PrefixTreeESpan: A Pattern Growth Algorithm for Mining Embedded Subtrees

Lei Zou* Yansheng Lu* Huaming Zhang※ Rong Hu*

*HuaZhong University of Science and Technology, 1037 Luoyu Road, Wuhan, P. R. China
※The University of Alabama in Huntsville, Huntsville, AL, 35806, USA
{zoulei,lys}@mail.hust.edu.cn , hzhang@cs.uah.edu , hurong@smail.hust.edu.cn

Abstract. Frequent embedded subtree pattern mining is an important data mining problem with broad applications. In this paper, we propose a novel embedded subtree mining algorithm, called *PrefixTreeESpan* (i.e. **Prefix-Tree**-projected **E**mbedded-**S**ubtree **pa**tter**n**), which finds a subtree pattern by growing a frequent *prefix-tree*. Thus, using divide and conquer, mining local length-1 frequent subtree patterns in *Prefix-Tree-Projected database* recursively will lead to the complete set of frequent patterns. Different from *Chopper* and *XSpanner* [4], *PrefixTreeESpan* does not need a checking process. Our performance study shows that *PrefixTreeESpan* outperforms *Apriori-like* algorithm: *TreeMiner* [6], and *pattern-growth* algorithms :*Chopper* , *XSpanner* .

1. Introduction

Mining frequent structural patterns from a large tree database [2, 4, 6] and a graph database [5] is a new subject in frequent pattern mining. Many mining subtree pattern algorithms have been presented. Generally speaking, these algorithms can be broadly classified into three categories [3]. The first category is Apriori-like, such as *TreeMiner*; the second category of algorithms is based on enumeration tree, such as *FREQT* [1] and *CMTreeMiner* [3]. The above two categories are based on candidate-generation-and-test framework. The last category is based on pattern-growth.,such as *Chopper* and *XSpanner* [4]. In this paper, we present a novel subtree pattern mining method based on pattern-growth, called **PrefixTreeESpan** (i.e.**Prefix-Tre**e-projected **E**mbedded-**S**ubtree **pa**tter**n**). Its main idea is to examine the prefix-tree subtrees and project their corresponding project-instances into the projected database. Subtree patterns are grown recursively by exploring only local frequent patterns in each projected database. Using divide and conquer, this algorithm finds the complete set of frequent patterns correctly.

To summarize our contributions in this paper: 1) we define *prefix-tree* and *postfix-forest* for tree databases. The importance of *prefix-trees* and *postfix-forests* is that for tree databases, they play the role of *prefix* subsequences and *postfix* subsequences as in sequence databases; 2) we define *GE* (i.e ***G**rowth **E**lement*) for a *prefix-tree*. Because of *GE*, any local frequent pattern in some projected database must correspond to one frequent subtree pattern in the original database. So no candidate checking is needed, which is different from *Chopper* and *XSpanner*; 3) we propose a pattern growth algorithm *PrefixTreeESpan* for mining frequent *embedded* subtrees.

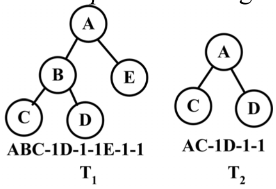

Fig.1. Embedded Subtree and Pre-Order-String

2 Preliminaries

Trees and **Embedded Subtree:** Our trees are always **ordered, labeled, and rooted,** which can be denoted as T=(V,v_0,E ,Σ,L), where (1) V is the set of nodes ; (2) v_0 is the root; (3)E is the set of edges; (4)Σ is an alphabet; (5) L is an function: V→Σ that assigns labels to nodes.

For a tree *T* with node set V, edge set *E* , we say that a tree T′ with node set V′, edge set E′, is an embedded subtree of *T* if and only if (1) V′ ⊆V, (2)the labeling of nodes of V in T is preserved in T′, (3) $(v_1,v_2)\in E'$, where v_1 is the parent of v_2 in T′, if and only if v_1 is an **ancestor** (including parent) of v_2 in T and (4) for v_1, v_2 ∈V′, *preorder(v_1)<preorder(v_2)* in T′ if and only if *preorder*(v_1) < *preorder(v_2)* in T. If T′ is an embedded subtree of T, it will be denoted as *T′ ∈T.* Obviously, in Fig.1, T_2 is embedded subtree of T_1.

Mining Frequent Subtree Patterns Task: Given a trees database *D*={T_i| i in a index set}, where T_i is a tree , and a minimal support count *min_sup≥0,* and a pattern tree t_i, we define $d(t_i,T)$=1 if and only if t_i ∈T, otherwise $d(t_i,T)$=0. The problem of mining frequent embedded subtrees is defined as to discover all pattern trees t_i, such that $Sup_D(t_i)=\sum_{T\in D} d(t_i,T)\geq min\_sup$, where $\boldsymbol{Sup_D(t_i)}$ is denoted as the support count of the pattern tree t_i in the tree database *D*.

Pre-Order-String: According to [2], we have the recursive definition for the *pre-order string* : (1) for a rooted ordered tree *T* with a single node *r* , the pre-order string of *T* is $S(T)= l_r$–1, where l_r is the label for the single node *r*, '–1' is called *end flag*; and (2) for a rooted ordered tree *T* with more than one node, assuming the root of *T* is *r* (with label l_r) and the children of *r* are $r_1 \ldots\ldots r_k$ from left to right then the pre-order string for *T* is $S(T)= l_r S(T_{r1})\ldots S(T_{rk})\ -1$.

3. PrefixTreeESpan: Mining Embedded Subtree Patterns by Prefix Tree Projections

Definition 1 (**Prefix-Tree**) Let *T* be a tree with *m nodes*, *S* be a tree with *n* nodes*, where* n≤m. When pre-order scanning the tree *T* from its root until to the *n-th* node, the scanning path forms a tree *M*. If the tree *S* is isomorphic to the tree *M*, we call *S* a ***Prefix-Tree*** of the tree *T*.

An example of prefix-trees is illustrated in Fig. 2. Given a tree *T* on the left, its five prefix trees are shown on the right, which have 1,2,3,4,5 nodes respectively.

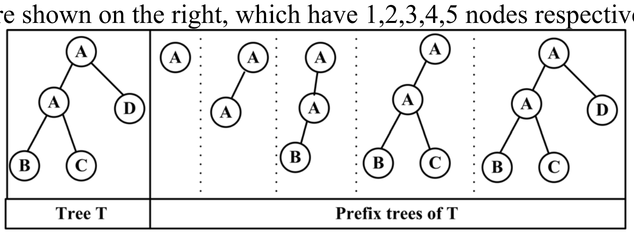

Fig.2. An example of Prefix trees

Example 1 (Running Example) Let *D* be a tree database as in Fig.3(a) and *min_support*=2. The set of node labels is {A,B,C,D,E}. Frequent subtrees can be mined by a **prefix-tree-projection** method in the following steps.

Step 1: Find length-1 frequent embedded subtree patterns. Scan the tree database *D* to find all frequent labels in trees. Each frequent label is a length-1 subtree patterns. They are <A^1 –1>, <B^1 –1>, <C^1 –1>, <D^1 –1> (*pre-order-string* of a tree). Obviously they are reported since they are frequent subtrees, as shown in Fig.3(b). Notice that Arabic numeral in the superscript of 'A' in <A^1 –1 > denotes the position of 'A' in pre-order traversing the pattern tree <A^1 –1 >.

Step 2: Divide search space. The complete set of frequent subtrees patterns can be partitioned into the following four subsets according to the four prefix trees: (1) the ones having prefix-tree <A^1 –1>; (2) the ones having prefix-tree <B^1 –1>;(3) the ones having prefix-tree <C^1 –1>; (4) the ones having prefix-tree <D^1 –1>.

Step 3: Find subsets of subtrees. Construct corresponding *projected database* and mine each recursively to find the subsets of patterns subtrees.

How to construct the projected database?

For example, in order to construct <A^1 –1>-projected database, we scan the database *D* to find all **occurrences** of node 'A'. For **each occurrence** of 'A', the nodes after the occurrence, according to **pre-order traversal**, are formed to the **project-instance**. Obviously, one project-instance is corresponding to one occurrence. All project-instances are collected as <A^1 –1>**-project-database**, as shown in Fig.4(a) . There are three **occurrences** of the node 'A' in the database D, as shown in Fig.3(a). For the fist occurrence of 'A', the nodes 'C' and 'E' are after the occurrence. They are formed the first project-instance. Though other nodes, such as 'A', 'B', 'D', are also after the first occurrence, they cannot be attached to the first occurrence directly or indirectly.

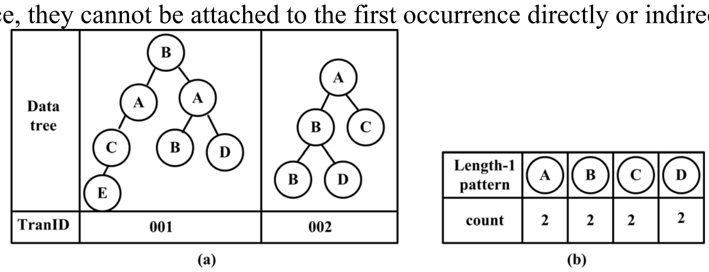

Fig. 3. (a) Database D and (b) frequent length-1 subtree patterns

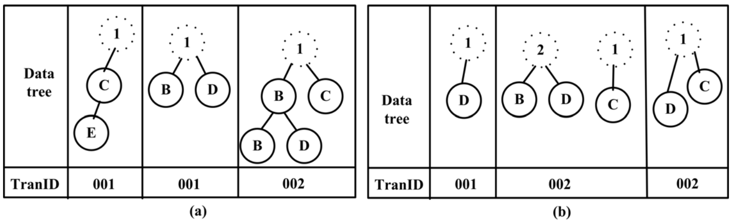

Fig. 4. (a) <A^1 –1>-projected database and (b) <A^1 B^2 –1 –1>-Projected database

---

Algorithm PrefixTreeESpan
Input:A tree database D, minimum support threshold *min_sup*
Output: All frequent subtree patterns
Methods:
1) Scan D and find all frequent label *b*.
2) **For each** frequent label *b*
3) Output pattern tree <b –1>;
4) Find all **Occurrences** of b in Database D, and construct <b –1>-projected database through collecting all corresponding *Project- Instances* in *D*;
5) **call** *Fre*(<b –1>, 1 , ProDB(D,<b -1>) , *min_sup*).

Function Fre(S, *n* , ProDB(D,S), *min_sup*)
Parameters: *S*: a subtree pattern ; *n*: the length of S; *ProDB(D,S)* : the <S>-projected database; *min_sup*: the minimum support threshold.
Methods:
1) Scan *ProDB(D,S)* once to find all frequent ***GEs*** *b*.
2) **For each** GE *b*
3) extent *S* by *b* to form a subtree pattern S' , and output S' .
4) Find all **Occurrences** of *b* in ProDB(D,S), and construct < S' > -projected database through collecting all corresponding *Project- Instances* in *ProDB(D,S)* ;
5) **call** *Fre* (S' , *n+1* , *ProDB(D,* S'),*min_sup*).

Fig .5. Algorithm PrefixTreeESpan

Definition 2 (Growth Element) Suppose that we have a tree *T* having *m* nodes, another tree T' having (m+1) nodes, and *T* is the **prefix-tree** of T' . The node ***n*** which exists in T' but not exists in *T*, denoted as ($T' \backslash T$), is called as the ***Growth Element*** (**GE** for short) of *T* w.r.t T' . Actually, the node ***n*** is the (m+1)-*th* node of T' by pre-order travel. Please notice that a *GE* represents not only the label of ***n***, but also its attaching position in *T*. A GE is always denoted as (*L*, *n*), where *L* is label, and *n* means that the GE is attached to the *n-th* node of the prefix-tree.

Scanning the <A^1 –1>-projected database, as shown in Fig.4(a), we will find all **GEs**. They are {(C,1),(E,1),(B,1),(D,1)}. But (E,1) is not frequent. So frequent subtree patterns having prefix tree <A^1 –1> can be partitioned into 3 subsets: (1) those having prefix tree <A^1 B^2 –1 –1>; (2) those having prefix tree <A^1 C^2 –1 –1>; (3) those having prefix tree <A^1 D^2 –1 –1>. These subsets can be mined through constructing respective projected databases and mining each recursively.

For example, in order to construct <A^1 B^2 –1 –1 >-projected database, we scan the

$<A^1 -1>$-projected database to find all occurrences of GE (B,1). Each occurrence is corresponding to a project-instance. All project-instances are formed $<A^1\ B^2 -1 -1>$-projected database. The GEs in $<A^1\ B^2 -1 -1>$-projected database are {(D,1),(B,2),(D,2),(C,1)}. Notice that, the GE (D,2) is different from the GE (D,1). Since (D,2) means that the GE is attached to the second node of prefix-tree $<A^1\ B^2 -1 -1>$, while (D,1) means that the GE is attached to the first. We will find that only GE (D,1) is frequent. Recursively, all frequent pattern subtrees having prefix-tree $<A^1\ B^2 -1 -1>$ can be partitioned into only 1 subset: those having prefix-tree$<A^1\ B^2 -1\ D^3 -1 -1>$. We have to omit the mining process due to space limited.

Our algorithm *PrefixTreeESpan* is given in Fig.5.

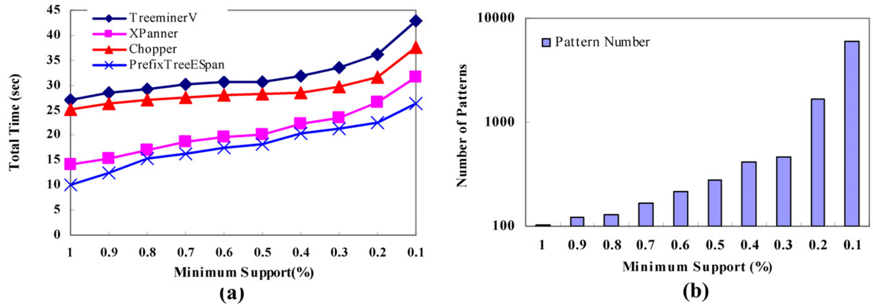

Fig.6. (a) The running time of experiments on T1M and (b) The number of patterns on T1M

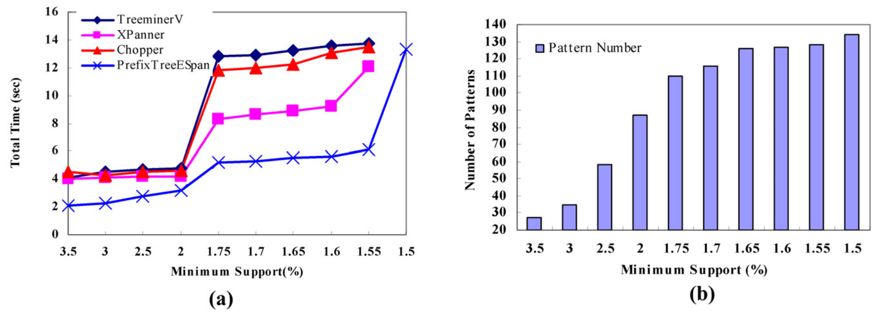

Fig.7. (a) The running time of experiments on CSLOG and (b) The number of patterns on CSLOG

4. Experiments

In this section, we evaluate the performance of *PrefixTreeESpan*, and compare it with some other important algorithms. All experiments are done on a 1.7GHz Pentium Ⅳ PC with 1 GB main memory, running RedHat Fedora Core 3. All algorithms are implemented in standard C++ with STL library support and compiled by g++ 3.4.2 compiler. We compare our method *PrefixTreeESpan* with other important algorithms, such as *TreeMiner*, *Chooper* and *XSpanner* on 2 datasets, i.e. *T1M* and *CSLOG**. *T1M* is a synthetic dataset, which is generated by the tree generation program* provided by

* Available at http://www.cs.rpi.edu/~zaki/software/

Zaki [6].For *T1M*, we set parameters T=1,000,000 , *N*=100, *M*=10,000,*D*=10, *F*=10. *CSLOG* is a real dataset, which is also provided by Zaki [6]. The details about the tree generation program, parameters and *CSLOG* are given in [6]. Fig.6a and Fig.7a show the total running time, and the number of patterns is shown in Fig.6b and Fig.7b. Generating candidate patterns and testing in algorithm *TreeMiner* consume too much time, which leads to not good performance. In *Chopper* and *XSpanner*, we have to check *l-patterns* discovered in the first step whether there exit subtree patterns corresponding to them, which is a computationally expensive task. But our method overcomes this problem.

5. Conclusions

In this paper, we present a novel algorithm *PrefixTreeESpan* (i.e. **Prefix-Tre**e-projected **E**mbedded-**S**ubtree **pa**tter**n**) for mining embedded ordered subtree patterns from a large tree database. Mining local length-1 frequent subtree patterns in *Prefix-Tree-Projected database* recursively will lead to the complete set of frequent patterns.

Acknowledgments:

We acknowledge to some valuable suggestions from Professor Jian Pei at Simon Fraser University. We also express our sincere thanks to Professor Mohammed J. Zaki at Rensselaer Polytechnic Institute for providing the synthetic data generator, CSLOGS data set and source codes of algorithm *Treeminer*. We thanks to Professor Wei Wang and his students at Fudan University. They provide us with the executable codes of algorithm *XSpanner*.

References:

1. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. *Efficient Substructure Discovery from Large Semi-structured Data*. in *Proc. Second SIAM Int'l Conf Data Mining, Apr.2002*. 2002.
2. Y. Chi, S. Nijssen, R.R. Muntz, and J.N. Kok, *Frequent Subtree Mining -An Overiew.* Fundamenta Informaticae, 2005.
3. Y. Chi, Y. Xia, Y. Yang, and M. Richard R, *Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees.* IEEE Transactions on Knowledge and Data Engineering, 2005.
4. C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. *Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining*. in *PAKDD 2004*. 2004.
5. X. Yan and J. Han. *CloseGraph: Mining Closed Frequent Graph Patterns*. in *KDD'03*. 2003.
6. M.J. Zaki. *Efficiently Mining Frequent Trees in a Forest*. in *Proceedings of the Int. Conf. on Knowledge Discovery and Data Mining*. 2002. Edmonton, Alberta, Canada.