of the form $\langle sig, ptr \rangle$. In a leaf node entry, sig is the signature of the document and ptr stores the number of documents sharing this signature. The signature of a directory node entry is the logical OR of all signatures in the node pointed by it and ptr is a pointer to this node. In other words, the signature of each entry *contains* all signatures in the subtree pointed by it. All nodes contain between c and C entries, where C is the maximum capacity and $c \geq C/2$, except from the root which may contain fewer entries. Figure 4 shows an example of a signature tree, which indexes 9 signatures. In this graphical example the maximum node capacity C is three and the length of the signatures six. In practice, C is in the order of several tens and the length of the signatures in the order of several hundreds.

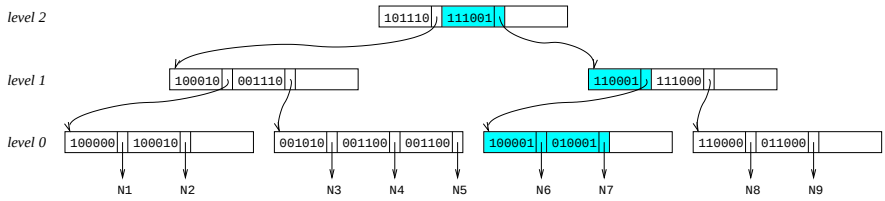

Fig. 4. Example of a Signature Tree

The tree can be used to efficiently find all signatures that contain a specific query signature (in fact, [7] have shown that the tree can also answer *similarity* queries). For instance, if $q = 000001$, the shaded entries of the tree in 4 are the qualifying entries to be followed in order to answer the query. Note that the first entry of the root node does not contain q, thus there could be no signature in the subtree pointed by it that qualifies the query.

In the first phase of our mining algorithm, we construct an SG–tree for the set SG of signatures, using the optimized algorithm of [7] and then use it in the second phase to facilitate counting. Thus, lines 5–6 in Figure 1 are replaced by a depth-first search in the tree for each candidate c, that is to count the number of document signatures containing the signature of c. As soon as this number reaches the threshold $\rho \times |D|$, search stops and c is inserted into F_{prev}. Note that we use a slight modification of the original structure of [7]; together with each signature g in a leaf node entry we store the number of documents $sup(g)$ having this signature (in replacement of the pointer in the original SG–tree).

4 Performance Studies

In this section, we evaluate the effectiveness and efficiency of our methodology using both synthetic and real data. The real data are from the DBLP archive [13]. First, we describe how the synthetic data is generated. Then, we validate the efficiency of our data mining algorithm on both synthetic and real data. All experiments are carried out in a computer with 4 Intel Pentium 3 Xeon 700MHZ processors and 4G memory running Solaris 8 Intel Edition. Java is the programming language.

4.1 Synthetic Data Generation

We generated our synthetic data using the NITF (News Industry Text Format) DTD [14]. Table 1 lists the parameters used at the generation process. First, we parse the DTD and build a graph on the parent-children relationships and other information like the relationships between children. Then, starting from the root element r of the DTD, for each subelement, if it is accompanied by "*" or "+", we decide how many times it should appear according to a Poisson distribution. If it is accompanied by "?", its occurrence is decided by a biased coin. If there are choices among several subelements

of r, then their appearances in the document follow a random distribution. The process is repeated on the newly generated elements until some termination thresholds, such as a maximum document depth, have been reached.

Name	Interpretation	Value
N	total number of docs	10000–100000
W	distribution of '*'	Poisson
P	distribution of '+'	Poisson
Q	probability of '?' to be 1	a number between 0 & 1
Max	distribution of doc depth	Poisson

Table 1. Input Parameters for Data Generation

4.2 Efficiency of the Mining Algorithm

In the first set of experiments, we compare the total running cost (including the I/O time) of our mining algorithm (denoted by MMIS) with the algorithm in [2](denoted by MFS). The efficiency of the two techniques is compared with respect to three sets of parameters: (1) the support thresholds, (2) the maximum size k of the mined MIS, (3) the number of documents.

The synthetic data used in the experiments were generated by setting the parameters of the distribution functions to: W=3, P=3, Q=0.7, Max=10. Except experiments in Figure 7 and 10, N = 10000 in synthetic data set and N = 25000 in real data set.(where documents were randomly picked from the DBLP archive.)

Varying the Support Threshold
Figures 5 and 8 show the time spent by MMIS and MFS on the synthetic and real data respectively. The support threshold varies from 0.005 to 0.1. We set k=10 for the real dataset and k=15 for the synthetic one. Both figures show the same trend: as the support threshold increases, the improvement of MMIS over MFS decreases. This is because the number of frequent structures decreases, which degrades the pruning effectiveness of F_k. The improvement in the DBLP data is smaller than the improvement in the synthetic data, because these documents are more uniform.

Note that in lines 5 – 7 of the algorithm in Figure 2, a large number of candidates would be pruned in the course of finding the MIS. Figure 11 shows for k=15, ρ=0.01, the percentage of candidates in the synthetic data, which are pruned because of this optimization. The amount is in the range of 84% to 97%, which is significant saving.

Varying k
Figures 6 and 9 show the time spent by MMIS and MFS on synthetic and real data respectively, for various values of k and ρ = 0.01. Observe that as k increases, the speedup of MMIS over MFS increases. The reason for this is that once k goes beyond the level at which the number of candidates is the maximum, the number of frequent structures in F_k becomes smaller. Yet, the optimization from the pruning still have noticeable effect on the speed.

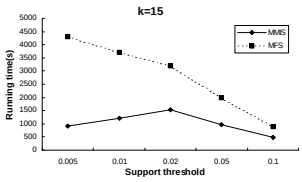

Fig. 5. Varying ρ (synth.)

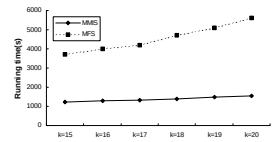

Fig. 6. Varying k (synth.)

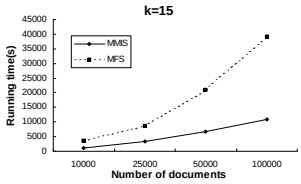

Fig. 7. Scalability (synth.)

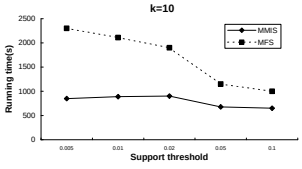

Fig. 8. Varying ρ (real)

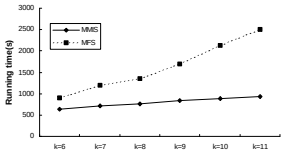

Fig. 9. Varying k (real)

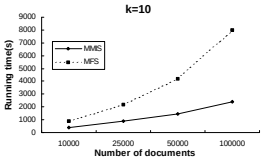

Fig. 10. Scalability (real)

Figures 12 and 13 show the number of MIS discovered for various k on the synthetic and real dataset, respectively. The numbers unveil: First, the total numbers of MIS in both cases are small. Secondly, the number of MIS do not grow much while k increases. Therefore, even if k is very large, the time to mine MIS is still acceptable. (Even if we mine *all* the MIS of any size.) This is also confirmed by the results in Figures 6 and 9.

size of candidates.	8	9	10	11	12	13
% pruned candidates	84	85	90	90	95	97

Fig. 11. Effectiveness in 2nd Phase

k	15	16	17	18	19	20
MIS	321	325	340	342	342	348

Fig. 12. MIS (Synth.)

k	6	7	8	9	10
MIS	69	73	75	78	78

Fig. 13. MIS (Real)

Varying the Number of Documents

Figures 7 and 10 show the time spent by MMIS and MFS on synthetic and real document sets of various cardinalities. For this experiment, k=10 for real data and k=15 for synthetic data, while $\rho = 0.01$ in both cases.

In both cases the speedup of MISS over MFS is maintained with the increase of problem size, showing that MMIS scales well. We observe that for the DBLP data, the speedup actually increases slightly with the problem size. This is due to the fact that DBLP documents have uniform structure and the number of distinct signatures does not increase much by adding more documents.

Mining MIS Using SG–tree

As we have discussed the usage of SG-tree as an optimization techniques, here we show the improvement achieved by SG-tree measured in running time. In the experiments, we compare the total running time (Including the I/O time) of two versions of our mining technique (a) MMIS and (b) MMIS-SG–tree (which is MMIS equipped with SG–tree

in the second phase) with MFS. In all experiments, the first and second optimization strategies discussed in Section 3.3 are applied.

First, we compare the time spent by the three methods for different values of ρ. For small values of $\rho(\leq 2\%)$, the SG–tree provides significant performance gain in mining, which is about 25-30% less time cost, while the impact of using the tree at search degrades as ρ increases. There are two reasons for this: (i) the number of candidates is reduced with ρ, thus fewer structures are applied on it and (ii) the SG–tree is only efficient for highly selective signature containment.

k=15, D=10,000, ρ=	0.005	0.01	0.02	0.05	0.1
MFS	4300	3700	3200	2000	900
MMIS	923	1211	1527	967	494
MMIS-SG–tree	660	870	1180	870	460

Fig. 14. Running Time on Synthetic Data

k=10, D=25000, ρ=	0.005	0.01	0.02	0.05	0.1
MFS	2300	2110	1900	1150	1000
MMIS	850	889	901	679	650
MMIS-SG–tree	600	640	650	610	630

Fig. 15. Running Time on Real Data

Next, we show the time spent by the three methods for different values of D, where ρ=0.01. In Table 16 and 17, again the advantage of using the SG–tree is maintained for small ρ for about 25-30% less time cost.

k=15, ρ=0.01 D=	10000	25000	50000	100000
MFS	3700	8700	21000	39000
MMIS	1211	3300	6610	10910
MMIS-SG–tree	900	2400	4700	7400

Fig. 16. Running Time on Synthetic Data

k=10, ρ=0.01 D=	10000	25000	50000	100000
MFS	900	2200	4200	8000
MMIS	400	889	1460	2400
MMIS-SG–tree	280	630	1080	1750

Fig. 17. Running Time on Synthetic Data

5 Conclusion and Future Work

Besides discovering frequent tree structures in XML documents, mining infrequent tree structures is also important in many XML applications such as query optimization and identification of abnormal cases. In this paper, we initiate the study of mining infrequent tree structures. Since all superstructures of an infrequent tree structure are always infrequent, it does not make sense to find all infrequent tree structures. We introduced the concept of Minimal Infrequent Structures (MIS), which are infrequent structures in XML data, whose substructures all are frequent. Based on MIS, it is easy to construct all infrequent tree structures.

In order to efficiently find all MIS, we developed a data mining algorithm which can be several times faster than previous methods. In addition, our algorithm is independent to the problem of indexing MIS. It can be used for other data mining applications (e.g., discovery of maximum frequent structures).

In the current work, we have focused on the applicability of our techniques in databases that contain a large number of XML trees (i.e., documents). However, our methodology could be adapted for arbitrarily structured queries (e.g., graph-structured queries with wild-cards or relative path expressions), by changing the definitions of the primary structural components (e.g., to consider relative path expressions like $a//b$, instead of plain edges), and the graph matching algorithms. Some preliminary work has been done on considering relative edges in [12], we plan to combine the encoding schema in [12] with our three phase algorithm in our future research.

Another interesting direction for future work is the incremental maintenance of the set of MIS. A preliminary idea towards solving this problem is to change the mining algorithm of Figure 1 to compute the exact counts of frequent structures of size k (instead of stopping as soon as the mininmum support has been reached). Then given a set ΔD of new XML documents, we apply the first and second phases of our algorithm for ΔD, to count the frequencies of all frequent structures of size k there. Having the exact count of frequent structures of size k in the existing and new document sets, we can then directly use the algorithm of [4] to compute the exact count of all frequent structures of size k in the updated document set $D + \Delta D$ and simply apply the third phase of our algorithm to update the MIS set. Integration of value elements within MIS and incrementally updating MIS are two interesting problems for further investigation.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of VLDB Conf.*, 1994.
2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, and H. Sakamoto. Efficient Substructure Discovery from Large Semi-structured Data. *Proc. of the Annual SIAM symposium on Data Mining*, 2002.
3. R. Bayardo. Efficiently Mining Long Patterns from Databases, *Proc. of SIGMOD Conf.*, 1998
4. D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Techniques. *Proc. of ICDE Conf.*, 1996.
5. L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. *Proc. of KDD Conf.*, 1998.
6. M. Kuramochi and G.Karypis. Frequent subgraph discovery. *Proc. of ICDM Conf.*, 2001.
7. N. Mamoulis, D. W. Cheung, and W. Lian. Similarity Search in Sets and Categorical Data Using the Signature Tree. *Proc. of ICDE Conf.*, 2003.
8. H. Toivonen. Sampling large databases for association rules. *Proc of VLDB Conf.*, 1996.
9. S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.
10. K. Wang and H. Liu. Discovering Structural Association of Semistructured Data *IEEE Transactions on Knowledge and Data Engineering*, 12(3):353–371, 2000.
11. L. H. Yang, M. L. Lee and W. Hsu. Efficient Mining of XML Query Patterns for Caching. *Proc. of VLDB Conf.*, 2003.
12. M. J. Zaki. Efficiently Mining Frequent Trees in a Forest. *Proc. of SIGKDD Conf.*, 2002.
13. DBLP XML records. *http://www.acm.org/sigmod/dblp/db/index.html*. Feb. 2001.
14. International Press Telecommunications Council. News Industry Text Format (NITF). *http://www.nift.org*, 2000.