Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining[⋆]

Chen Wang[1] Mingsheng Hong[1] Jian Pei[2] Haofeng Zhou[1] Wei Wang[1] Baile Shi[1]

[1] Fudan University, China
{chenwang, 9924013, haofzhou, weiwang1, bshi}@fudan.edu.cn
[2] State University of New York at Buffalo, USA. jianpei@cse.buffalo.edu

Abstract. Mining frequent tree patterns is an important research problems with broad applications in bioinformatics, digital library, e-commerce, and so on. Previous studies highly suggested that pattern-growth methods are efficient in frequent pattern mining. In this paper, we systematically develop the pattern growth methods for mining frequent tree patterns. Two algorithms, Chopper and XSpanner, are devised. An extensive performance study shows that the two newly developed algorithms outperform TreeMinerV [13], one of the fastest methods proposed before, in mining large databases. Furthermore, algorithm XSpanner is substantially faster than Chopper in many cases.

1 Introduction

Recently, many emerging application domains encounter tremendous demands and challenges of discovering knowledge from complex and semi-structural data. For example, one important application is mining semi-structured data [2, 4, 7, 11–13]. In [12], Wang and Liu adopted an Apriori-based technique to mine frequent path sets in ordered trees. In [7], Miyahara et al. used a directly generate-and-test method to mine tree patterns. Recently, Zaki [13] and Asai et al. [2] proposed more efficient algorithms for frequent subtree discovery in a forest, respectively. They adopted the method of rightmost expansion, that is, their methods add nodes only to the rightmost branch of the tree.

Recently, some interesting approaches for frequent tree pattern mining have been proposed. Two typical examples are reported in [6, 13]. These methods observe the *Appriori* property among the frequent tree sub-patterns: *every non-empty subtree of a frequent tree pattern is also frequent.* Thus, they smartly extend the candidate-generation-and-test approach to tackle the mining. The method for frequent tree pattern mining is efficient and scalable when the patterns are not too complex. Nevertheless, if there are many complex patterns in the data set, there can be a huge number of candidates need to be generated and tested. That may degrade the performance dramatically.

Some previous studies strongly indicate that the depth-first search based, pattern-growth methods, such as FP-growth [5], TreeProjection [1] and H-mine

[⋆] This research is supported in part by the Key Program of National Natural Science Foundation of China (No. 69933010), China National 863 High-Tech Projects (No. 2002AA4Z3430 and 2002AA231041), and US NSF grant IIS-0308001.

[8] for frequent itemset mining, and PrefixSpan [9] for sequential pattern mining, can mine long patterns efficiently from large databases. That stimulates our thinking: "*Can we extend the pattern-growth methods for efficient frequent tree mining?*" This is the motivation of our study.

Is it straightforward to extend the pattern-growth methods to mine tree patterns? Unfortunately, the previously developed pattern-growth methods cannot be extended simply to tackle the frequent tree pattern mining problem efficiently. There are two major obstacles. On the one hand, one major cost in frequent tree pattern mining is to test whether a pattern is a subtree of an instance in the database. New techniques must be developed to make the test efficient. On the other hand, there can be many possible "growing points" (i.e., possible ways to extend an existing pattern to more complex ones) in a tree pattern. It is non-trivial to determine the "good" growth strategy and avoid redundance.

In this paper, we systematically study the problem of frequent tree pattern mining and develop two novel and efficient algorithms, *Chopper* and *XSpanner*, to tackle the problem. In algorithm *Chopper*, the mining of sequential patterns and the extraction of frequent tree patterns are separated as two phases. For each sequential pattern, *Chopper* generates and tests all possible tree patterns against the database. In algorithm *XSpanner*, the mining of sequential patterns and the extraction of frequent tree patterns are integrated. Larger frequent tree patterns are "grown" from smaller ones.

Based on the above ideas, we develop effective optimizations to achieve efficient algorithms. We compare both *Chopper* and *XSpanner* with algorithm *TreeMinerV* [13], one of the best algorithms proposed previously, by an extensive performance study. As an Apriori-based algorithm, *TreeMinerV* achieves the best performance for mining frequent tree patterns among all published methods. Our experimental results show that both *Chopper* and *XSpanner* outperform *TreeMinerV* while the mining results are the same. *XSpanner* is more efficient and more scalable than *Chopper*.

The remainder of this paper is organized as follows. In Section 2, we define the problem of frequent tree pattern mining. Algorithm *Chopper* and *XSpanner* are developed in Section 3 and 4 respectively. In Section 5, we present the results on synthetic and real dataset via comparing with *TreeMinerV*. Section 6 concludes the paper.

2 Problem Definition

A *tree* is an acyclic connected graph. In this paper, we focus on *ordered, labelled, rooted trees.* A tree is denoted as $T(v_0, N, L, E)$, where (1) $v_0 \in N$ is the *root node*; (2) N is the set of *nodes*; (3) L is the set of *labels of nodes*, for any node $u \in N$, $L(u)$ is the label of u; and (4) E is the set of *edges* in the tree. Please note that two nodes in a tree may carry the identical label.

Let v be a node in a tree $T(v_0, N, L, E)$. The *level* of v is defined as the length of the shortest path from v_0 to v. The *height* of the tree is the maximum level of all nodes in the tree. For any nodes u and v in tree $T(v_0)$, if there exists a path v_0-...-u-...-v such that every edge in the path is distinct, then u is called an *ancestor* of v and v a *descendant* of u. Particularly, if (u, v) is an edge in the tree, then u is the *parent* of v and v is a *child* of u. For nodes u, v_1 and v_2, if

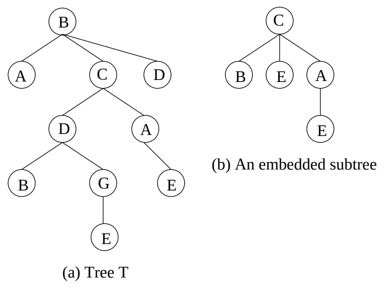
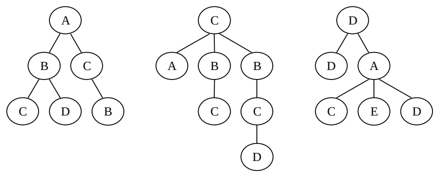


Fig. 1. Tree and embedded subtree.



TID	Tree
1	A1B2C3D3C2B3
2	C1A2B2C3B2C3D4
3	D1D2A2C3E3D3

Fig. 2. An example of tree database TDB.

u is the parent of both v_1 and v_2, then v_1 and v_2 are *siblings*. A node without any child is a *leaf node*, otherwise, it is an *internal node*. In general, an internal node may have multiple children. If for each internal node, all the children are ordered, then the tree is an *ordered tree*. We denote the k-th child of node u as $child^k(u)$. In the case that such a child does not exist, $child^k(u) = null$.

Hereafter, without special mention, all trees are labelled, ordered, and rooted. An example is shown in Figure 1(a). The tree is of height 4.

Given a tree $T(v_0, N, L, E)$, tree $T'(v'_0, N', L', E')$ is called an *embedded subtree* of T, denoted as $T' \sqsubseteq T$, if (1) $N' \sqsubseteq N$; (2) for any node $u \in N'$, $L(u) = L'(u)$; and (3) for every edge $(u, v) \in E'$ such that u is the parent of v, u is an ancestor of v in T. Please note that the concept embedded subtree defined here is different from the conventional one.[3] In Figure 1(b), an embedded subtree of tree T in Figure 1(a) is shown.

A *tree database* is a bag of trees. Given a tree database TDB, the *support* of a tree T is the number of trees in TDB such that T is an embedded subtree, i.e., $sup(T) = \|\{T' \in TDB | T \sqsubseteq T'\}\|$. Given a *minimum support threshold* min_sup, a tree T is called as a *frequent tree pattern* if $sup(T) \geq min\_sup$.

Problem statement. Given a tree database TDB and a minimum support threshold min_sup, The problem of *frequent tree pattern mining* is to find the complete set of frequent tree patterns from database TDB.

In an ordered tree, by a preorder traversal of all nodes in the tree, a *preorder traversal label sequence* (or *l-sequence* in short) can be made. For example, the l-sequence of tree T in Figure 1(a) is $BACDBGEAED$. Preorder traversal sequences are not unique with respect to trees. That is, multiple different trees may result in an identical preorder traversal sequence. To overcome this problem, we can add the levels of nodes into the sequence and make up the *preorder traversal label-level sequence* (or l^2*-sequence* in short). For example, the l^2-sequence of tree T is $B0A1C1D2B3G3E4A2E3D1$. We have the following results.

Theorem 1 (Uniqueness of l^2-sequences). *Given trees $T_1(v_1, N_1, L_1, E_1)$ and $T_2(v_2, N_2, L_2, E_2)$, their l^2-sequences are identical if and only if T_1 and T_2 are isomorphic, i.e., there exists a one-to-one mapping $f : N_1 \rightarrow N_2$ such that*

[3] Conventionally, a tree G' whose graph vertices and graph edges form subsets of the graph vertices and graph edges of a given tree G is called a subtree of G.

(1) $f(v_1) = f(v_2)$*; (2) for every node* $u \in N_1$, $L_1(u) = L_2(f(u))$*; and (3) for every edge* $(u_1, u_2) \in E_1$, $(f(u_1), f(u_2)) \in E_2$.

Lemma 1. *Let* S *be the* l^2*-sequence of tree* $T(v_0, N, L, E)$.

1. *The first node in* S *is* v_0 *whose level number is* 0*;*
2. *For every immediate neighbors* $L(u)iL(v)j$ *in* S, $j \leq (i+1)$*; and*
3. *For nodes* u *and* v *such that* u *is the parent of* v, $L(u)i$ $(i \geq 0)$ *is the nearest left neighbor of* $L(v)j$ *in* S *such that* $j = (i-1)$.

Although an l-sequence is not unique with respect to trees, it can serve as an *isomorph* for multiple trees. In other words, multiple l^2-sequences and thus their corresponding trees can be *isomers* of an l-sequence.

Given a sequence $S = s_1 \cdots s_n$. A sequence $S' = s'_1 \cdots s'_m$ is called a *subsequence* of S and S as a *super sequence* of S', denoted as $S' \sqsubseteq S$ if there exist $1 \leq i_1 < \cdots < i_m \leq n$ such that $s_{i_j} = s'_j$ for $(1 \leq j \leq m)$. Given a bag of sequences SDB, the *support* of S in SDB is number of S's super sequences in SDB, i.e., $sup(S) = \|\{S' \in SDB | S \sqsubseteq S'\}\|$.

Given a tree database TDB, the bag of the l-sequences of the trees in TDB form a sequence database SDB. We have the following interesting result.

Theorem 2. *Given a tree database* TDB*, let* SDB *be the corresponding* l*-sequence database. For any tree pattern* T*, let* $l(T)$ *be the* l*-sequence of* T*. Then,* $sup(T) \leq sup(l(T))$*; and* T *is frequent in* TDB *only if* $l(T)$ *is frequent in* SDB.

Theorem 2 provides an interesting heuristic for mining frequent tree patterns: we can first mine the sequential patterns in the l-sequence database, and then mine tree patterns accordingly. In particular, a sequential pattern in the l-sequence database (with respect to the same support threshold in both tree database and l-sequence database) is called an *l-pattern*.

Given a tree T, not every subsequence of T's l-sequence corresponds to an embedded subtree of T. For example, consider the tree T in Figure 1(a). The l-sequence is $BACDBGEAED$. CBD is a subsequence. However, there exists no an embedded subtree T' in T such that its l-sequence is CBD.

Fortunately, whether a subsequence corresponds to an embedded subtree can be determined easily from an l^2-sequence. Let S be the l^2-sequence of a tree T. For a node $v\ i$ in S, the *scope* of v is the longest subsequence S' starting from v such that the label-level of each node in S', except for v itself, is greater than i. For example, the l^2 sequence of tree T in Figure 1(a) is $B0A1C1D2B3G3E4A2E3D1$. The scope of $B0$ is $B0A1C1D2B3G3E4A2E3D1$ and the scope of $D2$ is $B3G3E4$. We have the following result.

Lemma 2. *Given a tree* T*. An* l*-sequence* $S = v_1 \ldots v_k$ *corresponds to an embedded subtree in* T *if and only if there exists a node* v_1 *in the* l^2*-sequence of* T *such that* $v_2 \ldots v_k$ *is a subsequence in the scope of a node* v_1.

3 Algorithm Chopper

Algorithm *Chopper* is shown in Figure 3. The correctness of the algorithm follows Theorem 2. In the first 2 steps, *Chopper* finds the sequential patterns (i.e., l-patterns) from the l-sequence database. Based on Theorem 2, we only need to

```
Input:    a tree database $TDB$ and a support threshold $min\_sup$;
Output:   all frequent tree patterns with respect to $min\_sup$;
Method:
(1)   scan database $TDB$ once, generate its $l$-sequence database $SDB$;
(2)   mine $l$-patterns from $SDB$ using algorithm $l$-PrefixSpan;
(3)   scan $TDB$, generate candidates according to $l$-patterns and find frequent tree;
```

Fig. 3. Algorithm Chopper

consider the trees whose l-sequence is an l-pattern. In the last step, *Chopper* scans the database to generate candidate tree patterns and verify the frequent tree patterns.

To make the implementation of *Chopper* as efficient as possible, several techniques are developed. The first step of *Chopper* is straightforward. Since the trees are stored as l^2-sequences in the database, *Chopper* does not need to form the explicit l-sequence database SDB. Instead, it uses the l^2-sequence database and just ignores the level numbers.

In the second step of *Chopper*, we need to mine sequential patterns (i.e., l-patterns) from the l-sequence database. A revision of algorithm PrefixSpan [9], called l-PrefixSpan, is used. Some specific techniques have been developed to enable efficient implementation of PrefixSpan. Interested readers should refer to [9] for a detailed technical discussion.

While PrefixSpan can find the sequential patterns, the tree pattern mining needs only part of the complete set as l-patterns. One key observation here is that *only those l-patterns having a potential frequent embedded tree pattern should be generated.* The idea is illustrated in the following example.

Example 1. Figure 2 shows a tree database as the running example in this paper. Suppose that the minimum support threshold is 2, i.e., every embedded subtree is a frequent tree pattern if it appears at least in two trees in the database.

The l^2-sequences of the trees are also shown in Figure 2. If we ignore the level numbers in the l^2-sequences, we get the l-sequences. Clearly, sequence $\langle BCB \rangle$ appears twice in the l-sequence database, and thus it is considered as a sequential pattern by PrefixSpan. However, we cannot derive any embedded tree in the database whose l-sequence is BCB. Thus, such patterns should not be generated.

We revise PrefixSpan to l-PrefixSpan to mine only the promising l-patterns and prune the unpromising ones. In l-PrefixSpan, when counting $sup(S)$ for a possible sequential pattern S from the (projected) databases, we count only those trees containing an embedded subtree whose l-sequence is S. Following Lemma 2, we can determine whether an l-sequence corresponds to an embedded subtree in a tree easily. For example, $sup(BCB) = 0$ and $sup(ABC) = 1$. Thus, both BCB and ABC will be pruned in l-PrefixSpan.

It can be verified that many patterns returned by PrefixSpan, such as AB, ABC, $ABCD$, etc., will be pruned in l-PrefixSpan. In our running example, only 9 l-patterns are returned, i.e., A, AC, AD, B, BC, BCD, BC, C and D.

In the last step of *Chopper*, a naïve implementation would be as follows. We can generate all possible tree patterns as candidates according to the l-patterns

Input and output: same as Chopper;
Method:
```
(1)   scan database $TDB$ once, find frequent length-1 $l$-patterns;
(2)   for each length-1 $l$-pattern $x_i$, do
(3)       output a frequent tree pattern $x_i0$;
(4)       form the $\langle x_i0\rangle$-projected database $TDB_{x_i0}$;
(5)       if $TDB_{x_i0}$ has at least $min\_sup$ trees then mine the projected database;
```

Fig. 4. Algorithm XSpanner

found by l-PrefixSpan, and then scan the tree database once to verify them. Unfortunately, such a naïve method does not scale well for large databases: There can be a huge number of candidate tree patterns!

Chopper adopts a more elegant approach here. It scans the tree database against the l-patterns found by l-PrefixSpan. For each tree in the tree database, *Chopper* firstly verifies whether the tree contains some candidate tree patterns. If so, the counters of the tree patterns will be incremented by one. Then, *Chopper* also verifies whether more tree candidate patterns corresponding to some l-patterns can be generated from the current tree. If so, they will be generated and counters will be set up with an initial value 1. Moreover, to facilitate the matching between a tree in the tree database and the l-patterns as well as candidate tree patterns, all l-patterns and candidate tree patterns are indexed by their prefixes. Please note that a tree is stored as an l^2-sequence in the database.

The major cost of *Chopper* comes from two parts. On the one hand, l-PrefixSpan mines the l-patterns. The pruning of unpromising l-patterns improves the performance of the sequential pattern mining here. On the other hand, *Chopper* has to check every tree against the l-patterns and the candidate tree patterns in the last step. In this step, only one scan needed.

Although l-PrefixSpan can prune many unpromising l-patterns, some unpromising l-patterns still may survive from the pruning, such as BCD in our running example. The reason is that the l-pattern mining process and the tree pattern verification process are separated in *Chopper*. Such unpromising l-patterns may bring unnecessary overhead to the mining. *Can we integrate the l-pattern mining process into the tree pattern verification process so that the efficiency can be improved further?* This observation motivates our design of *XSpanner*.

4 Algorithm XSpanner

Algorithm *XSpanner* is shown in Figure 4. Clearly, by scanning the tree database TDB only once, we can find the frequent items in the database, i.e., the items appearing in at least min_sup trees in the database. They are length-1 l-patterns. This is done in Step (1) in algorithm XSpanner.

Suppose that $x_1, \ldots, x_m$ are the m length-1 l-patterns found. We have the following two claims. On the one hand, for $(1 \leq i \leq m)$, x_i0 is a frequent tree pattern in the database. On the other hand, the complete set of frequent tree patterns can be divided into m exclusive subsets: the i-th subset contains the frequent tree patterns having x_i as the root.

Example 2. Let us mine the frequent tree patterns in the tree database TDB in Figure 2. By scanning TDB once, we can find 4 length-1 l-patterns, i.e., A, B, C and D. Clearly, $A0$, $B0$, $C0$ and $D0$ are the four frequent tree patterns. On the other hand, the complete set of tree patterns in the database can be divided into 4 exclusive subsets: the ones having A, B, C and D as the root, respectively.

The remainder of *XSpanner* (line (2)-(5)) mines the subsets one by one.

For each length-1 l-pattern x_i, *XSpanner* first outputs a frequent tree pattern x_i0 (line (3)). Clearly, to find frequent tree patterns having x_i0 as a root, we need only the trees containing x_i. Moreover, for each tree in the tree database containing x_i, we need only the subtrees having x_i as a root. We collect such subtrees as the $\langle x_i0\rangle$*-projected database*. This is done in line (4) in the algorithm.

In implementation, constructing a physical projected database can be expensive in both time and space. Instead, we apply the *pseudo-projection techniques* [9, 8]. The idea is that, instead of physically constructing a copy of the subtrees, we reuse the trees in the original tree database. For each node, the tree id and a hyperlink are attached. By linking the nodes labelled x_i together using the hyperlinks, we can easily get the $\langle x_i0\rangle$-projected database. Please note that such hyperlinks can be reused latter to construct other projected databases.

Example 3. Figure 5 shows the $B0$-projected database. Basically, all the subtrees rooted at a node B are linked, except for the leaf nodes labelled B. The leaf node labelled B in the left tree is not linked.

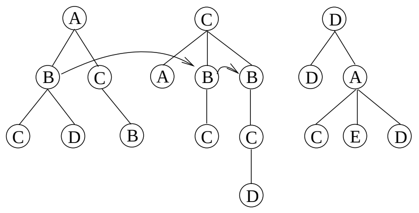


Fig. 5. The $B0$-projected database

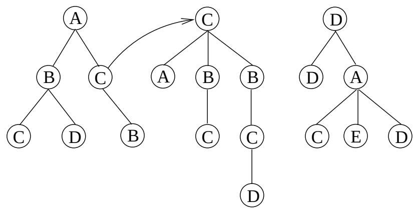
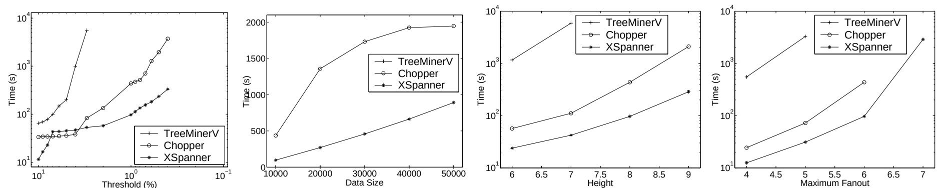


Fig. 6. The $C0$-projected database

Why the leaf nodes should not be linked? The leaf node cannot contain any embedded subtree larger than the tree pattern we have got so far. In other words, not linking such leaf nodes will not affect the result of the mining. Thus, it is safe to prune them in the projected database.

Please note that there can be more than one node with the same label in a tree, such as the tree at the middle of Figure 5. These subtrees are processed as follows: a node labelled with B is linked only if it is not a descendant of some other node that is also labelled with B. As another example, Figure 6 shows the $C0$-projected database. In the tree at the middle, only the root node is linked.

As shown in Figure 5, more than one subtree from a tree in the tree database may be included in the projected database. When counting the support of a tree pattern, such as BC, the pattern should gain support 1 from the same original tree, even it may be matched in multiple subtrees.



(a) Sup vs. time (b) Size vs. time (c) Height vs. time (d) Fanout vs. time

Fig. 7. Results on synthetic data sets

The projected databases can be mined recursively. In the $\langle x_i 0\rangle$-projected database, the length-1 l-patterns should be found. For each length-1 l-pattern x_j, $x_i 0 x_j 1$ is a frequent tree pattern. The set of frequent tree patterns having x_i as a root can be divided into smaller subsets according to x_j's.

In general, suppose that P is a frequent tree pattern and P-projected database is formed. Let S be the l-pattern of P. By scanning the P-projected database once, *XSpanner* finds frequent items in the projected database. Then, for each frequent item x_i, *XSpanner* checks whether Sx_i is an l-pattern in the P-projected database and there exists a frequent tree pattern corresponds to Sx_i. If so, then the new frequent tree pattern is output and the recursive mining continues. Otherwise, the search in the branch is terminated.

The correctness of the algorithm can be proved. Limited by space, we omit the details here.

5 Experiments and Performance Study

In this section, we will evaluate the performance of *XSpanner* and *Chopper* in comparison with *TreeMinerV* [13]. All the experiments are performed on a Pentium IV 1.7GHz PC machine with 512MB RAM. The OS platform is Linux Red Hat 9.0 and the algorithms are implemented in C++.

We wrote a synthetic data generation program to output all the test data. There are 8 parameters for adjustment: the number of the labels $|S|$, the probability threshold of one node in the tree to generate children or not p, the number of the basic pattern trees (BPT) $|L|$, the average height of the BPT $|I|$, the maximum fanout(children) of nodes in the BPT $|C|$, the data size of synthetic trees $|N|$, the average height of synthetic trees $|H|$ and the maximum fanout of nodes $|F|$ in synthetic trees. The actual height of each (basic pattern) tree is determined by the Gaussian distribution having the average of $|H|(|I|)$ and the standard deviation of 1.

At first, we consider the scalability with *minsup* of the three algorithms, while other parameters are:$S = 100, p = 0.5, L = 10, I = 4, C = 3, N = 10000, H = 8, F = 6$. Figure 7(a) shows the result, where the *minsup* is set from 0.1 to 0.004. In this figure, both X and Y axes have been processed by $\log_{10} T$ for the convenience of observation.

From the figure 7(a), we can find that when support threshold is larger than 5%, the three algorithms perform approximately the same. With the threshold becoming smaller, the algorithm *XSpanner* and *Chopper* begin to outperform *TreeMinerV*. What should be explained here is the reason when the threshold changes from 2% to 1%, the running time went up suddenly. We choose 10000 trees as our test dataset; however, there are only 100 node labels. So much more frequent structures are generated, which leads to the greater time consuming. During this period, we can find that *TreeMinerV* runs out of memory, while *XSpanner* and *Chopper* retain the ability to finish computing. This is because of the large amount of candidates generated by Apriori-based algorithms. It is also clear that when the threshold continues to decrease, *XSpanner* surpasses *Chopper* in performance. *XSpanner* estimates and differentiates between the isomers during the generation of frequent sequences, which can reduce a large number of frequent sequences but infrequent substructures generated when the threshold is low.

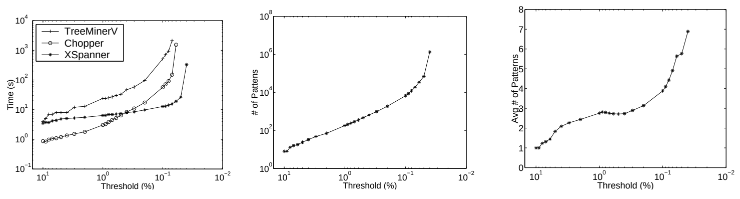
Then, figures 7((b) shows the scalability with data size. The data size N varies from 10000 to 50000, while other parameters are:$S = 100, p = 0.5, L = 10, I = 4, C = 3, H = 8, F = 6, minsup = 0.01$. Here we find the cost of both time and space of *XSpanner* and *Chopper* is extremely smaller than that of *TreeMinerV* which is halted for memory overflow. The reason is that *XSpanner* and *Chopper* can save time and space cost by avoiding false candidate subtree generation.

Finally, the scalability with tree size is shown in Figures 7(c) and (d). The Tree size of H and F varies, while other parameters are:$S = 100, p = 0.5, L = 10, I = 4, C = 3, N = 10000, minsup = 0.01$. In figure 7(c), we only vary H from 6 to 9. It is easy to find that, when H equals to 6 or 7, the performance of *XSpanner* and *Chopper* is better than that of *TreeMinerV*. However, when the trees become higher, the superiority grows. In particular, when H equals to 8 or 9, the two algorithms thoroughly defeat *TreeMinerV* for the reason that *TreeMinerV* is halted for memory overflow. In figure 7(d), the performance of the two algorithms and *TreeMinerV* is similar to the case above. *XSpanner* and *Chopper* performs better than *TreeMinerV*, while the fanout continues to increase.

From all of the experiments we did, we can conclude that it is an acceptable and efficient way to put the process of distinguishing isomers in the process of generating frequent sequences.

We also tested *XSpanner* and *Chopper* in Web Usage Mining. We downloaded the Weblog of Hyperreal (http://music.hyperreal.org), chose those dating from Sep. 10 to Oct.9 in 1998 as the input data. and then transformed the Weblog into tree-like data set which includes 12000 more records totally.

Figure 8(a) shows the performance of the three algorithms, where the $minsup$ is set from 0.1 to 0.0003. In this figure, both X and Y axes have been processed by $\log_{10} T$ for the convenience of observation. We can find, that the performance of *XSpanner* and *Chopper* is better than that of *TreeMinerV*. Especially, *TreeMinerV* is halted in 3 hours for memory overflow when $minsup = 0.0006$, while the two algorithms go well. From the figure, we notice that the performance of *XSpanner* is more stable than that of *Chopper*. With threshold decreasing, *XSpanner* surpasses *Chopper* gradually. It should also be noted that, *XSpanner* does not perform excellently until $minsup$ is dropped to 0.0003.



(a) Sup vs. time (b) Sup vs. # patterns (c) Sup vs. avg # nodes

Fig. 8. Results on real data sets

Finally, figure 8(b) shows number of frequent patterns generated by the algorithm, while figure 8(c) shows average number of nodes of frequent patterns generated by the algorithm, where the *minsup* is set from 0.1 to 0.0003.

6 Conclusions

In this paper, we present two pattern-growth algorithms, *Chopper* and *XSpanner*, for mining frequent tree patterns. In the future, we would like to explore pattern-growth mining of other complex patterns, such as frequent graph patterns.

References

1. R. C. Agarwal, et al. A tree projection algorithm for generation of frequent item sets. *J. of Parallel and Distributed Computing*, 61(3):350–371, 2001.
2. T. Asai, et al. Efficient substructure discovery from large semi-structured data. In *Proc. 2002 SIAM Int. Conf. Data Mining*, Arlington, VA.
3. D. Cook and L. Holder. Substructure discovery using minimal description length and background knowledge. *J. of Artificial Intelligence Research*, 1:231–255, 1994.
4. L. Dehaspe, et al. Finding frequent substructures in chemical compounds. In *KDD'98*, New York, NY.
5. J. Han, et al. Mining frequent patterns without candidate generation. In *SIGMOD'00*, Dallas, TX.
6. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM'01*, San Jose, CA.
7. T. Miyahara, et al. Discovery of frequent tree structured patterns in semistructured web documents. In *PAKDD'01*, Hong Kong, China.
8. J. Pei, et al. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM'01*, San Jose, CA.
9. J. Pei, et al. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01*, Heidelberg, Germany.
10. R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB'95*, Zurich, Switzerland.
11. J.T.L. Wang, et al. Automated discovery of active motifs in multiple RNA secondary structures. In *KDD'96*, Portland, Oregon.
12. K. Wang and H. Liu. Schema discovery for semistructured data. In *KDD'97*, Newport Beach, CA.
13. M.J. Zaki. Efficiently mining frequent trees in a forest. In *KDD'02*, Edmonton, Alberta, Canada.