PCITMiner – Prefix-based Closed Induced Tree Miner for finding closed induced frequent subtrees

Sangeetha Kutty Richi Nayak Yuefeng Li

Faculty of Information Technology
Queensland University of Technology
GPO Box 2434, Brisbane Qld 4001, Australia

{kutty,r.nayak,y2.li}@qut.edu.au

Abstract

Frequent subtree mining has attracted a great deal of interest among the researchers due to its application in a wide variety of domains. Some of the domains include bio informatics, XML processing, computational linguistics, and web usage mining. Despite the advances in frequent subtree mining, mining for the entire frequent subtrees is infeasible due to the combinatorial explosion of the frequent subtrees with the size of the datasets. In order to provide a reduced and concise representation without information loss, we propose a novel algorithm, PCITMiner (Prefix-based Closed Induced Tree Miner). PCITMiner adopts the prefix-based pattern growth strategy to provide the closed induced frequent subtrees efficiently. The empirical analysis reveals that our algorithm significantly outperforms the current state of the art algorithm, PrefixTreeISpan(Zou, Lu, Zhang, Hu and Zhou 2006b).

Keywords: Frequent subtree mining, closed, induced trees, subtrees, frequent mining

1 Introduction

Recently, there have been an increasing number of researches in frequent subtree mining due to the simplicity of the mining process and the potential of its application in various domains. Some of the domains include bio informatics, XML processing, database management, and web usage mining (Tatikonda, Parthasarathy and Kur 2006). Additionally, frequent subtree mining serves as the kernel function for other data mining techniques such as association rules mining, classification and clustering.

A number of algorithms have been proposed to extract frequent subtrees efficiently from a given tree dataset. However, there exists an immense disadvantage faced by these algorithms. For instance, there are often situations in which the number of frequent subtrees increases exponentially with the size of the tree dataset causing difficulties for the end-user to analyse the results (Chi, Yang, Xia and Muntz 2004b).

Due to the overwhelming number of frequent subtrees, the frequent subtree mining algorithms fail to provide a complete output. In order to provide a feasible solution as well as to improve the performance, the frequent subtree mining algorithms have focused on generating a concise but lossless representation of frequent subtrees. Two such popular representations of frequent subtrees are closed and maximal representations. One popular technique for generating the closed and maximal frequent subtrees is the CMTreeMiner (Chi et al. 2004b). However, this algorithm employs the "*generate-and-test*" technique to generate the closed frequent subtrees. Popularized by apriori-based algorithms(Agrawal, Mannila, Srikant, Toivonen and Verkamo 1996), the *generate-and-test* technique basically involves two processes namely the candidate generation from the smaller sized frequent subtrees and then the testing of the candidate frequent subtrees against the tree dataset. Unfortunately, the *generate-and-test* technique is an expensive operation when the numerous candidate frequent subtree checks are required (Termier, Rousset, Sebag, Ohara, Washio and Motoda 2005).

On the other hand, studies on the frequent itemset and sequential mining have clearly demonstrated that the pattern-growth technique provides improved performance in comparison to the *generate-and-test* technique to generate the frequent itemsets or sub-sequences(Pei 2002; Han, Pei and Yan 2005)especially on dense datasets. Hence, in this paper we propose an efficient algorithm called *PCITMiner* (**P**refix-based **C**losed **I**nduced **T**ree Miner), which combines the strengths of pattern growth technique and the closure property to discover the frequent closed induced subtrees. The experimental results indicate that our proposed approach outperforms the base algorithm *PrefixTreeISpan(Zou et al. 2006b)* by resulting in a reduced number of frequent subtrees and with an improved time performance in generating the frequent induced subtrees. The experimental results also manifests that our proposed approach is a better performing algorithm for heavily branched trees.

To the best of our knowledge, there exists no subtree mining algorithm using the pattern growth technique to identify the closed frequent induced subtrees. Hence, in this paper we develop the novel algorithm *PCITMiner* (**P**refix-based **C**losed **I**nduced **T**ree Miner) using the pattern growth technique for providing the closed frequent induced subtrees in a computationally efficient manner.

The rest of the paper is organized as follows. In Section 2, we will be presenting the background of the frequent subtree mining and the problem definitions. Section 3 contains a review of the frequent subtree mining algorithms. Section 4 details the description of *PCITMiner*, while Section 5 presents a performance comparison of running the PCITMiner algorithm against an implementation of PrefixTreeISpan algorithm(Zou et al. 2006b). PrefixTreeISpan was chosen as the baseline as it is the state-of-the-art algorithm in the area of frequent subtree mining using the pattern-growth technique.

2 Background concepts and Problem Definition

Before, explaining about the frequent subtree mining process we will look into what is meant by trees and the types of trees and subtrees.

A tree is denoted as $T = (V,v0, E, f)$, where V is the set of nodes; $v0$ is the root node which does not have any edges entering into it; E is the set of edges in the tree T; f is a mapping function $f: E \rightarrow V \times V$.

There are different types of trees namely *free trees* or *rooted trees*, *ordered* or *unordered trees* and *labelled* or *unlabelled trees*. If a given tree T has a root node, $v0$, then T is called as a *rooted tree* otherwise a *free tree*. An *ordered tree* is a tree which preserves a pre-defined ordering such as left-to-right among the set of nodes. Finally, if a tree T has labels for its edges then T is a *labelled tree*. The proposed technique will be applied on the *labelled rooted ordered trees.*

In the frequent mining of trees, it has been noted that often the entire tree will not be frequent rather there is a good possibility that parts of the tree are frequent. The parts of such trees are referred to as *subtrees*. A tree T' is a subtree of T if there exists a subtree isomorphism from T' to T. This implies that there is a one-to-one mapping from the vertices of T' to the vertices of T and it preserves the vertex labels, edge labels and adjacency then T' is a subtree of T. There exist different perspectives about subtrees and the two popular types of subtrees are the induced and embedded subtrees(Chi, Nijssen, Muntz and Kok 2005).

Embedded subtree

For a tree T with an edge set E and a vertex set V, a Tree T' with a vertex set V' and an edge set E' is an embedded subtree of T iff (1) $V' \subseteq V$; (2) $E' \subseteq E$; (3) the labelling of nodes of V' in T' is preserved in T; (4) $(v1,v2) \in E$ where $v1$ is the parent of $v2$ in T' iff $v1$ is the parent of $v2$ in T; and (5) for $v1,v2 \in V'$, preorder($v1$) < preorder($v2$) in T' iff preorder($v1$) < preorder($v2$) in T. An embedded subtree T' preserves the ancestor-descendent relationships among the vertices of the tree, T.

Induced subtree

For a tree T with an edge set E and a vertex set V, a Tree T' with a vertex set V' and an edge set E' is an induced subtree of T iff (1) $V' \subseteq V$; (2) $E' \subseteq E$; and (3) the labelling of the nodes of V' and E' in T' is preserved in T. An induced subtree T' is a subtree which preserves the parent-child relationships among the vertices of the tree, T.

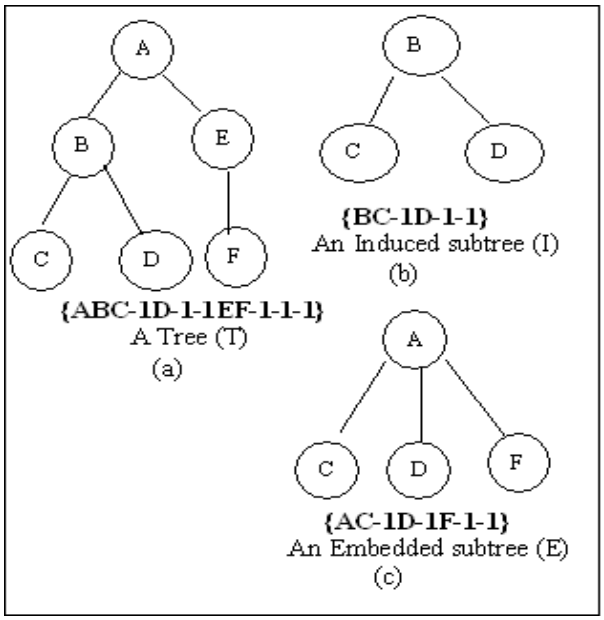


Figure 1: (a) A Tree, T (b) Induced subtree, I (c) Embedded subtree, E with their respective pre-order string in curly braces

For a given Tree, T (in Figure 1(a)), Figures 1(b) and 1(c) show the induced and embedded subtrees of T. It can be seen that the induced subtree, I preserves the parent-child relationships, and the embedded subtree, E preserves only the ancestor-descendent relationships. Hence, though A is not the parent of C, D and F, the embedded subtree, E has A as the root node as it is the ancestor of the three nodes C, D and F.

The curly braces in Figure 1 below the Tree, T and the subtrees, I and E, are their respective pre-order string format (as defined in (Chi et al. 2005)). The pre-order string format represents the pre-order traversal of a tree in a string like format where every node has a "–1" as its end flag. For a rooted ordered tree T with only one node r, the pre-order string of T is $S(T) = l_r –1$ where l is the label of the root node r. On the other hand, for multiple nodes for the rooted ordered tree T, where r is the root node and the children nodes of r are $r_1,...,r_k$ preserving left to right ordering. Then the pre-order string for T is $S(T)= l_r\ S(T_{r_1}) \ldots S(T_{r_k})\text{-}1$.

Having explained the background concepts about trees and subtrees, we will now detail the frequent subtree mining problem.

Problem definition for the frequent subtree mining

Given a tree dataset $D = \{T_1, T_2, T_3, \ldots, T_n\}$ with n number of trees, there exists a subtree $T' \subseteq T_k$ preserving the relationships (either the parent-child relationship or the ancestor-descendent relationship) among the nodes as that of the tree T_k. Support(T') (or frequency(T')) is defined as the percentage (or the number) of trees in D where T' is a subtree. A subtree T' is frequent if its support is not less

than a user-defined minimum support threshold. In other words, T' is a frequent subtree of the trees in D such that (*frequency* $(T')/|D|) \geq min\_supp$, where *min_supp* is the support threshold and $|D|$ is the number of trees in the tree dataset D.

Mining for the frequent induced subtrees from a huge tree dataset causes difficulties in analysing the result due to the combinatorial explosion in the number of frequent subtrees generated at lower support, consequently the frequent subtree mining algorithms become intractable. Hence, it is essential to control the number of subtrees generated. In order to reduce the number of subtrees without any information loss, two concise representations of frequent subtrees namely maximal and closed were proposed(Chi et al. 2004b).

Problem definition for Closed and Maximal subtree

In a given tree dataset, $D = \{T_1, T_2, T_3, ..., T_n\}$, if there exists two frequent subtrees T' and T'', T' is said to be maximal of T'' iff $\forall\ T' \supseteq T''$, $supp(T') \leq supp(T'')$; and a frequent subtree T' is closed of T'' iff for every $T' \supseteq T''$, $supp(T') = supp(T'')$. The latter property is called as *closure*. Based on the definition, it can be said that $M \leq C \leq F$, where

M = Number of Maximal frequent subtrees

C = Number of Closed frequent subtrees

F =Number of Frequent subtrees

Let us analyse the two concise representations, maximal and closed frequent subtrees. Firstly, we will apply closure on frequent subtrees generated from a given tree dataset D. Before that, we will define the frequent subtrees for a given *min_supp* of k. Let us assume that the frequent subtree mining result set $O = \{T'_1, T'_2, T'_3\}$ contains three frequent subtrees having a support of k, $k+1$ and k respectively. Also consider, $T'_1 \subseteq T'_3$, $T'_2 \subseteq T'_3$ and T'_3 does not have any superset. Applying the definition of the closed frequent subtrees on the frequent subtrees result set, O, it is found that T'_1 and T'_3 have the same support and $T'_3 \supseteq T'_1$. As a result, T'_1 is not closed and it can be removed from the output as its superset, T'_3, includes the information contained in T'_1. Also, there exists no superset of T'_3 therefore T'_3 is closed. Hence, T'_2 and T'_3 are the two closed frequent subtrees.

On the contrary, let us check whether T'_2 and T'_3 are maximal or not. We have not included T'_1 as the number of maximal frequent subtrees is less than the number of closed frequent subtrees and hence T'_1 cannot be maximal as it is not closed. According to the definition of maximal frequent subtrees, T'_3 is the maximal frequent subtree due to the reason that $T'_3 \supseteq T'_2$. There is only one maximal frequent subtree, which is a reduced number in comparison to the closed subtrees (that is two). The total number of output patterns are less considering the maximal frequent subtree representation, however, this representation suffers from information loss. Considering the above example, the frequent subtree T'_2 has a support of k+1 which implies that T'_2 occurs more than T'_3 but, based on the final output maximal pattern, it can be inferred that the support of T'_2 is k as the extra occurrence information has not been included in the output.

Therefore, the comparison of these two concise representations reveals that though the maximal frequent subtree representation provides reduced pattern set it results in information loss. Alternatively, the closed frequent subtree representation provides a concise pattern set without any information loss as the closure property eliminates only the redundant information. This reason is attributed for the popularity of closed frequent subtrees over the maximal frequent subtrees.

3 Related Research

Research on the frequent subtree mining spans from the frequent mining on different types of subtrees namely induced and embedded using various performance tuning techniques to the recent researches focusing on efficiently generating concise representations such as closed and maximal.

The seminal work on the embedded subtrees were conducted by the TreeMinerV(Zaki 2005), TreeMinerH(Zaki 2005) to generate the frequent embedded subtrees. Some of the earlier works on the frequent induced subtree mining include TreeFinder(Termier, Rousset and Sebag 2002), Freqt(Asai, Abe, Kawasoe, Arimura, Satamoto and Arikawa 2002), uFreqt(Asai, Arimura, Uno and Nakano 2003), HybridTreeMiner (Chi, Yang and Muntz 2004a), Unot(Asai et al. 2003) to generate the frequent induced subtrees.

The frequent subtree mining algorithms can be broadly classified into two groups namely (1) *generate and test* and (2) *pattern-growth* based on the strategy they adopt to identify the frequent subtrees. Algorithms such as TreeMiner(Zaki 2002) and Freqt(Asai et al. 2002) fall into the category of *generate-and-test* in which the candidate frequent subtrees are generated from the previous step and tested whether they are frequent or not using the tree dataset. These algorithms adopt a *level-wise* mining methodology where at each level the size of the newly discovered subtree is increased by one by joining or extending frequent subtrees generated from the previous level. For instance, the *(K+1)-Length* candidate frequent subtrees are generated by extending or joining the frequent *K–Length* subtrees where *K* refers to the number of levels in the subtree. A candidate *(K+1)-Length* frequent subtree is tested against the dataset to verify whether the candidate *(K+1)-Length* subtree is frequent or not. If it is frequent, then the *(K+1)-Length* subtree is included in the result set and it is used to generate *(K+2)-Length* subtree and this process is repeated until there are no more candidate subtrees that could be found.

The *Pattern-growth* strategy was adopted by Xspanner(Wang, Hong, Pei, Zhou, Wang and Shi 2004), Chopper(Wang et al. 2004), PrefixTreeISpan(Zou et al. 2006b) and PrefixTreeESpan(Zou, Lu, Zhang and Hu 2006a). These algorithms utilize the strategy of extending the discovered subtrees recursively until there are no frequent subtrees that could be found. As the extension of the discovered subtree is conducted by identifying the

extensions from the tree dataset, pattern-growth strategy does not involve any generation and testing of candidates. Though Xspanner and Chopper are included in the pattern-growth category, it adopts the *generate-and-test* techniques to generate candidates and hence it cannot be considered as a true candidate of the pattern-growth strategy. Experimental results of the frequent subtree mining algorithms such as PrefixTreeISpan utilizing the *pattern-growth* strategy shows improved runtime over the *generate-and-test* algorithms such as FreqT(Zou et al. 2006b).

With the increase in the dataset size there is an explosion in the number of frequent subtrees generated. In order to reduce the number of subtrees without any information loss, two popular concise representations namely maximal and closed were proposed. Some of the algorithms which generate these concise representations are PathJoin (Xiao and Yao 2003) and CMTreeMiner(Chi et al. 2004b). PathJoin (Xiao and Yao 2003) utilize a compact data structure called FST-forest to generate only the maximal frequent subtrees. PathJoin (Xiao and Yao 2003) faces a serious disadvantage as it does not utilize the maximal subtree generation to improve the performance. This is due to the fact that the pruning of frequent subtrees is applied as a post-processing step. On the other hand, CMTreeMiner(Chi et al. 2004b) was proposed to generate both closed and maximal frequent structures efficiently. As indicated by (Termier et al. 2005) that CMTreeMiner fails to produce satisfactory results for trees that have the high branching factor.

A parallel field to closed frequent subtree mining is the closed frequent subsequences. These closed frequent subsequences are extracted by mining sequential datasets. Some of the popular closed frequent sequence mining algorithms are BIDE(Wang and Han 2004) and CloSpan(Yan, Han and Afshar 2003). In contrast to sequences, trees have branches and hence the techniques to employ the closure property for sequential patterns could not be applied to trees. The sequences include extensions only in either the forward or backward direction in the same branch, it is difficult to utilize sequential pattern mining techniques to trees as it contains several branches.

It is evident from the previous research works that the *generate and test* algorithms are not an efficient vehicle for exploiting the full strength of closure. Additionally the existing techniques for itemsets and sequential patterns frequent mining cannot be naturally extended to deploy the closure property due to the structure of the trees. The impressive performance improvement obtained for the pattern-growth based techniques, PrefixTreeISpan and PrefixTreeESpan, as reported in (Zou et al. 2006b; Zou et al. 2006a) over the generate-and-test based techniques, FreqT and TreeMiner, inspired us to investigate the effect of deploying closure on the pattern-growth technique.

In this paper, we will be focusing only on the frequent induced subtrees and not on the emedded subtrees. The reason for this choice is two fold: Firstly, the application of the closure property on the frequent embedded subtrees will not reduce the result set significantly as the ancestor-descendant relationship is maintained in the frequent embedded subtrees. Secondly, the application of the closure property incurs a huge overhead due to the numerous closure checks required. Hence, in this paper we will focus only on generating the closed frequent induced subtrees using the *pattern-growth* technique. The following section details about the description of our proposed method PCITMiner (**P**refix-based **C**losed **I**nduced **T**ree Miner).

4 PCITMiner - Closure using Prefix Pattern Growth

In this section, we introduce PCITMiner for mining the closed induced subtrees from a tree dataset, *D*. Before we go into the details of PCITMiner, we will briefly introduce how the frequent induced subtrees are generated using the pattern growth technique.

Tree Id	Pre-order string of Trees
1	ABC-1D-1-1EF-1-1-1
2	ABC-1-1D-1-1
3	ABC-1-1C-1-1
4	ABC-1-1C-1-1

Table 1: Tree dataset

We will be using the tree dataset, *D* provided in Table 1 as our running example in this paper. The tree dataset, *D* contains Tree Ids and the trees which are represented in the pre-order string format as defined in Section 2. The Tree Id in Table 1 is the unique id for the trees in *D*. In this tree dataset, D, we have four trees and they contain the nodes labelled as A, B, C, D, E and F. The pictorial representation of the tree with Tree Id 1 is illustrated in Figure 2(a). This tree dataset is mined for the frequent subtrees with a *min_supp* of 2.

4.1 The frequent subtree generation

As we are using the pattern growth technique for the frequent subtree generation, we will explain the prefix-based pattern growth technique where the patterns are subtrees. There are three phases involved in the prefix-based frequent subtree generation and they are:

1. The *1-Length* frequent subtree generation
2. Projecting the dataset using prefix trees
3. Mining the projected instances dataset

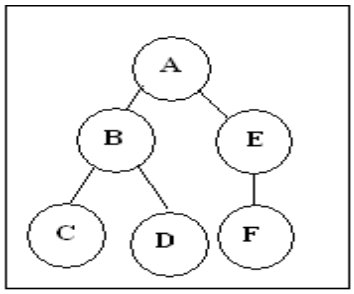
4.1.1 The *1-Length* frequent subtree generation

The prefix-based subtree growth technique starts with a scan of the tree dataset, *D* to determine the *1-Length* frequent subtrees. A *1-Length* frequent subtree contains only one node and is represented using the following format $Sub_X = (<X^a – 1>: Supp)$ where *X* represents the node label, the superscript '*a*' specifies the position in the pre-order traversal of the subpattern Sub_X, '*–1*' is used for the end flag for the node labelled *X,* and *Supp* is the support of the subtree. For a *K-Length* subtree, the representation can be obtained by replacing *X* in the Sub_X with the pre-order traversal and including a superscript for the pre-order positions. Hence the subtree representation is $(<X^a\ Y^b\ Z^c\ -1 \ldots K^n\ -1\ -1>: Supp)$ where

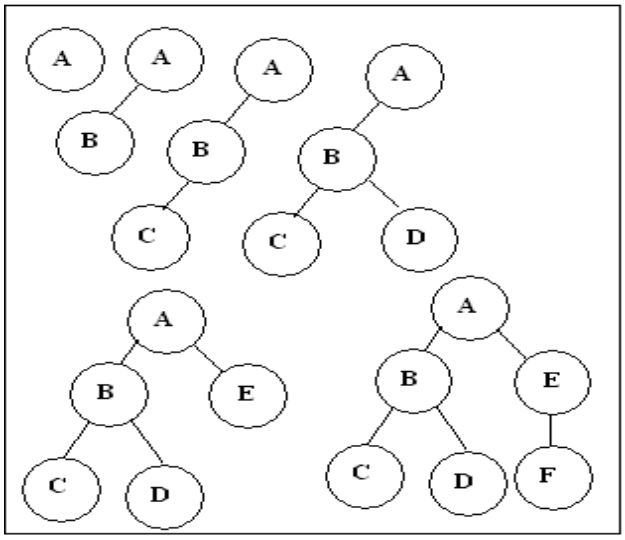
X, *Y*, *Z* and *K* are node labels and superscripts *a, b, c...n* are the increasing positions of nodes in the pre-order traversal of the subtree.

By scanning the tree dataset, D in Table 1 for a given *min_supp* of 2, the four *1-Length* frequent subtrees generated are (<A^1-1>:4), (<B^1-1>:4), (<C^1-1>:4), (<D^1-1>:2) where A, B, C and D are node labels. The superscript '1' on the node labels represents its position in the pre-order traversal of each of the subtrees. The '–1' in each of the subtrees represent the end flag for the node. Finally, the number after a ':'(colon) gives support of each of the subtrees. The subtrees having node labels A, B or C individually has a support of 4 and the subtree having node labelled D has a support of 2.

The tree dataset, D is scanned again and partitioned into four subsets using each of the four subtrees serving as the prefix-tree. Using the definition of the prefix-tree in (Zou et al. 2006b) we will explain what is meant by a prefix-tree using the first tree in our example dataset D from Table 1.



(a)



(b)

Figure 2: Prefix Trees of Tree T (in (a))

Definition 1 (Prefix-Tree)

Let there be a tree *T* with *m* nodes, *T′* be a tree with *n* nodes, where $n \leq m$. The pre-order scanning of tree *T* from its root until its *n-th* node results in a tree *T′*. If the tree *T′* is isomorphic to tree *T* then *T′* is called the prefix of Tree *T* (Zou et al. 2006b).

The Figure 2(b) shows the prefix-trees for the tree T illustrated in Figure 2(a). The 6 prefix-trees containing 1, 2, 3, 4, 5 and 6 nodes are identified for the tree T.

4.1.2 Projecting the dataset using the prefix trees

The next step in this process involves projecting the dataset using the prefix trees generated. Consider the example tree dataset, *D* in Table 1, there are four prefix-trees having a single node. Using these four 1-node prefix-trees, the dataset *D* is partitioned into four prefix-projected datasets namely (i) <A^1 -1 >:4) (ii) (<B^1-1>:4) (iii) (<C^1-1> :4) and (iv) (<D^1-1>:2). To build the prefix-projected dataset for a given prefix-tree *T′*, every tree in *D* is checked whether it contains the prefix-tree *T′*. If a tree *T* contains the prefix-tree *T′*, then the projected instance of T is included in the *T′*-prefix-projected instances dataset.

Definition 2 (The Prefix projected instance)

Consider a tree dataset $D = \{T_1, T_2, T_3, \ldots, T_n\}$ and a prefix subtree *T′* with *n* nodes. If there exists a tree $T_x \in D$ with *m* nodes which contains the prefix-tree *T′*, then the *T′*-prefix projected instance of T_x is the pre-order scanning of T_x from $n+1$ node to *m*.

Tree Id	Pre-order string of Trees
1	BC-1D-1 -1
2	BC-1-1 D-1
3 3	BC-1-1 C-1
4 4	BC-1-1 C-1

Table 2: <A^1-1> projected instances dataset

Tree Id	Pre-order string of Trees
1	C-1D-1
2	C-1
3	C-1
4	C-1

Table 3: <B^1-1> projected instances dataset

Tree Id	Pre-order string of Trees
1	C-1 D-1
2	C-1
3	C-1
4	C-1

Table 4: <A^1B^2-1-1> projected instances dataset

Tree Id	Pre-order string of Trees
1	C-1
2	C-1

Table 5: <A^1B^2-$1C^3$-1-1> projected instances dataset

Tables 2, 3, 4, and 5 provide the projected instances dataset of the prefix-trees <A^1-1>, <B^1-1>, <A^1B^2-1-1>, <A^1B^2-$1C^3$-1-1> respectively. To improve the efficiency,

the projected instances from the infrequent *Length-1* subtrees are eliminated. It can be noted in Table 2 that the tree with Tree Id1 does not contain the nodes E and F as they are infrequent and hence they were eliminated in projection. The generated projected instances are mined using the technique detailed in the following subsection.

4.1.3 Mining the projected instances dataset

As a next step in the prefix-pattern growth, each of the projected instances dataset is mined to identify the Growth Element (GE) (Zou et al. 2006b), which is defined as follows.

Definition 3: ***Growth Element (GE)***
Given two trees T' and T with m and m+1 nodes respectively, where T' is the prefix of T. If there occurs a node n in Tree T but not in T' then the node n is the Growth Element (GE) of T' w.r.to T.
If there is any *frequent* GE then the corresponding projection is partitioned and mined recursively until there are no more frequent GEs. For instance, for the partitioned dataset $<A^1$ -1 > provided in Table 2, the GEs are nodes labelled B, C and D as they occur as first nodes in the projected instance. The support of GEs, B, C and D is 4, 2 and 1 respectively in the $<A^1$ –1 > prefix-projected dataset. Hence only B and C are frequent GEs. Since the mining process outputs the induced subtrees, the position of nodes is important in counting the support. For example, the node labelled D occurs twice in the dataset in Table 2 but it occurs in different positions that is why it is not frequent. In other words, the subtrees should have parent-child relationship. In Tree Id 1, the parent of D is B and in the second tree (Tree Id 2), the parent of D is A. Hence, the support of D is 1 in the $<A^1$ –1 > prefix-projected dataset. Using the two GEs, B and C, two separate projections are constructed and mined for the frequent subtrees. Tables 4 and 5 give the projection for $<A^1B^2$ –1 –1 > and $< A^1B^2$-$1C^3$-1-1> respectively.

4.2 Closure

So far we have seen how the frequent subtrees are generated using the prefix-pattern growth technique. Table 6 lists the frequent subtrees using this approach. It can be seen from Table 6 that subtrees such as ($<A^1$-1 >:4), ($<B^1$-1>:4), ($<C^1$-1>:4), ($<A^1B^2$-1 –1>:4), ($<B^1C^2$-1-1>:4) are subsets of ($<A^1B^2C^3$-1-1-1>: 4) with the same support. Hence, instead of generating all the frequent subtrees, only a superset of the frequent subtrees with the same support can be represented as output. By doing so, the number of the frequent induced subtrees is reduced by eliminating only the redundant frequent subtrees and hence there is no information loss. This property of reducing the redundant frequent subtrees is called as the closure property as discussed in Section 2. From Table 6, the subtree ($<A^1C^2$-1-1>: 2) $\subseteq$ ($<A^1B^2C^3$-1 -$1C^4$-1-1>: 2), and hence using the closure property the subtree ($<A^1C^2$-1-1>: 2) can be safely removed from the result set. As the node labelled D is not included in other closed frequent induced subtrees, subtree ($<D^1$-1>:2) is included in the output.

Number of nodes	Frequent Subtrees
1	($<A^1$-1 >:4), ($<B^1$-1>:4), ($<C^1$-1>:4), ($<D^1$-1>:2)
2	($<A^1B^2$-1 –1>:4), ($<B^1C^2$-1-1>:4), ($<A^1C^2$-1-1>:2)
3	($<A^1B^2C^3$-1-1-1>: 4) ($<A^1B^2$-$1C^3$-1-1>: 2)
4	($<A^1B^2C^3$-1 -$1C^4$-1-1>: 2)

Table 6: Frequent induced subtrees from prefix-pattern growth algorithm

Table 7 summarizes the closed frequent induced subtrees with only 3 closed frequent induced subtrees in comparison to 10 frequent induced subtrees (as shown in Table 6). On comparing Tables 6 and 7 it is interesting to note that closure has reduced the number of frequent induced subtrees by three-fold.

Number of nodes	Frequent Subtrees
1	($<D^1$-1>:2)
3	($<A^1B^2C^3$-1-1-1>: 4)
4	($<A^1B^2C^3$-1 -$1C^4$-1-1>: 2)

Table 7: Closed frequent induced subtrees

Now the challenge is to impose closure on the frequent induced subtrees using the prefix-based subtree mining. A naïve approach to impose closure is to first generate all the frequent induced subtrees and then eliminate the subtrees based on their support by checking the closure property, as shown in Tables 6 and 7. It is an expensive task when there are a large number of frequent subtrees generated and hence, it is essential to identify an efficient method, which provides the closed result set. There are a number of approaches proposed in the frequent itemset and sequential mining (Wang and Han 2004; Yan et al. 2003). Unlike, the itemset or sequential mining, trees have branches and hence we cannot apply closure using these techniques. Hence, we propose two methods to apply closure efficiently and they are:

1. Search Space reduction using the backward scan
2. Bi-directional Extension Closure checking

4.2.1. Search space reduction using the backward scan

This technique does a backward scan to reduce the search space using the following lemma:

Lemma 1:
Let there be two *1-length* frequent subtrees L_k and L_k' in a given tree dataset, D. If L_k' is the parent node of L_k in all trees in D then the projection of L_k is stopped as the parent node L_k' will include all the subtrees generated using the prefix-tree L_k.
Using the running example tree dataset D in Table 1, we will explain, how to reduce the search space using the backward scan technique. This technique is applied after

the first scan of the dataset where the *1-Length* frequent subtrees are known. The *1-Length* frequent induced subtrees are (<A^1-1 >:4), (<B^1-1>:4), (<C^1-1>:4), (<D^1-1>:2). As the node labelled A is a root node in all the trees it is not checked for its parents. Hence, this technique is applied for the subtrees (<B^1-1>:4), (<C^1-1>:4) and (<D^1-1>:2).

The checking of the parent node of <B^1-1> in each of the trees in the tree dataset *D* reveals that the parent node is <A^1-1> in all the trees. This information state that the parent node of <B^1-1> (i.e. <A^1-1>) and <B^1-1> have the same support. Consequently, <B^1-1> can be pruned from growing since the projections for the parent node <A^1-1> will include the projections for <B^1-1>. By doing so, the number of subtrees and the number of projections required are reduced. Due to the reduced search space, the efficiency of the algorithm is improved.

4.2.2. The Bi-directional Extension Closure checking

After reducing the search space using the backward scan, there occurs some of the subtrees which are not closed. In order to check the closeness of the generated frequent subtrees, the bi-directional extension closure checking is performed.

According to the definition of a frequent closed induced subtree, a prefix-tree, $T_p=e_1,e_2,...e_n$ is non-closed if there exist at least one extension event, e' which can be used to create a prefix-tree T_p' having the same support as that of T_p. The prefix-tree T_p can be extended in the following ways:

1. *Predecessor node extension as in* $T_p' = e_1,e_2,...,e_n\, e'$
2. *Internal node extension as in* $T_p' = e_1,e_2\, e',...,e_n$
3. *Successor node extension as in* $T_p' = e_1,e_2,...\, e',e_n$

The bi-directional extension closure checking involves two events namely the *forward-extension event* and the *backward-extension event*. With reference to the event *n* given by e_n, in the situation 1, e' occurs after the event e_n and hence it is a *forward extension event*. On the other hand, in the situation 2 and 3, e' occurs before the event e_n and hence it is a *backward extension event*.

Theorem 1:

If there exists neither the forward-extension event nor the backward extension event in regard to a prefix-tree T_p' then T_p' must be a closed frequent subtree.

A naive approach to check whether there occurs any forward-extension closure checking is enumerating all the frequent sub-trees and then checking their support. However, this is an expensive operation due to very large number of checks required. The following lemma is utilised to check for the forward-extension event efficiently.

Lemma 2: Forward-extension event

For a prefix tree T_p', its complete set of forward-extension events is equivalent to the set of its frequent GEs whose supports are equal to the support of T_p'. If any of the GE for given projection has same support as T_p' then T_p' is not closed.

Using the running example provided in Table 1, the GE is *C* for the prefix-tree <A^1B^2-1-1>. The support of <A^1B^2-1-1> is 4 and the support of the GE *C* is 4 and hence <$A^1B^2C^3$-1-1-1> is not closed. On the other hand, consider the prefix-tree <$A^1B^2C^3$-1-1-1> having the GE *C*. The support of <$A^1B^2C^3$-1-1-1> is 4 and the support of the GE *C* is 2 and hence <$A^1B^2C^3$-1-1-1> may be closed. We say <$A^1B^2C^3$-1-1-1> may be closed, as we need to check for closure using the backward extension event to confirm the closure. This forward event checking is not a computational expensive step and hence it is used for reducing the number of closed frequent induced subtrees. In order to check for the backward extension event, the following lemma is used.

Lemma 3: Backward-extension event

If there exists a prefix-tree T_p with *m* nodes and a prefix-tree T_p' with the common *m* nodes and an additional node *b* having the same support as that of T_p then T_p is not closed and *b* is a backward extension event w.r.to T_p

There are two types of backward-extension events:

1. The parent extension of GE
2. The sibling extension of GE

The backward extension event to GE is the extension of the parent of GE and hence it is handled by the backward scan technique. On the other hand, the backward extension to sibling extension is the extension of sibling nodes of GEs. For instance, in the prefix-tree <A^1B^2-1C^3-1-1>, the sibling extension event is the node labelled C, which is an extension of the node <B2-1> and it is in a different branch resulting in <$A^1B^2C^3$-1C^4-1-1-1>. Unlike the sequential mining, due to the existence of branches in trees, there occurs the sibling extension event in a different branch from <A^1B^2-1C^3-1-1>. Hence, it requires the closure checking across several branches.

In order to efficiently check for closure for backward extension events across several branches, a technique called "maintain-and-test" is deployed to check for closure. A naïve approach to check for closure is to check for all the backward extension events having the same support. However, it is an expensive operation and hence to reduce the number of checks, a parameter, which is the sum of the tree ids, is included to check for closure. To apply this technique, we first check whether for a given subtree *T'*, there exists a backward extension to edge E resulting in *T''* having the same support and sum of tree ids as *T'*. If it exists then they are checked for closure.

Figures 3 and 4 outline the algorithm PCITMiner and the subroutine ***Fre*** respectively. PCITMiner starts with the scan of the database and identifies the *1-Length* frequent subtrees *b*. After finding the *1-Length* frequent subtrees, it employs the backScan property by checking the support of the predecessor of each *b* and the support of each *b*. If they are same, then the projection for *b* could be pruned, as the predecessor for *b* will include *b* in its output. Otherwise, using the recursive subroutine ***Fre*** outlined in Figure 4, recursively identifies all the occurrences of *b* in the dataset *D* to construct <*b-1*> projected database by collecting all the corresponding project-instances in *D*. The subroutine ***Fre*** checks for the forward extension event and the backward extension event against the projected database. This subroutine is recursively called until there are no more frequent GEs to form the projected dataset.

Algorithm PCITMiner

Input: A tree dataset *D*, minimum support threshold (*min_supp*)
Output: All closed induced frequent subtrees

Methods:
1. Scan *D* and find all *1-length* frequent label *b*.
2. For each frequent label *b*
   2.1 If the supp (predecessor of *b*) = = supp(*b*) then Do not project the dataset.
   2.2 else,
   2.2.1 Find all occurrences of *b* in dataset *D*, and construct <*b*−*1*>-projected dataset (i.e. *ProDB(D,<b − 1 >))* through collecting all corresponding Project-Instances in *D*.
   2.2.2 Call ***Fre*** *(< b − 1 >, 1, ProDB(D,< b − 1 >), min sup, supp(b))* to mine the projected dataset and obtain frequent induced subtrees until no more subtrees that could be found.

Figure 3: Algorithm PCITMiner

Function *Fre* (T_p, n, *ProDB(D,* T_p*), min_supp,prepat_supp)*

Input: A prefix-tree T_p, the length of T_p(n), <T_p>-projected dataset(*ProDB(D,*T_p*)*), the minimum support threshold (*min_supp*), the support of the previous pattern used to generate this projected dataset (*prepat_supp*)
Output: *C*: Closed frequent induced subtrees

Methods:
1. Scan *ProDB(D,* T_p *)* once to find all the *1-length* frequent *GEs*($GE_{0, \dots, k}$) according to Lemma 1.
2. output=true.
3. Count the support of all *GEs*.
4. If supp(GE_0 || GE_1,...,|| GE_k) == supp(T_p) then Do not output the subtree, output = false.
5. For each GE_b
   5.1 if GE_b is frequent then
      5.1.1 Extend T_p with *b* to form a subtree pattern T_p'.
      5.1.2 if (output) then
         Insert T_p' into *C*.
   5.2 else
      5.2.1 Check T_p' for occurrence of any of its subset with the same support and sum of tree ids in the output *C*. If there exists any subset for T_p' then remove the subset of T_p' and insert T_p' into *C*.
6. Find all occurrences of GE_b in *ProDB(D,* T_p*),* construct the <T_p'>-projected database (i.e. *ProDB(D,* T_p' *)*) through collecting all corresponding Project-Instances in *ProDB(D,* T_p*)*.
7. Call ***Fre***(T_p' , n + 1, *ProDB(D,* T_p'), *min_supp prepat_supp*)

Figure 4: Recursive function *Fre*

5 Experimental evaluation

All the experiments were conducted on the Intel Pentium-4 PC with 2.39GHz processor and 1GB main memory, running Windows XP. Both the algorithms PrefixTreeISpan and PCITMiner were written in C++ with the STL library support and compiled with the Microsoft Visual C++ .Net compiler. The experiments were conducted on the synthetic datasets generated.

The Zaki's treegenerator[1] has been often used to generate the synthetic datasets for benchmarking the tree mining algorithms. Using the Zaki's tree generator there are two datasets generated namely the F5 and D10 datasets with the parameters as indicated in Table 8 where "f" represents the fan out factor, "d" the depth of the tree, "n" the number of unique labels for the trees, "m" the total number of nodes in a parent tree and "t" indicates the number of trees.

Name	Description
F5	-f 5 -d 10 -n 100 -m 100 -t 100000
D10	-f 10 -d 10 –n 100 -m 100 -t 100000

Table 8: Datasets and their parameters

Studies have indicated that the performance of some of the existing closed frequent subtree mining algorithm degrades for datasets having a high branching factor (Termier et al. 2005). To evaluate the performance of PCITMiner with high branched trees, two datasets F5 and D10 with varied branches, fan out factors of 5 and 10 respectively, are generated.

The proposed algorithm, PCITMiner is compared with the prefix-based pattern-growth algorithm PrefixTreeISpan (Zou et al. 2006b) to show the benefit of closure. The output of the PrefixTreeISpan algorithm is frequent induced subtrees. Experimental studies on PrefixTreeISpan (Zou et al. 2006b) with FreqT (the *generate-and-test* based frequent subtree mining method) already has shown that PrefixTreeISpan outperforms FreqT (Zou et al. 2006b). So in this paper, we do not conduct any empirical analysis with the generate-and-test method. As the objective of this study is to apply closure on the pattern growth algorithms hence CMTreeMiner is also not used as a benchmark as the latter algorithm is based on the candidate "*generate-and-test*" approach.

Figures 5 and 6 presents the experimental results on the number of subtrees and the run time in seconds for PCITMiner and PrefixTreeISpan on the F5 dataset. Figure 5 reveals that the PCITMiner reduces the number of subtrees by about three-fold in comparison to PrefixTreeISpan. The benefit is larger for the relatively lower support-threshold (where a large number of subtrees are generated).

[1] http://www.cs.rpi.edu/~zaki/software

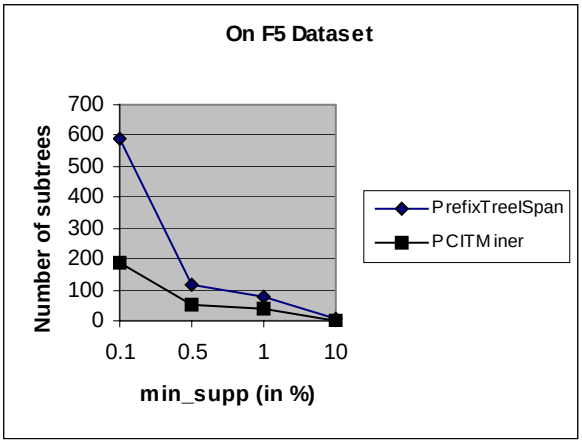


Figure 5: Number of subtrees of PCITMiner and PrefixTreeISpan against various min_supp on F5 dataset

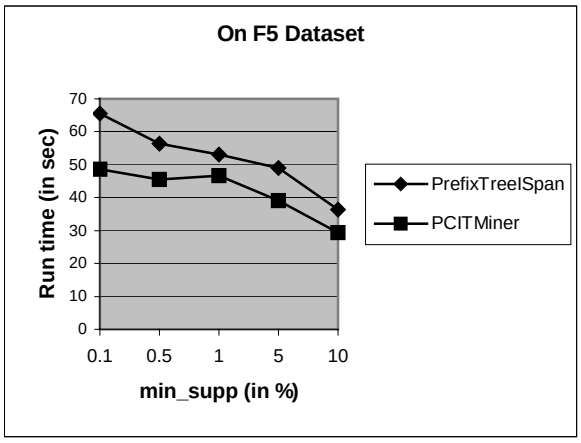


Figure 6: Run times of PCITMiner against PrefixTreeISpan against various min_supp on F5 dataset

Figure 6 reveals that with the reduced number of subtrees, PCITMiner mines the frequent subtrees faster than the base algorithm PrefixTreeISpan. The improvement obtained in F5 dataset can be attributed to the number of back scan pruning. Figure 7 shows the increase in the number of projections pruned with the reduced support threshold.

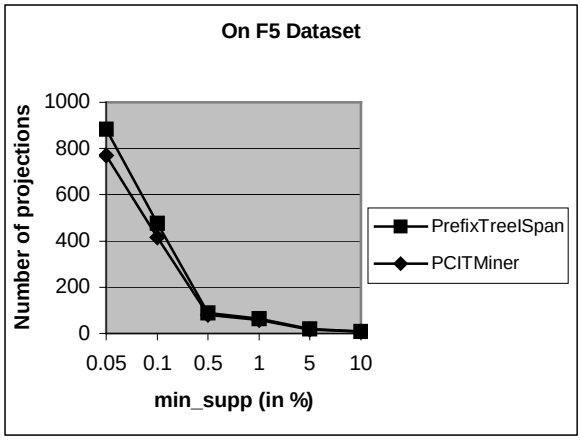


Figure 7: Number of projections of PCITMiner against PrefixTreeISpan against various min_supp on F5 dataset

Figures 8 and 9 presents the experimental results on the run time in seconds and the number of subtrees for PCITMiner and PrefixTreeISpan on D10 dataset respectively. The PCITMiner achieves improved performance over PrefixTreeISpan by reducing the number of subtrees by about seven-fold at lower support values.

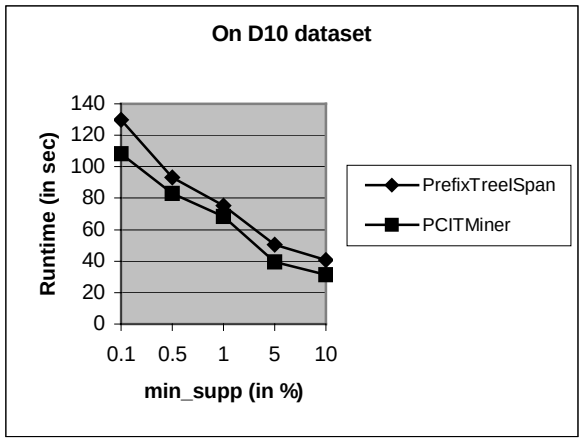


Figure 8: Run times PCITMiner against PrefixTreeISpan against various min_supp on D10 dataset

The comparison of experimental results of the F5 and D10 datasets clearly indicates that PCITMiner remains unaffected with high branched trees (large fan out factor). PCITMiner shows the improved performance in run time as well as reducing the number of subtrees efficiently in both the data sets. Moreover, the saving in terms of the number of output patterns is more apparent with the high-branched trees.

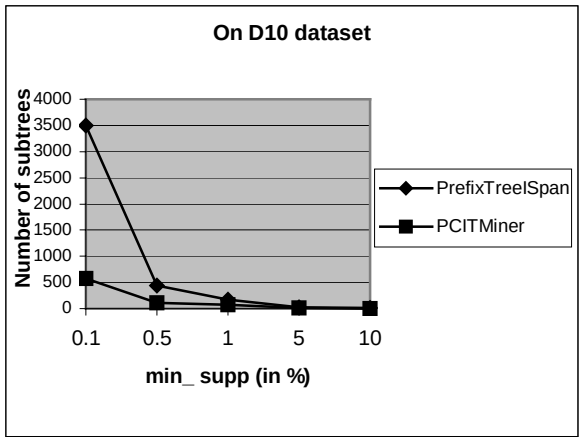


Figure 9: Number of subtrees of PCITMiner and PrefixTreeISpan against various min_supp on D10 dataset

6 Conclusions and Future work

In this paper, we have proposed PCITMiner for generating the closed frequent induced subtrees using the pattern-growth technique. The experimental results clearly indicate that PCITMiner performs faster and produces reduced number of frequent subtrees without any information loss. Contrary to the existing closed frequent subtree mining algorithms the proposed

algorithm PCITMiner performs efficiently for high branched trees. We would like to apply this closure technique for embedded subtrees and to graph-based frequent mining as a future work.

7 Acknowledgement

We would like to thank Lei Zou at HuaZhong University of Science and Technology for kindly providing us the base algorithm PrefixTreeISpan.

8 References

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo.(1996): Fast discovery of association rules. In *Advances in knowledge discovery and data mining, 1996.,*307-328: American Association for Artificial Intelligence.

Asai, T., K. Abe, S. Kawasoe, H. Arimura, H. Satamoto and S. Arikawa.(2002): Efficient substructure discovery from large semi-structured data. *2nd SIAM International Conference on Data Mining*.

Asai, T., H. Arimura, T. Uno and S. Nakano.(2003): Discovering Frequent Substructures in Large Unordered Trees. *The 6th International Conference on Discovery Science*.

Chi, Y., S. Nijssen, R. R. Muntz and J. N. Kok.(2005):Frequent Subtree Mining- An Overview. *Fundamenta Informaticae*, **66**: 161-198. IOS Press.

Chi, Y., Y. Yang and R. R. Muntz.(2004a): HybridTreeMiner: an efficient algorithm for mining frequent rooted trees and free trees using canonical forms. *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, 11-20.

Chi, Y., Y. Yang, Y. Xia and R. R. Muntz.(2004b): CMTreeMiner: Mining both closed and maximal frequent subtrees. In *In The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*.

Han, J., J. Pei and X. Yan.(2005): Sequential Pattern Mining by Pattern-Growth: Principles and Extensions. In *Foundations and Advances in Data Mining*.

Pei, J. 2002. *Pattern-growth methods for Frequent pattern mining*, School Of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada.

Tatikonda, S., S. Parthasarathy and T. M. Kur.(2006): TRIPS and TIDES: new algorithms for tree mining. *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management*, 455-464. Arlington, Virginia, USA: ACM.

Termier, A., M.-C. Rousset and M. Sebag.(2002): TreeFinder: a first step towards XML data mining. *ICDM 2002. Proceedings. 2002 IEEE International Conference on Data Mining, 2002.*, 450-457.

Termier, A., M.-C. Rousset, M. Sebag, K. Ohara, T. Washio and H. Motoda.(2005): Efficient mining of high branching factor attribute trees. *Proc. Fifth IEEE International Conference on Data Mining*.

Wang, C., M. Hong, J. Pei, H. Zhou, W. Wang and B. Shi.(2004): Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. In *Advances in Knowledge Discovery and Data Mining*.

Wang, J. and J. Han.(2004): BIDE: Efficient Mining of Frequent Closed Sequences. In *Proc. 20th International Conference on Data Engineering*: IEEE Computer Society.

Xiao, Y. and J.-F. Yao.(2003): Efficient data mining for maximal frequent subtrees. *Proc. Third IEEE International Conference on Data Mining(ICDM03)*, 379-386.

Yan, X., J. Han and R. Afshar.(2003): CloSpan: Mining Closed Sequential Patterns in Large Datasets. *Proc. SIAM International Conference on Data Mining*.

Zaki, M. J.(2002): Efficiently mining frequent trees in a forest. *Proc. Eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 71-80. Edmonton, Alberta, Canada: ACM Press.

Zaki, M. J.(2005):Efficiently mining frequent trees in a forest: algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, **17 (8)**: 1021-1035.

Zou, L., Y. Lu, H. Zhang and R. Hu.(2006a): PrefixTreeESpan: A Pattern Growth Algorithm for Mining Embedded Subtrees. In *Web Information Systems , WISE 2006*.

Zou, L., Y. Lu, H. Zhang, R. Hu and C. Zhou.(2006b): Mining Frequent Induced Subtrees by Prefix-Tree-Projected Pattern Growth. *Proc. Seventh International Conference on Web-Age Information Management Workshops, 2006. WAIM '06.*, 18.