Direct Mining of Closed Tree Patterns With Subtree Constraint

Viet Anh NGUYEN Koichiro DOI Akihiro YAMAMOTO

Graduate School of Informatics, Kyoto University

Two critical bottle necks in mining frequent tree patterns from tree databases are the exponential number of mined patterns and the lack of user focus on the mining process. We propose, in this paper, an algorithm that solves the problems for unordered attribute trees by mining only the compact representation of tree patterns, i.e. closed tree patterns, and allows users to mine only trees of their interest by specifying subtree constraints. The experimental results show the efficiency of our algorithm.

1. Introduction

Although there have been many algorithms on efficient tree pattern mining, e.g. [Zaki 02, Asai 02, Chi 04], in many cases, the overwhelming number of patterns generated may confuse users. We consider the problem of mining unordered closed tree patterns with subtree constraint. Subtree constraint is a tree itself which is specified by the user and must be included in all patterns generated. In practice, users may not be interested in all frequent subtrees but only some that are super trees of a given pattern tree. Subtree constraint would also be useful, for example, in Web-mining, where the user want to extract common patterns around some given information from Web pages. The proposed method is efficient firstly because it mines only concise representation of tree patterns, and secondly because it adopts a top-down method which helps to minimize the number of scan over the dataset and to avoid many redundant generating and checking of intermediate candidate subtrees as of the case of bottom-up method.

2. Preliminaries

A *labeled tree* is an acyclic connected graph whose every vertex (or node) is assigned a label. A *rooted tree* is a tree that has a special node called the *root*. A node v on the path from the root to a node w is called an *ancestor* of node w, in which case w is called a *descendant* of v. Each of the closest descendants of v is called a *child* of v, in which case v is called the *parent* of the child. A tree S is called a *subtree* of a tree T ($S \preceq T$) if there exists a one-to-one mapping φ from nodes in S to T, such that φ preserves the parent-child relation as well as the node labels. If $S \preceq T$ holds, we also say S occurs in T, T is a *super tree* of S, or T contains S.

Let $D = \{T_0, ..., T_n\}$ be a database where each of its transactions is a rooted, labeled tree. We call any tree T that occurs in D a *pattern*. The *support* of a tree T in D is defined as the number of trees in D that contains T. A tree T is called *frequent* if its support is greater than or equal to a threshold (*minsup*) specified by a user. A frequent subtree T is *closed* if no proper super tree of T has the same support that T has. Let S be a pattern specified by the user, the task is to enumerate all closed subtrees containing S.

Contact: Nguyen Viet Anh, Graduate School of Informatics, Kyoto University, Yoshida Honmachi, Sakyo-ku, Kyoto, 606-8501, Japan, Tel: +81-75-753-5628, Email: vietanh@iip.ist.i.kyoto-u.ac.jp

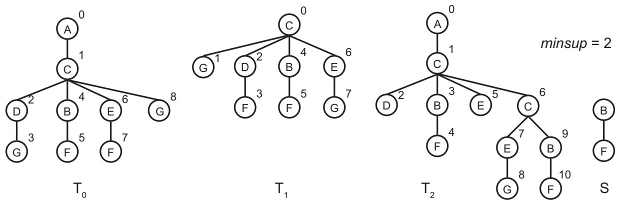

Figure 1: The running example

Existing algorithms could be efficient for mining all frequent tree patterns, but none of them are suitable for the problem of mining with subtree constraint. The only way to obtain the patterns satisfying the constraint is to generate all patterns and then filter out those which are not super tree of the subtree constraint. This approach is not efficient, especially when the databases to be mined are huge and changed continually. Our goal is to develop a computationally efficient method toward this problem.

The database given in Figure 1 is used as our running example. The database consists of three transactions, the *minsup* is set to 2, and the pattern S contains two nodes B and F. Each transaction has a unique id, i.e., T_0, T_1, and T_2, respectively. Each node in a transaction is assigned a unique number which is the node's index in the preorder traversal of the tree.

3. Mining under subtree constraint

The proposed method comprises of two phases. The first phase is to find all roots of closed subtrees containing the subtree constraint S. In the second phase, for each root node found in the first phase, we construct a data structure called an *RG-tree* which is the representation of subtrees rooted at the root node. We then propose an efficient algorithm to generate all closed trees directly from the *RG-tree*.

3.1 Searching for the roots

Root nodes of closed subtrees containing S can be found from a set composed of paths from the occurences of S to the roots of transactions where S occurs. This set is called

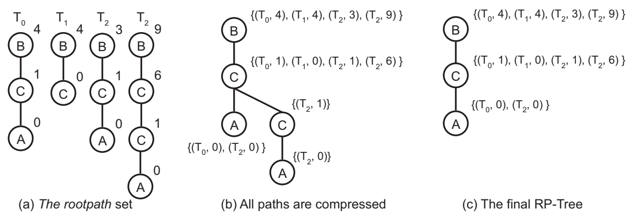
T_0
T_1
T_2
T_2
4
4
3
9
B
C
A
1
0
0
6
(a) *The rootpath* set
$\{(T_0, 4), (T_1, 4), (T_2, 3), (T_2, 9)\}$
$\{(T_0, 1), (T_1, 0), (T_2, 1), (T_2, 6)\}$
$\{(T_2, 1)\}$
$\{(T_0, 0), (T_2, 0)\}$
$\{(T_2, 0)\}$
(b) All paths are compressed
$\{(T_0, 4), (T_1, 4), (T_2, 3), (T_2, 9)\}$
$\{(T_0, 1), (T_1, 0), (T_2, 1), (T_2, 6)\}$
$\{(T_0, 0), (T_2, 0)\}$
(c) The final RP-Tree

Figure 2: Building an *RP-tree*

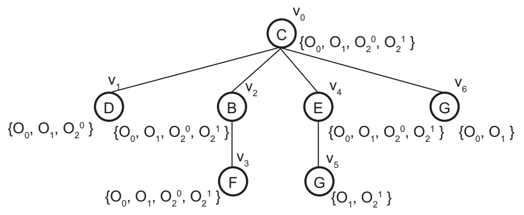
v_0
C
$\{O_0, O_1, O_2^0, O_2^1\}$
v_1
D
$\{O_0, O_1, O_2^0\}$
v_2
B
$\{O_0, O_1, O_2^0, O_2^1\}$
v_3
F
$\{O_0, O_1, O_2^0, O_2^1\}$
v_4
E
$\{O_0, O_1, O_2^0, O_2^1\}$
v_5
G
$\{O_1, O_2^1\}$
v_6
G
$\{O_0, O_1\}$

Figure 3: An *RG-tree*

a *rootpath* set and is compressed into a compact data structure called an *RP-tree*. Roots of *downward closed* subtrees (defined later) are generated directly from the *RP-tree*.

Each node v of an *RP-tree* is assigned a set of occurrences of the node v in D denoted by $OccList(v)$. The label of the root r of the *RP-tree* is the label of the root of S, and the $OccList(r)$ is the set of all occurences of S in D.

Figure 2 illustrates an example of building the *RP-tree* for the running example. Figure 2(a) shows the four paths of the *rootpath* set. After all the paths in the *rootpath* set are compressed, we obtain the *RP-tree* in Figure 2(b). Infrequent nodes will be pruned out from the *RP-tree* because they cannot be roots of any frequent subtree. Figure 2(c) shows the final *RP-tree*.

Let v be a node of an *RP-tree*. A frequent subtree T rooted at v is called *downward closed* if no super tree of T with the same root v has the same support that T has. Note that a downward closed subtree is not necessarily a closed subtree. A downward closed subtree T with the root v is closed in D if and only if there does not exist a super tree T' formed by adding a parent node to the root node v of T such that T' has the same support as T.

Lemma 1 *Let v be a node in an RP-tree. A downward closed subtree rooted at v contains the pattern S as its subtree.*

Let v and w be nodes of an *RP-tree*. The node v is called *transaction-matched* with the node w if for each transaction $T_i \in D$ where v occurs (with occurrences in $OccList(v)$) there exists at least one occurrence in $OccList(w)$.

Lemma 2 *Let v be a node of an RP-tree. If v is not transaction-matched with any of its child nodes, then there exists at least one downward closed subtree rooted at v which is also a closed subtree in D, otherwise, no downward closed subtree rooted at v be a closed subtree in D.*

```
Input: a database $D$ of trees, a minimum support
       $minsup$, a pattern $S$;
Output: roots of all downward closed subtrees
        containing $S$;
1 $Occ \leftarrow$ all occurrences of $S$ in $D$;
2 $rpSet \leftarrow$ all root paths started from all $occ \in Occ$;
3 $rpTree \leftarrow$ Build_RP_Tree($rpSet$,$minsup$);
4 $out \leftarrow$ Get_Roots($rpTree$);
```

Algorithm 1: Generating all the roots

Algorithm 1 outputs the roots of all downward closed subtrees containing a pattern S. The *Build_RP_Tree* procedure builds the *RP-tree* as we have discussed above. The *Get_Roots* function scans the *RP-tree* in the breadth-first manner starting from the root. If a node v being visited has no child nodes or v is not *transaction-matched* with any of its child nodes then v is added to the *out* set. For example, the root node A $\{(T_0, 0), (T_2, 0)\}$ and the root node C $\{(T_0, 1), (T_1, 0), (T_2, 1), (T_2, 6)\}$ are generated from the *RP-Tree* in Figure 2. The node B $\{(T_0, 4), (T_1, 4), (T_2, 3), (T_2, 9)\}$ is *transaction-matched* with its child node (the node C $\{(T_0, 1), (T_1, 0), (T_2, 1), (T_2, 6)\}$) and thus is not generated as a root node.

Lemma 3 *Let v be a root node generated by Algorithm 1. Then a downward closed subtree T rooted at v is not closed in D if there is a child node v' of v in the RP-tree such that $Occ_D(T)$ is transaction-matched with $OccList(v')$, where $Occ_D(T)$ is the occurrence set of T in D.*

A checking function based on Lemma 3 is proposed in Subsection 3.3.

3.2 Constructing the global trees

This subsection describes the construction of the so-called *RG-tree* which is the representation of all subtrees having the same root node. We only consider the database of attribute trees [*1]. A node v of an *RG-tree* is also assigned a set of all occurrences of v in D denoted by $OccList(v)$.

The *RG-tree* for a root node r with occurrence set $OccList(r)$ is constructed from the top to the bottom. First, the node r is made as the root of the *RG-tree*. For a node v already been constructed, we scan for child nodes of v at all occurrences of v in D. Nodes with same label are grouped and made as a child node of v in the *RG-tree*. Suppose w is a child node of v in the *RG-tree*, for an occurrence occ of w in some tree T, we add the corresponding occurrence o of the root node r to the *OccList* of w if there exists a path from o to occ in T.

The *RG-tree* for the root node C $\{(T_0, 1), (T_1, 0), (T_2, 1), (T_2, 6)\}$ is shown in Figure 3. For the brevity, four occurrences are rewritten as O_0, O_1, O_2^0, and O_2^1, respectively.

3.3 Mining from the global trees

The proposed method works in a level-wise manner starting from the root of the *RG-tree*. First, downward closed subtrees are generated and those which are not closed in

[*1] A tree is an *attribute tree* if any two sibling nodes do not have the same label.

Table 1: A Transposed database

Occurrence	Tid	Items
O_0	0	v_1 v_2 v_4 v_6
O_1	1	v_1 v_2 v_4 v_6
O_2^0	2	v_1 v_2 v_4
O_2^1	3	v_2 v_4

```
Input: a RG-tree $G$ with root $r$, a minimum
       support $minsup$
Output: all frequent closed subtrees of the RG-tree
1 $T \leftarrow r$;
2 $C \leftarrow$ all children of $r$ in the tree $G$;
3 $out \leftarrow \emptyset$;
4 MineClosed($G, minsup, T, C, out$);
5 return $out$;
```
Algorithm 2: Mining from *RG-tree*

D will be pruned out. Given an *RG-tree* G, each node of G is numbered according to the index of the node in the preorder traversal of G. A subtree of G sharing the same root with G is called a *root subtree* of G. Given T, a root subtree of G, we define $L_i(T)$ as the set of all nodes at the depth i of T, $P_i(T)$ the set of all nodes in G that are parent nodes of nodes in $L_i(T)$, and $C_i(T)$ the set of all child nodes in G of nodes in $P_i(T)$.

For a set C of some nodes at the level i of a root subtree T of an *RG-tree* G, we define the transposed database of C (denoted by C_{trans}) as the following. Transactions of C_{trans} are the occurrences of nodes in C, and items in C_{trans} are the nodes in C.

The transposed database of the set of all nodes at level 1 of the *RG-tree* in Figure 3 is given in Table 1.

Lemma 4 *For a root subtree T of an RG-tree, let $CFI_i(T)$ be the set of all closed frequent itemsets in the transposed database of $C_i(T)$. If T is downward closed, then for every level i of T, $L_i(T) \in CFI_i(T)$.*

Lemma 4 suggests a method to mine all downward closed subtrees from a *RG-tree* based on the mining of closed frequent itemsets from the corresponding transposed databases. A root subtree is grown level by level with the starting level (level 0) containing only one node that is the root of the *RG-tree*. To obtain nodes at level $i+1$, a transposed database for child nodes of nodes at level i is created. Each closed frequent itemset becomes a candidate of set of nodes at level $i+1$ of the tree being grown. The method is given in Algorithm 2.

The operator $\oplus$ on Line 15 in the *MineClosed* procedure grows the subtree T one more level by adding all child nodes of $CIS[i]$ to T using the *parent-child* relationships of the *RG-tree G*. Line 13 of the MineClosed procedure ensures the subtree T is always generated after T' if $T \preceq T'$, and thus, we can apply the CheckClosed function to prune out T if it is not a downward closed subtree. The CheckClosed function also helps to prune out downward closed subtrees that are not closed in D by using a checking technique developed based on Lemma 3. The PruneOcc procedure deletes, from the *OccList* of nodes at level $i+1$, occurrences which do not appear in the occurrence set of nodes at level i so that all trees generated are frequent.

```
1  $C_{trans} \leftarrow$ Make_Transposed_DB ($C$);
2  $CIS \leftarrow$ all closed freq. itemsets mined from $C_{trans}$;
3  if $CIS = \emptyset$ then
4      if CheckClosed $(G, T, out) = true$ then
5          $out \leftarrow out \cup T$;
6      end
7  else
8      if $sup(T) > max_{I \in CIS}\ sup(I)$ then
9          if CheckClosed $(G, T, out) = true$ then
10             $out \leftarrow out \cup T$
11         end
12     end
13     sort itemsets in $CIS$ in size descending order;
14     for $i$ = 0 to $|CIS| - 1$ do
15         $T' \leftarrow T \oplus CIS[i]$;
16         $C' \leftarrow$ all child nodes in $G$ of nodes in $CIS[i]$;
17         PruneOcc ($C'$);
18         MineClosed ($G, minsup, T', C', out$);
19     end
20 end
```
Procedure MineClosed($G, minsup, T, C, out$)

```
1 if $\exists t \in out$ s.t. $t$ having the same support as $T$ then
2     return false;
3 end
4 $r \leftarrow root(G)$;
5 if $\exists v$ s.t. $v$ is a child node of $r$ in the RP-tree and
  $Occ(T)$ is transaction-matched with $OccList(v)$
  then
6     return false;
7 end
8 return true;
```
Function CheckClosed(G, T, out)

Figure 4 is an example of generating all closed subtrees from the *RG-tree* in Figure 3. Each subtree is assigned an index indicating the order in which the subtree is created. Subtrees inside the rectangles are closed and frequent.

4. Experiments

We conducted the experiments on three different real life datasets CSLOGS [*2], TREEBANK [*3] , and DBLP [*3]. To obtain the attribute trees, we pruned the datasets by removing all but the first occurrences of the repeated labels. All experiments are measured on a 2.4GHz Intel Core 2 Duo CPU with 2 GB of RAM.

*2 http://www.cs.rpi.edu/~zaki/software/
*3 http://www.cs.washington.edu/research/xmldatasets/

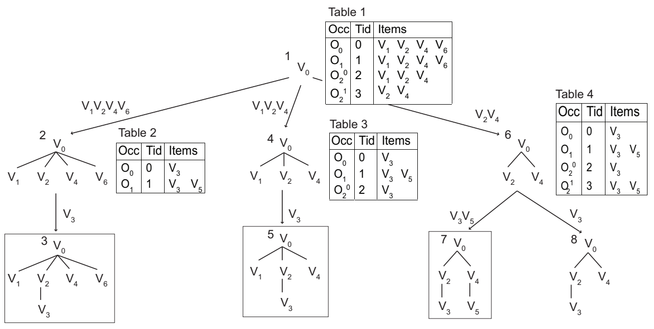

Figure 4: Mining from the *RG-tree* in Figure 3

Table 2: CMTreeMiner (CMT) versus SCCMiner (SCC)

Dataset	Runtime (sec)		# closed subtrees	
Minsup	CMT	SCC	CMT	SCC
CSLOGS 100	125	0.828	7,348	110
90	208	0.844	10,446	110
80	384	0.86	14,131	222
70	754	0.875	19,882	334
60	1,516	0.891	30,301	565
TREEBANK 900	308	1.265	7,879	73
700	363	1.531	11,666	115
500	451	1.922	19,497	197
300	610	3.01	41,171	468
100	1,281	8.688	186,665	2,988
DBLP 500	796	1.156	279	4
400	800	1.156	286	4
300	803	1.172	299	4
200	808	1.172	333	4
100	875	1.172	342	4

We first compared our algorithm called SCCMiner with CMTreeMiner [Chi 04], a very efficient closed tree patterns mining algorithm. For CMTreeMiner, we assume zero cost in the post-processing stage to obtain the closed subtrees satisfying the subtree constraint. The subtree constraint S is selected as the one that appears in 1% of all closed subtrees.

From Table 2, we can see that SCCMiner is very competitive in running time in all the settings. The SCCMiner is also efficient in the memory usage which is less then 10% of the memory used by CMTreeMiner in most cases and very stable as the minimum support goes down. Please note that we did not say CMTreeMiner is inefficient. CMTreeMiner is efficient to mine all closed tree patterns, whereas SCCMiner turns out to be more efficient when mining under subtree constraint.

Figure 5 shows the scalability of SCCMiner with respect to the constraint selectivity. For each dataset, we fixed the minimum support threshold value and varied the subtree constraint S of different selectivity of 1%, 2%, 4%, 8%, and 16%. As can be seen from the figure, SCCMiner achieves a linear scalability with the constraint selectivity for all datasets.

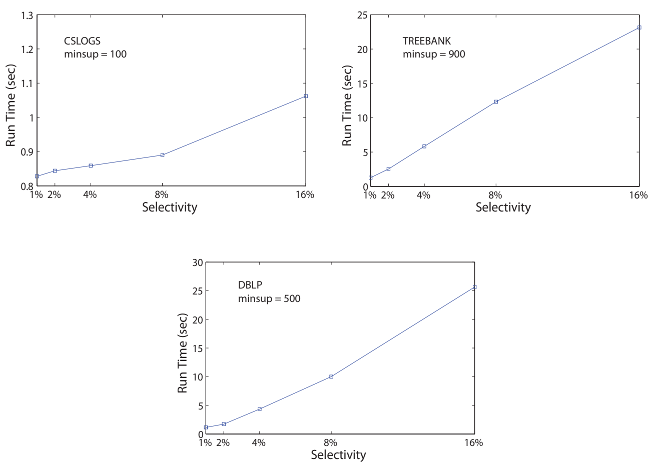

Figure 5: Scalability with constraint selectivity

One of the most interesting aspects of the SCCMiner algorithm is it can find vary rare patterns (patterns of very low support value e.g., 0.003%, which corresponds to a subtree occurring in only 2 transactions in the CSLOGS dataset) within a second.

5. Conclusion

In this paper we have proposed an algorithm for mining unordered closed tree patterns with subtree constraint which allows the user to give a partial specification of patterns of interest from an attribute tree database. The subtree constraint leads to less but more interesting mining results to the user. Our experimental results show the effectiveness of the algorithm developed. The proposed algorithm can be easily extended to mine with multiple subtree constraints and other monotone constraints. Further research can be defining similar constraints and developing similar algorithms for general tree databases and graph databases as well.

References

[Asai 02] Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., and Arikawa, S. Efficient Substructure Discovery From Large Semi-structured Data, in *Proc. the Second SIAM International Conference on Data Mining (SDM2002)*, pp. 158-174, (2002).

[Chi 04] Chi, Y., Yang, Y., Xia, Y., and Muntz, R.R. CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees, in *Proc. the Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, pp. 63-73, (2004).

[Zaki 02] Zaki M.J. Efficiently Mining Frequent Trees in a Forest, in *Proc. the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD02)*, pp.71-80, (2002).