May 2004, Vol.19, No.3, pp.309-319 J. Comput. Sci. & Technol.

Chopper: Efficient Algorithm for Tree Mining

Chen Wang, Ming-Sheng Hong, Wei Wang, and Bai-Le Shi

Department of Computing and Information Technology, Fudan University, Shanghai 200433, P.R. China

E-mail: chenwang@fudan.edu.cn

Received March 25, 2003; revised August 11, 2003.

Abstract With the development of Internet, frequent pattern mining has been extended to more complex patterns like tree mining and graph mining. Such applications arise in complex domains like bioinformatics, web mining, etc. In this paper, we present a novel algorithm, named *Chopper*, to discover frequent subtrees from ordered labeled trees. An extensive performance study shows that the newly developed algorithm outperforms *TreeMinerV*, one of the fastest methods proposed previously, in mining large databases. At the end of this paper, the potential improvement of *Chopper* is mentioned.

Keywords data mining, semi-structured data, labeled ordered tree

1 Introduction

Mining the frequent pattern from data set is a key progress in data mining research. Previously, most of the efforts are focused on the independent data such as the items in the marketing basket. However, objects in the real world often have close relationship with each other. For example, given a set of transactions, where a transaction records the items brought by a customer at a time, a set of items is called a frequent pattern if it is contained in at least *min_sup* transactions, where *min_sup* is a user-specified minimum support threshold. How to gain the frequent pattern from these relations becomes the objective of the research in recent years. We call this Frequent Structure Mining (FSM)[1].

Among exploratory mining methods, an important problem is that of Association Rule Mining[2], which aims to find the item sets that frequently occur in many database transactions. Another important method is Sequence Mining[3], whose task is to discover a sequence of item sets shared across time among a large number of objects in a given database. These mining tasks can be placed within a generic framework, which we term Frequent Pattern Mining (FPM).

An interesting case of tree structure mining is web usage mining[4,5]. Suppose that a database records the web log information of the users' access patterns, we can analyze it with various mining approaches among which the simplest one is to ignore all linkage information and regard users' access to each page as independent. In this way, the mining results are the page sets that are frequently visited by users. However, the order of the pages visited by users also contains much information. Thus, if we model one access pattern of a user as an ordered labeled tree (ignoring back edges) and analyze such tree sets, we will eventually be able to obtain the paths users visit frequently. This information is certainly helpful to the reconstruction of the web site.

Another typical example is the semi-structured document mining. In recent years, XML has become a popular way of storing datasets because the semi-structured nature of XML facilitates the storage and exchange of information between different databases. Tree-structured XML documents are the most widely used in applications. Given a set of such XML documents, one would like to extract all frequent subtrees that appear in the collection.

Utilizing Association Rule Mining and Sequence Mining, we can obtain useful information from such semi-structured document as XML, web log and so on. At present, some work has been done on how to extract patterns from tree-like data. Distinguishing isomorphism is the vital problem they have to face. Although most methods have made a lot of efforts to handle this problem, the result is still not satisfying. One major cost in frequent tree pattern mining is to test whether a pattern is a subtree of an instance in the database.

Here, we present a new idea to deal with isomorphism problem and develop an algorithm to solve the problem efficiently.

*Correspondence

This paper is supported by the Key Program of National Natural Science Foundation of China (Grant No.69933010) and the National High-Tech Development 863 Program of China (Grant Nos.2002AA4Z3430 and 2002AA231041).

1.1 Related Work

Currently, some researches have been done on mining semi-structured data[1,6−11], among which [1, 9, 10] are focused on the problem of mining frequent subtrees. The SUBDUE system[12] discovers graph patterns using MDL principle. However, it may miss some significant patterns, since it performs a greedy search. Wang and Liu[11] adopted Apriori-style technique to mine frequent path sets in ordered trees. Miyahara *et al.*[10] tried a direct generate-test method. Dehaspe *et al.*[7] developed an algorithm WARMA to mine frequent patterns in graph structures. Since a tree is a special case of graph structures, we can apply WARMA to frequent tree mining. However, the problem of solving isomorphism in graphs is certainly more complex than that in trees. Therefore, this complicated algorithm is not appropriate to be applied to the field of tree mining. Besides, most of those methods are based on Apriori algorithm, in which the generation of a large number of candidate item sets is the bottleneck.

Zaki[1] and Asai *et al.*[6] proposed efficient algorithms for frequent subtree discovery in a forest on SIGKDD2002 and SDM02 respectively. They adopted the method of rightmost expansion to add nodes only to the rightmost branch of the tree. Zaki utilized the sequence and scope in the *TreeMinerH* and *TreeMinerV* algorithms, while Asai *et al.* proposed the FREQT algorithms. These algorithms are similar to our work to some degree, but the basic idea of our work is different from that of theirs. They smartly extend the candidate-generation-and-test approach to tackle the mining, while we employ the depth-first search based, pattern-growth methods. For another thing, they check subtree isomorphism at the beginning of the algorithm, while our algorithm starts this job later in the process, which essentially lead to the difference in mining efficiency.

1.2 Our Contribution

With dimensions of tree-like data increasing, we must spend a lot of time and spaces to check subtree isomorphism. At present, most algorithms solve the isomorphism problem at the beginning. Conversely, our algorithm deals with the problem later. Consequently, we can use the technique to minimize the number of structures for testing and reduce the efforts of solving the problem.

We put forward the idea of "distinguishing isomorphs first and then processing their isomers", which can minimize the search space and cut useless branches out as early as possible. This algorithm improves the efficiency. Therefore, we advance *Chopper* algorithm whose basic idea is to discover all frequent isomorphs from sequences and then obtain their isomers from those frequent isomorphs.

Chopper algorithm is composed of two steps, one utilizes the improved *PrefixSpan*[13] method to discover all frequent sequence retaining the relationship of ancestor-descendant between nodes in trees, the other finds frequent structures from the sequences being generated in the early course.

After forming the basic algorithm of *Chopper*, we optimize some places. For example, traditional *PrefixSpan* algorithm mines frequent sequences which do not regard the relationship of ancestor-descendant between nodes in trees. Aiming at tree-like data, we can add to PrefixSpan algorithm such function as checking the relationship of ancestor-descendant between nodes in trees and then promote its ability to mine special frequent sequences. At this rate, the algorithm will greatly reduce the number of frequent sequences to be generated, also lighten the burden of distinguishing isomorphism problem, and promote the efficiency of algorithm consequently.

Subsequently, we compare *Chopper* algorithm with Zaki's *TreeMinerV* algorithm by experiment. *TreeMinerV* algorithm adopts the method of rightmost expansion to add nodes only to the rightmost branch of the tree. So it is efficient and scalable when the patterns are not very complex and large. Nevertheless, *TreeMinerV* algorithm would generate a lot of candidate item sets and result in bottleneck of performance. Therefore, *Chopper* algorithm manifests better capability than *TreeMinerV*.

The rest of this paper is organized as follows. In Section 2, we define the basic concepts of the frequent subtree mining and state our problem formally. In Section 3, we introduce *Chopper* algorithm, based on idea of isomorphism, in detail and illustrate the process. In Section 4, we offer experimental results on synthetic and real datasets to evaluate the proposed mining algorithm. In Section 5, we conclude the paper.

2 Preliminary Statement

2.1 Terms and Concepts

A *tree* is an acyclic connected graph. A *forest* is a collection of trees, where each tree is a con-

nected component. A forest can thus be viewed as an acyclic graph. In general, every tree T has one and only one vertex v_0 as its "*root*". We can use this vertex to represent the tree, denoted as $T(v_0)$. Then we call this tree *rooted tree*. In this paper, we focus only on rooted tree. Given a rooted tree $T(v_0)$, the *level of node* v is defined as the length of the path from v_0 to v. The *height* of a tree is the maximum level of all nodes in a tree.

Definition 1 (Relationship Between Nodes). *Given a rooted tree $T(v_0)$, consider any path starting from v_0. If node u precedes node v, u is called an ancestor of v, and v a descendant of u. If there is only one edge between u and v, u is called the parent of v, and v the child of u. This edge is called a branch, denoted as $b = (u; v)$. If several nodes share the same parent, they are called siblings.*

We denote the term of relationship between ancestor u and its descendant v as $\mathcal{AD}(u; v)$, or $\mathcal{AD}$ for short, and the term of relationship between parent u and its child v as $\mathcal{PC}(u; v)$ or $\mathcal{PC}$. It is obvious that $\mathcal{PC}$ is a special case of $\mathcal{AD}$.

Therefore, we can denote a rooted tree as $T(v_0) = (N; B)$, in which v_0 is the root, N is the collection of nodes in the tree, and B is the collection of branches in the tree. An ordered tree is a rooted tree in which the children of each node are ordered, i.e., if a node has k children, then we can designate them as the first child, second child, and so on.

Definition 2 (Labeled Tree). *Given a rooted tree $T(v_0)$, a label set L, and a collection of all nodes N, T is a labeled tree iff there exists a mapping f: $N \to L$, that $v \in N$; $f(v) = I \in L$. The tree is then denoted as $T(v_0) = (L(N); B)$.*

It becomes evident from the statement above that different nodes can have the same label, i.e., nodes in a labeled tree may not be labeled uniquely.

Definition 3 (Ordered Labeled Tree). *Given a tree $T(v_0)$, T is an ordered labeled tree iff T is ordered and labeled.*

Definition 4 (Database of Ordered Labeled Tree). *Let TDB denote a database of ordered labeled trees, in which each tuple exists in the format of $\langle TID; T_i \rangle$ is the label of tree-like data. T_i is the ordered labeled tree.*

Definition 5 (Subtree). *Given an ordered labeled tree $T(v_0) = (L(N); B)$, a tree T' is a subtree of $T(v_0)$ iff*

1) *T' is an ordered labeled tree $T'(v') = (L(N'); B')$ with v' as its root;*
2) *node set N' is a subset of N;*
3) *for $b = (v_i; v_j) \in B'$, v_i is v'_js ancestor in T;*
4) *the labels of v_i and v_j correspond with their labels in T.*

Please note that the concept subtree defined here is different from the conventional one. [①] In Fig.1(b), a subtree of tree T in Fig.1(a) is shown.

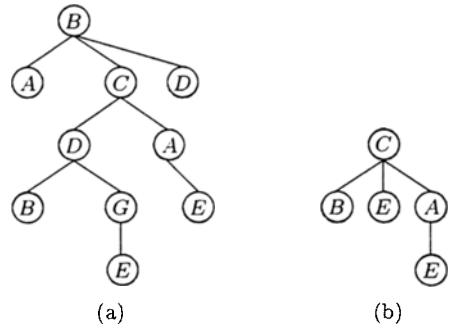

Fig.1. Examples of ordered, labelled, rooted tree and subtree. (a) Tree T. (b) Embedded subtree.

2.2 Problem Statement

Definition 6 (Frequent Subtree). *Given a database of ordered labeled trees TDB and a subtree T, the support value of T is defined as $S(T) = \frac{|p(T)|}{N}$, where $p(T)$ is the number of all trees that contain subtree T, and N is the number of trees in TDB. Hence, a structure T is a frequent subtree iff $s(T) \geqslant$ minsup. Here, minsup is a user-specified threshold, called minimum support.*

Given a database of ordered labeled trees TDB and a minimum support minsup, the problem we want to solve is to find all frequent subtrees.

As mentioned above, most of the existing methods are Apriori-based. However, these methods cannot avoid candidate generation, which greatly consumes the system resource. In this paper, we propose a method based on the depth-first-search-based, pattern-growth ideas. It reduces the cost of candidate generation dramatically. In addition, tree isomorphism is a subtle problem to handle. Though in recent work, the cost of it can be limited within $O(n \log^3 n)$, it still deteriorates the performance besides the huge cost of candidate generation. We develop some techniques to reduce this cost.

[①] 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.

3 Algorithm Chopper

In the first part of this section, we will describe the method of frequent pattern mining as a whole. Then the main steps of the algorithm are stated in detail.

3.1 Basic Idea of the Algorithm

In this section, we propose a general idea to solve the problem of searching for frequent subtrees. Some properties of an ordered labeled tree will be brought out as follows. We choose pre-order sequence as the basis of describing an ordered labeled tree, which cannot represent a tree distinctly. Therefore, we have to remember the level number of each element of the sequence in the tree. Thus, we can describe a tree uniquely with the combination of pre-order sequence and level sequence. For instance, $B1A2C3D3E2F3G3G2D3$ is used to represent the tree in Fig.2.

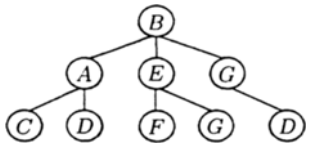

Fig.2. Ordered labeled tree.

In this paper, we can deal with the above sequence instead of a tree.

Property 1. *Given a node X_n in a tree of TDB, where X is the label of this node and n is its level number. The parent node of X_n in the tree is the nearest node ahead of it which has a level number $n-1$.*

This conclusion can be made easily from the pre-order sequence.

Property 2. *The sequence representing an ordered labeled tree with level number has the following properties:*

1) *A node's level number is either just bigger than the level number of the adjacent node ahead by one, or no more than the level number of the adjacent node ahead;*

2) *The first node of the sequence has the smallest level number, which is the only one in the sequence.*

As we discuss above, using only pre-order sequence without level number it is impossible to determine an ordered labeled tree. Based on this fact, we propose the concepts *isomorph* and *isomer*. The term of isomorph comes from chemistry, which means that two kinds of materials have the same components and the different structures. For example, diamond and graphite are both denoted by the chemical symbol C, which represents the carbon atoms, but they differ from each other, especially in the physical features. The reason is, although they are both composed of carbon elements, their structures among the carbon atoms are quite different. So, we call the symbol C the isomorph of diamond and graphite, and the diamond and graphite are the isomers of the isomorph C. Therefore, we introduce the concept of isomorphism to our algorithm, which can explain our problem vividly. In our paper, pre-order sequence can be viewed as isomorphs, and the sequence combined with level number is its isomer. As shown in Fig.3, the sequence of $ABCD$ is the isomorph, and $A1B2C3D4$, $A1B2C3D2$ and $A1B2C2D3$ are isomers.

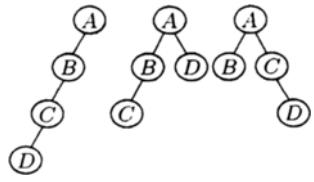

Fig.3. Isomorph and isomers.

In the following part of this section, we focus on the efficiency improvement of the ordered-labeled tree mining. We put forward the important theorem below based on the concept of isomorphism and frequent structure.

Theorem 1. *If the isomorph expression is not frequent, the corresponding isomer is infrequent.*

Proof. Assuming that there is an infrequent isomorph expression with a frequent isomer structure, the structure would exist in some original trees and the number of these trees would be larger than the minimum support. For the isomorph expressions of these trees, that infrequent expression must be their subset, otherwise the isomer structure will not exist in the trees. Therefore, the number of the isomorph expression in the trees is greater than the minimum support. It is a contradiction to the assumption above. □

From this theorem, we can make a conclusion that if the isomorph expression is infrequent, then the isomer structure is infrequent too, which is very important to the problem of searching for frequent subtree. We can first deal with the isomorph expression. If it is not frequent, the work of its isomer structure will be omitted, which will save us much time and work. What is more, when the isomorph expression is very long with a lot of isomer structures, the improvement of efficiency is remarkable.

For graphs, the problem of searching for isomorphism is known to be an NP-hard problem[14]. This problem also exists in trees[15], and we will solve it later in our algorithm by greatly reducing the number of structures to be tested. In our algorithm, we first check the expressions, and then filter out the infrequent ones, which will save the space when searching for the isomer, especially when the expression is long.

Based on the concept of "isomorph first, isomer later", *Chopper* algorithm first obtain isomorph expressions by using some method of searching for frequent sequence, and then process them to find isomer structures. When we search for frequent sequences, we do not need to know their structures, thus the process is efficient.

Chopper algorithm consists of the following two steps. Firstly, it searches for frequent sequence, and secondly it processes the sequence obtained above to find isomer structure. The algorithm is shown in Fig.4. The first step deals with scanning ordered labeled trees. By using some mining sequence method, we could find all frequent sequences. The second step is to scan TDB once again and construct all frequent structures according to each frequent sequence. After finishing the scan, all those with support greater than the minimum support will be output.

```
Algorithm 1. Chopper
INPUT: tree dataset TDB, support threshold minsup
OUTPUT: a set of frequent subtree
Chopper (TDB, minsup)
{
  ReadFile (TDB); //read TDB
  FindSequence (TDB, minsup, FSDB);
  InitialFrequentTree(ft,FSDB);
  //find isomers in FSDB and store them
  outputFSP=Process(ft,TDB);
  For each isomer i in outputFSP
    if (isomer i is no less than minsup)
      output(isomer i);
}
```

Fig.4. Algorithm of Chopper.

Most methods of mining frequent structures also consider isomorphism problem, such as [1]. But they consider it at the beginning of their algorithms, so they have to face a lot of work at first. In fact, a great part of this work is unnecessary, because a lot of sequences do not keep $\mathcal{AD}$ in trees. In this situation, we do not consider isomer structure in the first step. We only find frequent sequence and prune unnecessary sequences as much as we can. Thus the number of structures to be tested is minimized.

We will discuss these two steps in detail in Subsections 3.2 and 3.3, respectively.

3.2 Finding Frequent Sequences Keeping $\mathcal{AD}$

There are many methods of searching for frequent sequence, most of which are based on the concept of Apriori. Most cost of these algorithms is concentrated on the candidate generation and processing. PrefixSpan[13] is another method, which can reduce a lot generation of candidate, and is more efficient than Apriori. So we adopt PrefiSpan as the base of *Chopper* algorithm.

PrefixSpan is an efficient algorithm mining sequential patterns from sequence databases. The general idea of PrefixSpan is as follows. Given a sequence database *SDB* and a minimum support threshold *min_sup*. To facilitate the elaboration of ideas, we assume that each sequence is a string, i.e., any element of a sequence contains only one item. PrefixSpan firstly scans *SDB* to find frequent items as length-1 sequential patterns. Suppose that length-1 patterns $\langle x_1\rangle$, ..., $\langle x_n\rangle$ are found. Then, the complete set of sequential patterns can be divided into n distinct subsets: the i-th subset ($1 \leqslant i \leqslant n$) contains the sequential patterns with prefix $\langle x_i\rangle$.

To mine the sequential patterns with prefix $\langle x_i\rangle$, the $\langle x_i\rangle$-projected database is formed, which contains the sequences having x_i. For each sequence in the $\langle x_i\rangle$-projected database, all infrequent items as well as all items before the first occurrence of x_i should be ignored. Then, PrefixSpan finds the length-2 sequential patterns in $\langle x_i\rangle$-projected database in the form of $\langle x_i x_j\rangle$. According to x_j's, the sequential patterns with prefix $\langle x_i\rangle$ can be further divided into subsets and can grow recursively.

Some specific techniques have been developed to enable efficient implementation of PrefixSpan. Interested readers might refer to [13] for a detailed technical discussion.

The traditional PrefixSpan method is used to mine the frequent sequences. However, for the tree mining, we can make some optimization to improve its efficiency. Since the frequent structures must keep $\mathcal{AD}$, we can apply some techniques to reduce the amount of the generated frequent sequences.

The improved PrefixSpan algorithm, finding Frequent Sequences keeping $\mathcal{AD}$, is shown in Fig.5.

Suppose there is an ordered-labeled tree shown in Fig.6(a). We need to record all occurrences of the frequent nodes for efficiency. Fig.6(b) shows the

Algorithm 2. Finding Frequent Sequences keeping $\mathcal{AD}$
INPUT: a dataset *DB*, support threshold minsup, and a parameter Sequence which is initialized to NULL
OUTPUT: the list of frequent sequences and related information

```
FindSequence (DB, minsup, Sequence)
{
  node-list=find all frequent nodes in DB;
  For each node in node-list
  {
    result=Join(Sequence,node);
    support=KeepingAD(result,DB);
    if(support is larger than minsup)
    {
      output(result);
      NewPDB = CreatePDB(result);
      FindSequence(NewPDB,minsup,result)
    }
  }
}
```

Fig.5. Finding frequent sequences keeping $\mathcal{AD}$.

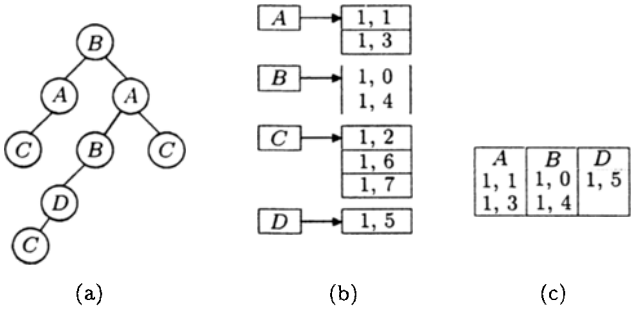

Fig.6. Example of keeping $\mathcal{AD}$.

data structure we adopt. Each element in the list are stored in the form of $\langle TID; position \rangle$, where *TID* records the IDs of the original tree, and *position* stores the position of the node in the pre-order sequence of the tree, which begins with 0. Fig.6(b) also shows the result after scanning the database. If A, B, C and D are the frequent nodes, we scan the tree once more to gain the information of these nodes. If the TID of tree in Fig.6(a) is 1, A has occurred twice, in positions 2 and 4, so the list of A will contain two elements: $\langle 1, 1 \rangle$ and $\langle 1, 3 \rangle$. Other nodes' information can be gained in the same way. When we deal with the prefix, we use the intersection operator on all the TID fields related with the nodes in the prefix, then, do the further analysis on those trees in the TID intersection set. For each tree with the TID in the set, we only need to extract the information about the prefix. For example, for the prefix ABD in Fig.6(a), we construct a structure shown in Fig.6(c) according to the fields in Fig.6(b). This structure is beneficial to the pattern counting. The number of distinct TID is the pattern frequency. If the shared TID numbers of two patterns (one is a prefix, the other is a node) are less than the minimum threshold, they cannot be combined as a frequent pattern, so they can be dropped. This structure can be set up in one scanning. For the further projected database, we only need to ignore the first part of each node's list.

3.2.1 *Shrinking the Projected Database*

When dealing with the projected database of 1-length sequences, we can shrink it efficiently. For instance, for the tree $B1A2C3A2B3D4C5C3$ shown in Fig.6(a), instead of the projection $CABDCC$, we generate the projected database of A by projecting it into two trees: C and $BDCC$. This is because the result of the projection is in a two-tree one if we do it directly, but for convenience in the later part, dividing them into two is a trivial method. But this technique can only be applied in the root of the original tree. If the length of prefix is more than one, the rest nodes are all the descendants of the root, and belong to one tree, which cannot be divided.

Meanwhile, this projected database do not need to copy the sequences from the original one. What we need is only to store the head and end links of the corresponding position in the two trees. This technique can be referred as *pseudo-project*.

3.3 Finding Frequent Substructures

Once we discover all frequent sequences, we enter the second step of the algorithm shown in Fig.7.

Algorithm 3. Finding Frequent Structures
INPUT: a tree database TDB, frequent sequences set *ft*
OUTPUT: isomer set

```
Process(ft, TDB)
{
  tree-list = find all trees in TDB;
  For each tree t in tree-list
  {
    node-list = find all nodes in tree t;
    For each node n in node-list
      //expand current node in ft
      ExpandNode(ft, n);
  }
}
```

Fig.7. Finding frequent structures.

In this algorithm, we should scan the corresponding trees in the original dataset (*TDB*). Firstly, we extract sequences from the current tree

according to the frequent sequences obtained from the first phase, and generate all possible isomers for them. Then we decide whether these isomers exist in the current tree or not. If it is in the tree, the isomer is added to the result list, or the count of the corresponding isomer in the result list is increased by one. Every node in each tree is recursively processed, until the whole *TDB* is scanned. When the algorithm finishes, all the isomers are generated and the corresponding times of appearances are counted.

For example, given a tree $A1B2C3D2$, the sequences for all its possible isomers can be: A, AB, B, ABC, AC, BC, C, $ABCD$, ABD, ACD, AD, BCD, BD, CD and D. If the sequence ABC exists in the frequent sequences resulting from the first phase, the isomer $A1B2C3$ will be added to the result list; if $A1B2C3$ has been in the result list, its counter is increased by one. If the sequences do not exist in the frequent sequences, the isomers need not be generated or tested any more.

Now let us use an example to illustrate how the isomers are generated by the algorithm.

Example 1. Given the data set of ordered labeled trees in Fig.8, now comes to the tree with TID i. Meanwhile, all the frequent sequences discovered are (A, B, C, D, AB, AD, DC, ABC, ABD, ADC, $ABDC$).

Because there are four nodes in the current tree, four recursive steps are needed here, as shown in Fig.9. In the first step, we scan node A. For it is the root node, no ancestor node of it is in the index list. From the frequent sequences, we get that the sequence A is frequent, so the isomer A_1 is added to the result list, and the node A is inserted in the index list.

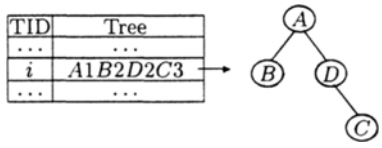

Fig.8. Data for finding isomer.

In the following three steps, we scan the index list reversely to get all the ancestors for the current node, and expand the ancestors with their child nodes and nodes connected to them. After the current node is added to their isomers, we get new candidate isomers. Then we decide whether the sequence representation of each candidate isomer belongs to the frequent sequences. If it is, the isomer is recorded; otherwise, it is discarded. Meanwhile, the current node is inserted into the index list.

After an ordered labeled tree is scanned, all isomers from it are collected. If the isomer has been in the result list, its corresponding counter increases. Continue to scan all the trees one by one, until the whole data set has been scanned. Then the algorithm stops.

4 Experiments

According to the conclusions in [1], we only

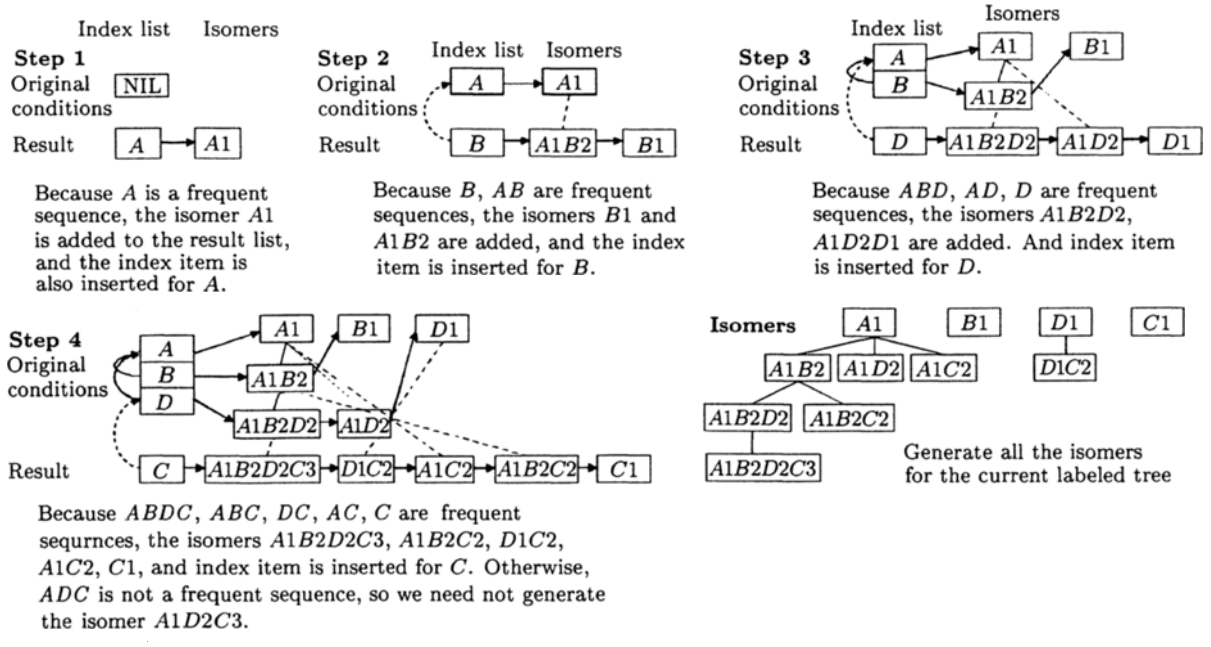

Fig.9. Process of finding isomer.

choose *TreeMinerV* as the opponent of *Chopper*. It shows that *Chopper* outperforms *TreeMinerV*, especially in the case of large amount of complex tree structures.

All the experiments are performed on a Pentium IV 1.7GHz PC with 512MB RAM. The OS is Red Hat Linux 9.0 and the algorithm is implemented in C++.

4.1 Experiments on Synthetic Data

4.1.1 Synthetic Data Generation

All the generated data are in the form of (*TreeID*, *TreeSeq*), where *TreeID* identifies each ordered labeled tree and *TreeSeq* is the pre-order code of that tree.

We wrote a synthetic data generation program to output all the test data. There are 8 parameters for data generation adjustment. They are: the number of the labels S, the probability threshold of one node in the tree to generate children 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 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.

A synthetic dataset is generated in 3 phases as follows. First, we generate a label set of S. Then, according to above parameters, we generate all the BPT and synthetic trees. Last, we use the BPT to overlap in the synthetic trees and replace or add some nodes and branches in synthetic trees to obtain the final dataset.

The second phase in the generation can be done according to the following steps. For a given node, we assign a label from label set with equal possibility. Then we use a random-number generator to get a number. If the number exceeds p, we determine how many children this node can have by choosing a number n between 0 and $C(F)$ in equal possibility and call this method recursively to generate its children; if the number is less than p, we must turn back to its parent to deal with the next sibling or turn back more in the case that all the siblings have been generated. This process will terminate when the height of the tree reaches the value specified by the parameter $I(H)$.

In the overlapping phase, the height and maximum fanout of the synthetic tree is not changed. We choose a BPT from the BPT set with equal possibility, randomly choose a node within valid level in the synthetic tree, treat it as a temporary root, and overlap the BPT's root by this root. All the nodes under the temporary root should be reassigned a label with the label of the corresponding node in BPT. If there is no such node, add a new one. This procedure will terminate when all the chosen BPT nodes are put into the synthetic tree. Now the synthetic tree is ready.

4.1.2 Performance on Synthetic Data

At first, we consider the scalability with *minsup* of the two algorithms, while other parameters are: $S = 100$, $p = 0.5$, $L = 10$, $I = 4$, $C = 3$, $N = 10{,}000$, $H = 8$, $F = 6$. Fig.10 shows the result, where the *minsup* is set from 0.1 to 0.003. In this figure, both X and Y axes have been processed by $\log_{10} T$ for the convenience of observation. We can find, that *Chopper* is a winner, especially, *TreeMinerV* is halted in 3 hours for memory overflow when *minsup* = 0.02, while the *Chopper* goes well. It should also be noted that, *Chopper* does not perform excellently until *minsup* is dropped to 0.004.

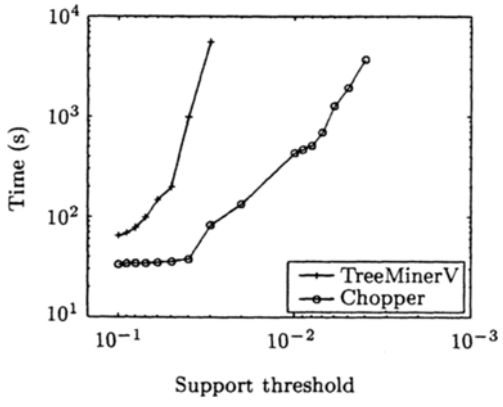

Fig.10. MinSup vs. time.

Fig.11 shows the scalability with data size. In this figure, the Y axe has been processed by $\log_{10} T$ for the convenience of observation. The data size N varies from 10,000 to 50,000, 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 *Chopper* is extremely less than that of *TreeMinerV* which is halted for memory overflow. The reason is that *Chopper* can save time and space cost by avoiding false candidate subtree generation.

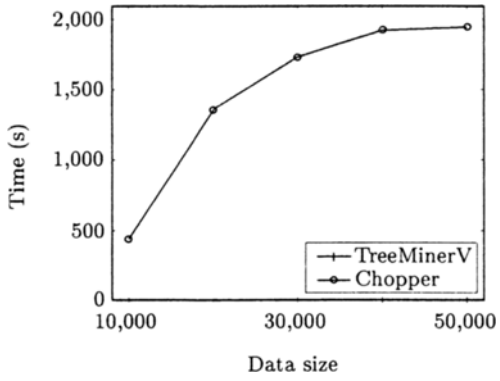

Fig.11. Data size vs. time.

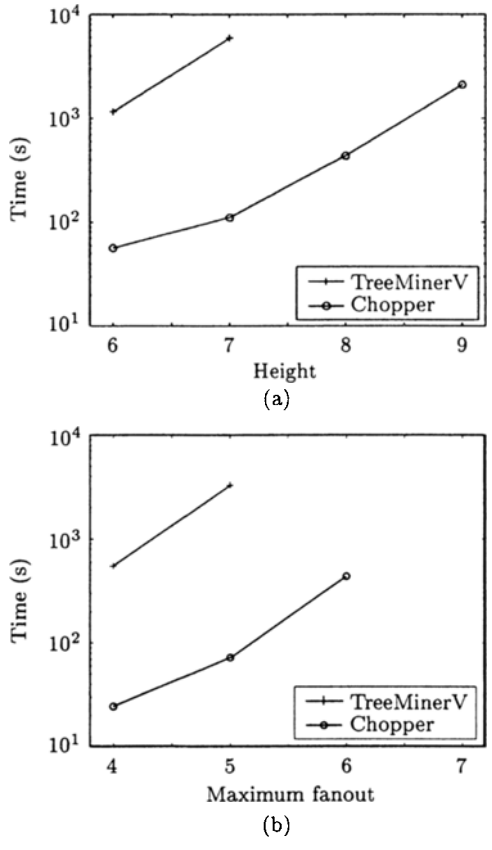

Fig.12. Tree size vs. time. (a) Height vs. time. (b) Maximum fanout vs. time.

Finally, the scalability with tree size is shown in Fig.12. In this figure, the Y axes have been processed by $\log_{10} T$ for the convenience of observation. In Fig.12(a), we only vary H from 6 to 9. It is easy to find that, when H equals 6 or 7, the performance of *Chopper* is better than that of *TreeMinerV*. However, when the trees become higher, the superiority grows. In particular, when H equals 8 or 9, *Chopper* thoroughly defeats *TreeMinerV* for the reason that *TreeMinerV* is halted for memory overflow. In Fig.12(b), the performance of *Chopper* and *TreeMinerV* is similar to the case above. *Chopper* certainly performs better than *TreeMinerV*, while the fanout continues to increase.

4.2 Experiments on Real Data

We used *Chopper* in Web Usage Mining. We downloaded the Weblog[16] of Hyperreal[17], chose those dated from Sept.10 to Oct.9, 1998 as the input data, and then transformed the Weblog into tree-like data set which included over 12,000 records totally.

4.2.1 Performances on Real Data

Fig.13 shows the performance of the two algorithms, where the *minsup* is set from 0.1 to 0.0006. 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 *Chopper* is better than that of *TreeMinerV*. Especially, *TreeMinerV* is halted in 3 hours for memory overflow when *minsup* = 0.0006, while the *Chopper* goes well.

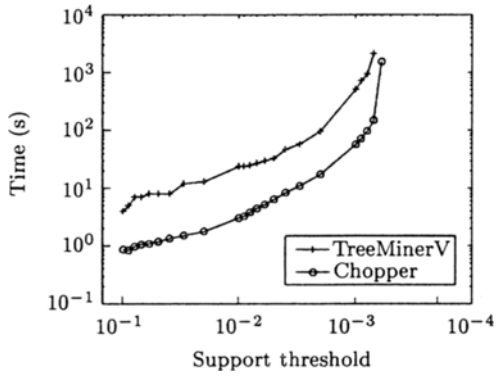

Fig.13. MinSup vs. time.

Finally, Fig.14(a) shows the number of frequent patterns generated by the algorithm, while Fig.14(b) shows the average number of the nodes of the frequent patterns generated by the algorithm, where the *minsup* is set from 0.1 to 0.0006. For the convenience of observation, both X and Y axes have been processed by $\log_{10} T$ in Fig.14(a), while Y axe has been processed by $\log_{10} T$ in Fig.14(b).

4.2.2 Results on Real Data

Fig.15 shows some results. So if we modify the

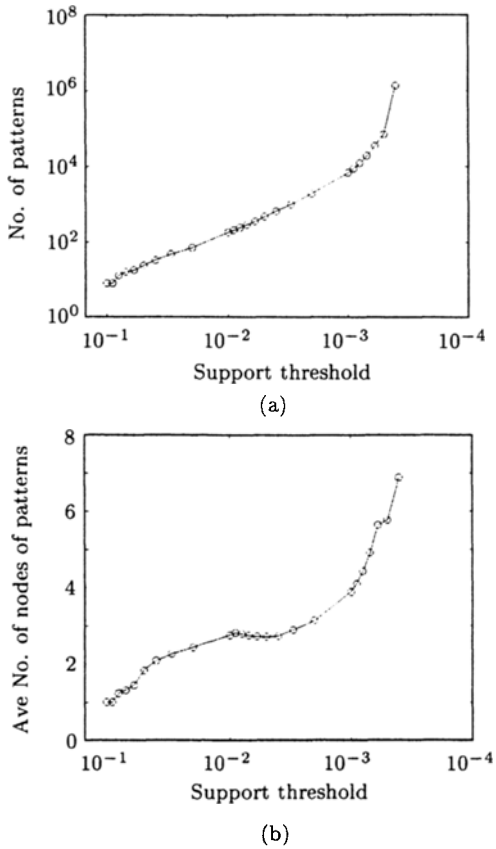

Fig.14. MinSup vs. patterns. (a) MinSup vs. the number of patterns. (b) MinSup vs. the Avg No. of nodes of patterns.

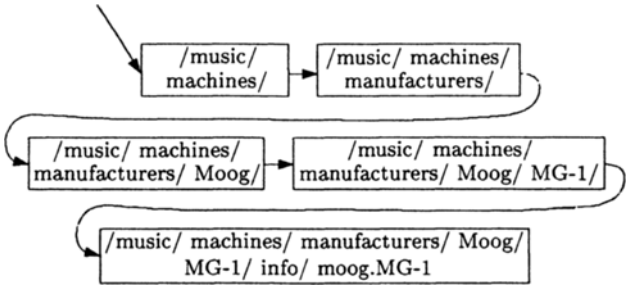

Fig.15. Example of the result.

web site architecture according to the results shown in the figure, it will be more efficient for users to browse this site, which will in turn improve the hit ratio of the web site.

The result can be explained and exploited as follows: After examining the mining result, we found that among the users who visited the webpage of Moog, approximately half of them visited the page MG-1. So we suggest that the webmaster place a "Hot" tag to the right of MG-1's link. Similarly, it is interesting to find that almost all the remaining users continued to visit "moog.MG-1", which consists of the description and comments of synth MG-1 from people who bought it. Usually, when someone is interested in a product and may plan to buy it, it is likely that he or she wants to hear others' comments on this product. These words are not like those in commercials, and often offer more down-to-earth and reliable information. This explains the webpage access pattern quite well. We also notice that currently the section containing "moog.MG-1" is not arranged at the top of the webpage. Considering the psychological factor of users described above, it is wise to re-build this page by placing "moog.MG-1" at the top. In short, if we modify the web site architecture according to the results shown in the figure, it will be more efficient for users to browse this site, which will in turn further improve the hit ratio of the web site.

Still, there are some other results in Fig.16. Here we do not explain them any more.

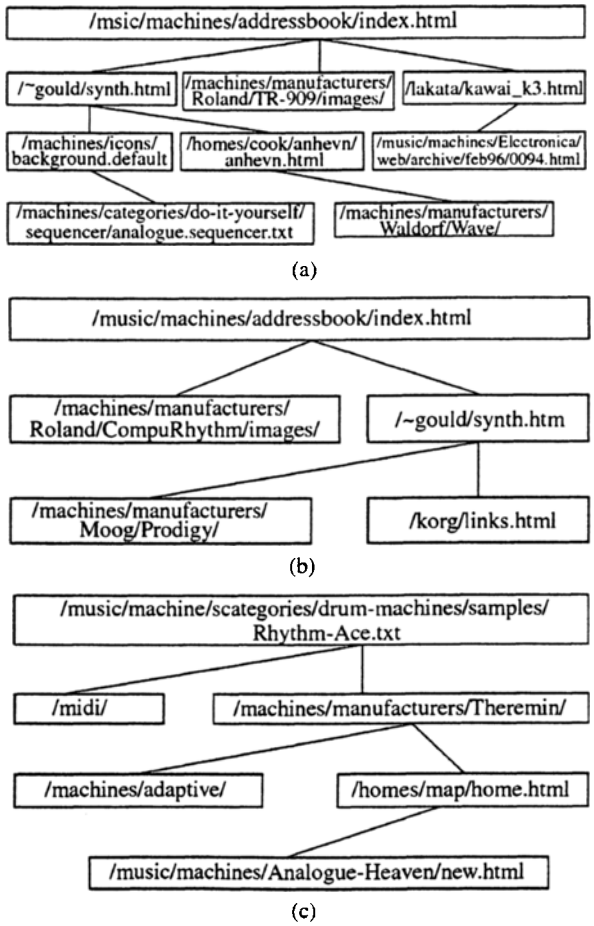

Fig.16. Some results of the Weblog analysis. (a) 19970212 *minsup* = 0.002; (b) 19980731 *minsup* = 0.005; (c) 19980824 *minsup* = 0.003.

5 Conclusion

We are drown in the world of numerous and complicated documents. We expect eagerly to find a path to the golden nuggets in a short time. Here we present a new mechanism to achieve this goal. Many documents are semi-structured, which contain some incomplete structural information, for instance, the XML documents. There are many similar substructures in these documents, so it is very useful to gain the frequent structures through mining, which leads to the results of similar information among these documents. In this paper, we introduce a new algorithm *Chopper*, which can tackle the problem efficiently.

The next step for us is to improve our work with the aim of solving some special structures in the documents, such as the nested or cyclic ones. Other interesting work includes coupling the structural information within the process of sequence generating tightly, with the hope of making the performance better.

Acknowledgement Here we want to give our sincere thanks to Ming-Sheng Hong and Jin Pan for the coding work and helpful discussions.

References

[1] Zaki M J. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Copyright 2002 ACM 1-58113-567-X/02/0007, July 2002.

[2] Cook D, Holder L. Substructure discovery using minimal description length and background knowledge. *Journal of Artificial Intelligence Research*, 1994, 1: 231–255.

[3] Agrawal R, Mannila H, Srikant R *et al.* Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, Fayyad U *et al.* (eds.), AAAI Press, Menlo Park, CA, 1996, pp.307–328.

[4] Cooley R, Mobasher B, Sravastava J. Web mining: Information and pattern discovering on the World Wide Web. In *8th IEEE Int. Conf. Tools with AI*, Newport Beach, California, USA, Nov. 1997, pp.558–567.

[5] Zaki M J. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, Jan/Feb 2001, 42(1/2): 112–120. Special issue on Unsupervised Learning.

[6] Asai T, Abe K, Kawasoe S *et al.* Efficient substructure discovery from large semi-structured data. In *Proc. SDM'02*, Hyatt Regency, Crystal City, Arlington, Virginia, USA, Apr. 2002, pp.158–174.

[7] Deahaspe L, Toivonen H, King R D. Finging frequent substructures in chemical compounds. In *Proc. KDD-98*, New York, USA, 1998, pp.30–36.

[8] Matsuda T, Horiuchi T, Motoda H *et al.* Graph-based induction for general graph structured data. In *Proc. DS'99*, New York, USA, 1999, pp.340–342.

[9] Mannila H, Meek C. Global partial orders from sequential data. In *Proc. KDD2000*, Boston, USA, 2000, pp.161–168.

[10] Miyahara T, Shoudai T, Uchida T *et al.* Discovery of frequent tree structured patterns in semistructured Web documents. In *Proc. PAKDD-2001*, Hong Kong, China, 2001, pp.47–52.

[11] Wang K, Liu H. Schema discovery for semistructured data. In *Proc. KDD'97*, Newport Beach, USA, 1997, pp.271–274.

[12] Wang J T L, Shapiro B A, Shasha D *et al.* Automated discovery of active motifs in multiple RNA secondary structures. In *Proc. KDD-96*, Portland, USA, 1996, pp.70–75.

[13] Pei J, Han J, Mortazavi-Asl B *et al.* PrefixSpan: Mining sequential patterns by prefix-projected growth. In *Proc. ICDE01*, Heidelberg, Germany, April 2001, pp.215–224.

[14] Scott Fortin. The graph isomorphism problem. Technical Report No. TR96-20, Dept. of Computer Science, University of Alberta, 1996.

[15] Richard Cole, Ramesh Hariharan, Piotr Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time. In *Proc. the 10th Annual ACMSIAM Symposium on Discrete Algorithms*, Robert E Tarjan, Tandy Warnow (eds.), Baltimore, Maryland, USA, Jan. 1999, pp.245–254.

[16] http://music.hyperreal.org

[17] http://www.cs.washington.edu/research/adaptive

Chen Wang was born in 1976. He received his B.E. degree and M.S. degree in computer science from Soochow University in 1999 and 2002 respectively. Now, he is currently a Ph.D. candidate in computer science at Fudan University. His research interests include data mining, database and knowledge base.

Qing-Qing Yuan was born in 1978. She received her B.E. degree and M.S. degree in computer science from Fudan University in 2000 and 2003 respectively. Her research interests include data mining, database and knowledge base.

Hao-Feng Zhou was born in 1975. He received his B.E. degree in computer science from Shanghai University in 1997, his M.S. degree and Ph.D. in computer science from Fudan University in 2000 and 2003 respectively. His research interests include data mining, database and knowledge base.

Wei Wang was born in 1970. He received the M.S. degree in 1992 and the Ph.D. degree in 1998. Now he is an associate professor of the Dept. of Computing and Information Technology, Fudan University. His main research areas include spatial-temporal database, constraint database, index technology and semistructure database.

Bai-Le Shi was born in 1935. He received the M.S. degree in 1956. Now he is a chief professor of the Dept. of Computing and Information Technology, Fudan University. His main research areas include object-oriented database, knowledge database, digital library.