Tree mining: Equivalence classes for candidate generation

Federico del Razo Lopez, Anne Laurent, Pascal Poncelet, Maguelonne Teisseire

Tree mining: Equivalence classes for candidate generation

F. Del Razo López[a], A. Laurent[b], M. Teisseire[b] and P. Poncelet[b]
[a]*Toluca Institute of Technology – ITT, Av. Instituto Tecnológico S/N – Col. Ex-Rancho La Virgen, Metepec, Edo. de México C.P. 52140, México*
[b]*University Montpellier 2 – LIRMM, 161, rue Ada, Montpellier, France*
E-mail: {laurent,teisseire, poncelet}@lirmm.fr

Abstract. With the rise of active research fields such as bioinformatics, taxonomies and the growing use of XML documents, tree data are playing a more and more important role. Mining for frequent subtrees from these data is thus an active research problem and traditional methods (e.g., itemset mining from transactional databases) have to be extended in order to tackle the problem of handling tree-based data. Some approaches have been proposed in the literature, mainly based on generate-and-prune methods. However, they generate a large volume of candidates before pruning them, whereas it could be possible to discard some solutions as they contain unfrequent subtrees. We thus propose a novel approach, called *pivot*, based on equivalence classes in order to decrease the number of candidates. Three equivalence classes are defined, the first one relying on a right equivalence relation between two trees, the second one on a left equivalence relation, and the third one on the ground of a root equivalence relation. In this paper, we introduce this new method, showing that it is complete (i.e., no frequent subtree is forgotten), and efficient, as illustrated by the experiments led on synthetic and real datasets.

Keywords: Data mining, tree mining, candidate generation, equivalent classes

1. Introduction

Tree data are playing a more and more important role for instance in the data exchanges (i.e. XML is now a standard format) or for scientific data which are often based on trees (e.g. biology). The extraction of frequent substructures, also known as *tree mining*, from huge tree databases is thus an active and important problem addressed by researchers from the data mining community [2,6,11,12,17]. In this context, a subtree is said to be frequent if it appears in more than σ trees from the database, where σ is called the minimal support. Frequent subtrees are very informative and they can be used to help the user or the system to be aware of what occurs in the database (for instance to discover knowledge for phylogenetic, or to mine for mediator schemas). However, it has been shown that regular data mining techniques such as algorithms extracting frequent itemsets from transactional data are not appropriate for tree mining since the data being considered are much more complex. Researchers have thus developed several methods to tackle the problem. These methods are based on the well-known level-wise generate-and-prune methods: first, candidates are generated, then they are afterwards tested to discard non frequent candidates and used to build new bigger candidates. Those propositions address several problems: (i) the representation of tree data has to be as efficient as possible (time and memory), (ii) the definition of tree inclusion, (iii) the generation of candidates and (iv) the validation of frequent candidates.

All the methods proposed in the literature suffer from the large number of candidates generated whereas some of them could be discarded before being validated over the database. As the step consisting in counting in how many trees from the database a subtree is included is very time-consuming and memory-consuming, it is mandatory to decrease the number of candidates in order to remain scalable.

In previous works, we have defined an efficient representation of the tree data structure and an optimized support counting method [7,8]. In this paper, we focus on the step of candidate generation. Our approach is based on equivalence classes, which are based on a *pivot* tree which acts as a *representative* (typical of each equivalence class). Three kinds of equivalence classes (and thus three kinds of pivots) are considered: *left*, *right* and *root*. A *left pivot tree* (resp. right), for a tree T, corresponds to the tree composed by all the nodes but the most-left leaf (resp. right) of T. The equivalence relations, which these classes are based on, will be detailed later on in this paper.

A tree where the root has no more than one child is called a *root pivot*. Each set of trees having k nodes is associated with the set of pivots that are necessary to build the equivalence classes. From these classes, it is then easy to build candidates having $k+1$ nodes by combining two trees sharing the same pivot. Roughly speaking, the method is the same as Apriori-like methods [1] where the pivot of size $k-1$ is completed by the two specific items (here nodes) in order to build a $k+1$-node tree. By using this approach, we decrease the number of candidates as we guarantee that no subtree which will be non frequent will be generated. Moreover, we show that our approach is complete, as we generate **all** the frequent subtrees.

This paper is organized as follows. Section 2 recalls the basic definitions concerning trees. Section 3 presents existing work, mostly focusing on how candidates are generated. Section 4 introduces our approach pivot and demonstrates its completeness. Experiments are reported in Section 5. Finally, Section 6 concludes the paper.

2. Basic definitions

A *rooted* tree T is an undirected, connex and acyclic graph, with a special node r called the root of T, and denoted by $root(T)$. We say that T is *labelled* if each node have assigned a symbol from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this paper are rooted, labelled and ordered. Formally a tree T is defined as a five-tuple $T = (V, E, r, \preceq, \mathcal{L})$ where: i) V is a finite set of nodes, ii) $E \subset V \times V$ is a binary relation of *parentage* between the nodes from T, iii) r is the root of T, iv) $\preceq \subseteq V \times V$ is a sibling relation of the children of each internal node $n \in V$, and v) $\mathcal{L}: V \rightarrow \Sigma$ is the labelling function for the nodes in V. The *degree* of a node $u \in V$ is defined by the size of its neighborhood, such as $deg(u) = |E(u)|$. The size of T is $|V|$.

A *path* from node u_1 to node u_k is a sequence of nodes and edges $(u_1, e_1, u_2, e_2, \ldots, e_{k-1}, u_k)$ in T such as $e_i = (u_i, u_{i+1})$ for all $i < k$ and $u_1 \neq u_k$. As an edge can only be defined by its initial and final nodes, the path from u_1 to u_k can be simply given by the succession of nodes $(u_1, u_2, \ldots, u_k)$ included in the path. The path is denoted by (u_1, u_k). The length of a path is the number of edges in the succession.

The *transitive closure* E^+ of the parentage between the nodes belonging to T are defined by:

$$E^0 = \{(u,u) | u \in V\};$$
$$E^{n+1} = \{(u,w) | \exists v \in V : (u,v) \in E \wedge (v,w) \in E^n\};$$
$$\text{for } n \geqslant 0, \text{ and}$$

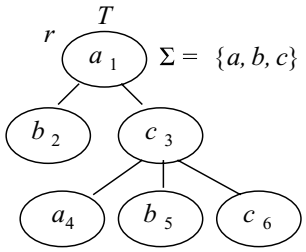


Fig. 1. A preorder numerated tree.

$$E^+ = \bigcup_{n>0} E^n.$$

Given a rooted, ordered, labelled tree (a tree for short) T the following relations can be defined depending on the unique path (r, u), from the root of T to all node $u \in V$.

The *parent* of a node $v \in V \setminus \{r\}$ is the node being the source of the edge (u, v) such as $parent(v) = \{u \in V | (u, v) \in E\}$. The root has no parent: $parent(r) = \emptyset$. For an edge $(u, v) \in E$, u is said to be the parent of v, and v the child of u. The set of *children* of a node $u \in V$ is denoted $children(u) = \{v \in V | (u, v) \in E\}$. If $children(u) = \emptyset$, the node u is a leaf.

The *ancestors* of a node $v \in V \setminus \{r\}$ are all the nodes from path (r, v) but v. The set of the ancestors of a node v is denoted by $ancestors(v) = \{u \in V | (u, v) \in E^+\}$. If u is an ancestor of v, then v is said to be a *descendant* of u. The set of all the descendants of u is denoted by: $descendants(u) = \{v \in V | (u, v) \in E^+\}$.

A node u is a *sibling* of a node $v \in V \setminus \{r\}$ if it shares the same parent: $parent(u) = parent(v)$. The set of siblings of a node is denoted $siblings(u) = \{v \in V | parent(u) = parent(v)\}$.

If the nodes $u, v, y \in V \setminus \{r\}$ and $w \in V$ are considered, together with a parentage relation $(w, u) \in E$, then it is possible to distinguish between the children of a node: given $u, v \in V$, the node u is the *left-most-child* of w, denoted by $first(w)$, if $first(w) = \{u | \nexists v, (w, v) \in E \wedge v < u\}$. The node v is the *right-most-child* of w, denoted by $last(w)$, if $last(w) = \{v | \nexists u, (w, u) \in E \wedge v < u\}$. We say that the node v is the *next-sibling* of node u, denoted by $next(u)$, if $next(u) = \{v | (w, v) \in E \wedge u < v \wedge \nexists y$ such as $(w, y) \in E \wedge u < y < v\}$. The node u is the *previous-sibling* of v, denoted by $previous(v)$, if $next(v) = \{u\}$.

The *left-most-branch* of T, denoted by $lmb(T)$, corresponds to the path $(v_1, v_2 \ldots, v_k)$, where $v_1 = r$ such that $first(v_{i-1}) = \{v_i\}, \forall i, 2 \leqslant i \leqslant k$. The final node v_k in $lmb(T)$ is called *left-most-leaf* and it is denoted by $lml(T)$.

The *right-most-branch* of T, denoted by $rmb(T)$, corresponds to the path $(v_1, v_2 \ldots, v_k)$, where $v_1 = r$ such that $last(v_{i-1}) = \{v_i\}, \forall i, 2 \leqslant i \leqslant k$. The final node v_k in $rmb(T)$ is called *right-most-leaf* and is denoted by $rml(T)$.

Example 1. *Figure 1 shows a tree T with its root r, the set of nodes $V = \{1, 2, 3, 4, 5, 6\}$, and the relations between the nodes:*

$$\begin{aligned} E &= \{(1,2), (1,3), (3,4), (3,5), (3,6)\} \\ E^+ &= \{(1,2), (1,3), (3,4), (3,5), (3,6), \\ &\quad (1,4), (1,5), (1,6)\} \end{aligned}$$

Here we have the following relationships: $parent(6) = \{3\}$, $children(3) = \{4, 5, 6\}$, $ancestors(6) = \{3, 1\}$, $descendants(1) = \{2, 3, 4, 5, 6\}$, $siblings(4) = \{5, 6\}$ $first(3) = \{4\}$, $last(3) = \{6\}$, $next(2) = \{3\}$, $previous(3) = \{2\}$, $lmb(T) = (1, 2)$, $rmb(T) = (1, 6)$, $lml(T) = \{2\}$ *and* $rml(T) = \{6\}$.

Considering a tree $T = (V, E, r, \preceq, \mathcal{L})$, a tree $S = (V', E', u, \preceq', \mathcal{L}')$ is said to be a *subtree* of T if $u \in V$ and $V' = u \bigcup descendant(u)$ and $E' = V' \times V'$ and $\preceq' \subseteq V' \times V'$ such that $(v, v') \in \preceq' \iff (v, v') \in \preceq$ and $\mathcal{L}'$ is the restriction of $\mathcal{L}$ from V to V'.

2.1. Tree mining

Given a tree database $D = \{T_1, \ldots, T_n\}$, we say that a subtree S is supported by a tree $T \in D$ if S is a subtree of T. Then the support of S with respect to a tree T is given by:

$$supp(S, T) = \begin{cases} 1, \text{ if } S \text{ is a subtree of } T \\ \\ 0, \text{ otherwise} \end{cases}$$

The support of S with respect to a database D is defined as

$$supp(S, D) = \frac{\sum_{T_i \in D} supp(S, T_i)}{|D|}$$

If $supp(S, D)$ is greater or equal to a user-defined threshold σ, then subtree S is considered *frequent*. Otherwise S is said to be an infrequent subtree. Given a tree database D and a minimum support σ ($0 \leqslant \sigma \leqslant 1$), the goal of *tree mining* is to find the collection of frequent subtrees in D.

3. Related work

In this section, we give a short survey of the existing approaches, focusing on the generation of candidates.

To efficiently extract frequent subtrees, the methods introduced in [2–5,12,16,17] use the so-called level-wise approaches. At each step of the algorithm, candidate trees are generated based on the ones that have been found frequent at the previous step. The k level thus refers to the step where trees of size k (k nodes) are being treated. These trees are then tested on the database in order to count the number of trees they are included in. This number corresponds to the $support$ value.

In the existing methods, the step of tree generation if thus critical as every frequent subtree **must** be proposed as a candidate in order to be complete, whereas it is decisive to generate as few candidates as possible in order to remain scalable.

Existing approaches consider that the generation of candidates is done by extending the right most branch [3–5,13,17]. The trees are numbered in a depth-first enumeration: for every tree T having k nodes, we have $root(T) = 1$ and $rml(T) = k$. $\mathcal{F}$ denotes the set of frequent trees and $\mathcal{F}_k$ denotes the set of frequent trees having k nodes (for every k, $\mathcal{F}_k$ is a finite set). The generation of candidates relies on equivalence classes as all the trees having k nodes are represented by the depth-first enumeration and are then grouped based on the equivalence relation *has same* $k - 1$ *prefix*. Two trees belonging to the same equivalence class only differ in the last node (position and label).

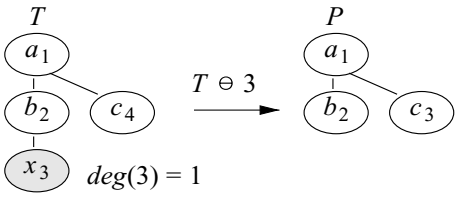


Fig. 2. Pivot P obtained by removing the left most leaf $rml(T)$ from T.

It is easy to show that this method is complete. However, this method leads to a large number of non frequent candidates. In order to limit the number of candidates, existing approaches rely on the so-called *anti-monotonicity* property [10] claiming that if a tree is non frequent, then every super-tree will be non frequent as well. In the case of trees, such a naive pruning is far too time-consuming. In order to tackle this problem, we thus propose to reduce the number of trees being generated instead of generating candidates and then pruning them.

gSpan [13] can be viewed as similar to our approach, as: 1) it handles graphs, 2) it considers the enumeration of graphs by using the right-most extension, 3) the methods are based in the Pattern-growth approach for the generation of candidates to avoid duplicates.

However there are differences that prevent from comparing these methods. First of all, the algorithm gSpan defines a labeled graph as $G = (V, E, \mathcal{L})$ whereas, PIVOT defines a Tree (graph) as $T = (V, E, r, \preceq, \mathcal{L})$, where there cannot exist any cycle. In other words, gSpan is designed to work with a more general model including: trees, lattices, sequences, whereas our approach is particularly aimed at working with rooted, labeled and ordered trees. Second, it should be noted that to validate the graph inclusion, gSpan is based on the isomorphism definition validating vertices and edge labels, whereas the method proposed here (PIVOT) is based on the inclusion test of nodes, label nodes, vertices, edges and the sibling order relation.

4. The pivot method

In this section, our pivot method is introduced. This method allows us to combine equivalence classes (left, right and root) to generate candidates.

4.1. Definitions

This section aims at introducing the $\ominus$ and $\oplus$ operators, which our approach relies on.

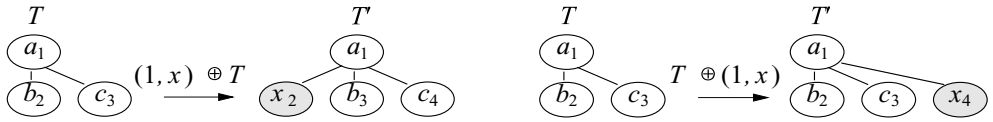
Definition 1 (**Deletion**). *Given a tree T such as $size(T) = k$ and $u \in V$ such as $deg(u) = 1$, we define the $\ominus$ operator on the trees such as $T \ominus u$ returns the subtree P of size $k - 1$ included in T from which the node u has been deleted. The subtree P is called the pivot. Figure 2 illustrates how $\ominus$ works.*

Remark: The $\ominus$ operator can not be applied on trees where the degree of the root node to be deleted equals 1 to avoid two trees as a result.

Definition 2 (**Insertion**). *Given a tree T and a pair $(p, l), 1 \leqslant p \leqslant k$, the $\oplus$ operator is such that:*

- *$T \oplus (p, l)$ inserts a new node with label l as the last child of the node being at position p in T,*
- *$(p, l) \oplus T$ inserts a new node with label l as first child to the node being at position p in T.*

Figure 3 illustrates the $\oplus$ operator.



(a) left insertion (b) right insertion

Fig. 3. Insertion of a new node in tree T.

4.2. Right equivalence class

Let $\mathcal{F}_k$ be the set of frequent trees having k nodes and two k-trees $F, F' \in \mathcal{F}_k$. Given a tree F, the function $\delta(F) = F \ominus rml(F)$ returns a pivot subtree P_δ of size $k-1$. This function defines the *right equivalence relation*, denoted by $\mathcal{P}_\delta$, as

$$F \mathcal{P}_\delta F' \iff \delta(F) = \delta(F')$$

Note that it is trivial to show that $\mathcal{P}_\delta$ is an equivalence relation as it relies on an equality. The relation $\mathcal{P}_\delta$ is thus reflexive, symmetrical and transitive. Trees F and F' belong to the same *right equivalence class* if and only if they share a common structure at position $k-1$. We denote $[\mathcal{P}_\delta(F)]$ the right equivalence class of F defined by:

$$\mathcal{P}_\delta(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\delta F'\}$$

Given a set of trees $\mathcal{F}$, we recall that the quotient set of $\mathcal{F}$ by the equivalence relation $\mathcal{P}_\delta$ corresponds to the set of the equivalence classes of $\mathcal{F}$ depending on $\mathcal{P}_\delta$ (i.e. the set $\{[\mathcal{P}_\delta(F)] | F \in \mathcal{F}\}$). Note that every tree $F \in \mathcal{F}$ belongs to one and only one equivalence class. It can thus be represented by its right pivot associated with the right most leaf (position and label) that distinguishes it from the other trees belonging to the same class.

In the following we note an equivalence class corresponding to a pivot by the notation $[P_X]$ where P stands for the pivot and X differs from the different equivalence classes. A right equivalence class $[P_\delta]$, sharing a common pivot subtree, can be represented as a pivot tree P_δ together with a list $R = \{R[1], \ldots, R[|R|]\}$ of pairs $(R[i]_1, R[i]_2)$ corresponding to every tree $F \in \mathcal{F}$ such that $F \ominus rml(F) = P_\delta$ where $R[i]_1$ ($i \in [1, |R|]$) corresponds to the position $rml(F)$ and $R[i]_2$ corresponds to the label of $rml(F)$. We thus have:

$$[P_\delta] \sim \{P_\delta, R\}$$

4.2.1. Generation of candidates regarding $\mathcal{P}_\delta$

Definition 3 (**Product**). *Given a right equivalence class $[P_\delta] \sim \{P_\delta, R\}$ and two trees F, F' such that $F \mathcal{P}_\delta F'$, and considering $(p, l), (p', l')$ two pairs in R such that $F = P_\delta \oplus (p, l)$ and $F' = P_\delta \oplus (p', l')$, then $\otimes$ operator is defined [14] to build a new $[F]$ right equivalence class. The $\otimes$ operator is applied on each pair in R such that:*

- **if** $(p == p')$ **then**
  $[F].R+ = \{(l', parent(rml(F))), (rml(F), l')\}$
- **if** $(p > p')$ **then**
  $[F].R+ = \{(parent(rml(F))), l')\}$

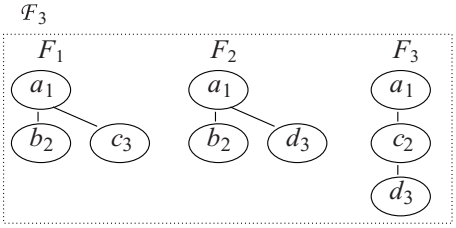


Fig. 4. Frequent trees of size 3.

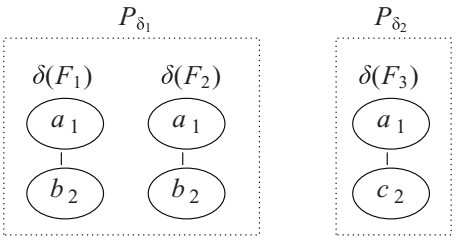


Fig. 5. Right pivots.

At each step (level) k of the algorithm, the candidates of level $k+1$ are generated by computing the product of all the pairs of elements of size $k-1$ in each equivalence class (for the pairs that share their $k-2$ first nodes). All the combinations of compatible trees (regarding the right pivot) are thus obtained.

Remark: Existing approaches, such as [3,4,17], considering an extension by the right most branch, can be represented by means of $\otimes$. However, the use of a single equivalence relation for generating the candidates leads to the generation of a large number of candidates containing at least one non frequent subtree, which should thus be discarded. We propose to combine several equivalence relations in order to decrease the number of candidates being generated.

We thus rely on the right equivalence relation defined above, but also on a left equivalence relation and a root equivalence relation, as described below.

Example 2. *Let us consider the set of frequent trees of size* 3*, denoted by* $\mathcal{F}_3$*, depicted on Fig. 4. We apply the function* δ *on the set* $\mathcal{F}_3$ *by eliminating the right most leaf from each tree such that :* $\delta(F_1) = F_1 \ominus rml(F_1)$*,* $\delta(F_2) = F_2 \ominus rml(F_2)$ *and* $\delta(F_3) = F_3 \ominus rml(F_3)$*. As* $\delta(F_1) = \delta(F_2)$*, then it is a common pivot denoted by* P_{δ_1}*. Moreover, the pivot obtained from* F_3 *is identified by* P_{δ_2}*. The pivot trees* P_{δ_1} *and* P_{δ_2} *are shown on Fig. 5. These pivot trees allow us to build the following equivalence classes:*

$$[P_{\delta_1}] \sim \{P_{\delta_1}, \{(1,c),(1,d)\}\}$$
$$[P_{\delta_2}] \sim \{P_{\delta_2}, \{(2,d)\}\}$$

Note that the first class $[P_{\delta_1}]$ *regroups the frequent trees* F_1 *and* F_2 *that can be built from the* (p,l)*-extensions* $(1,c)$ *and* $(1,d)$*, meaning that* $F_1 = P_{\delta_1} \oplus (1,c)$ *and* $F_2 = P_{\delta_1} \oplus (1,d)$*.*

The product operator $\otimes$ *applied on the pairs from the class* $[P_{\delta_1}]$ *generates the following two classes:*

$$[F_1] \sim \{F_1, \{(1,c),(3,c),(1,d),(3,d)\}\}$$
$$[F_2] \sim \{F_2, \{(1,d),(3,d),(1,c),(3,c)\}\}$$

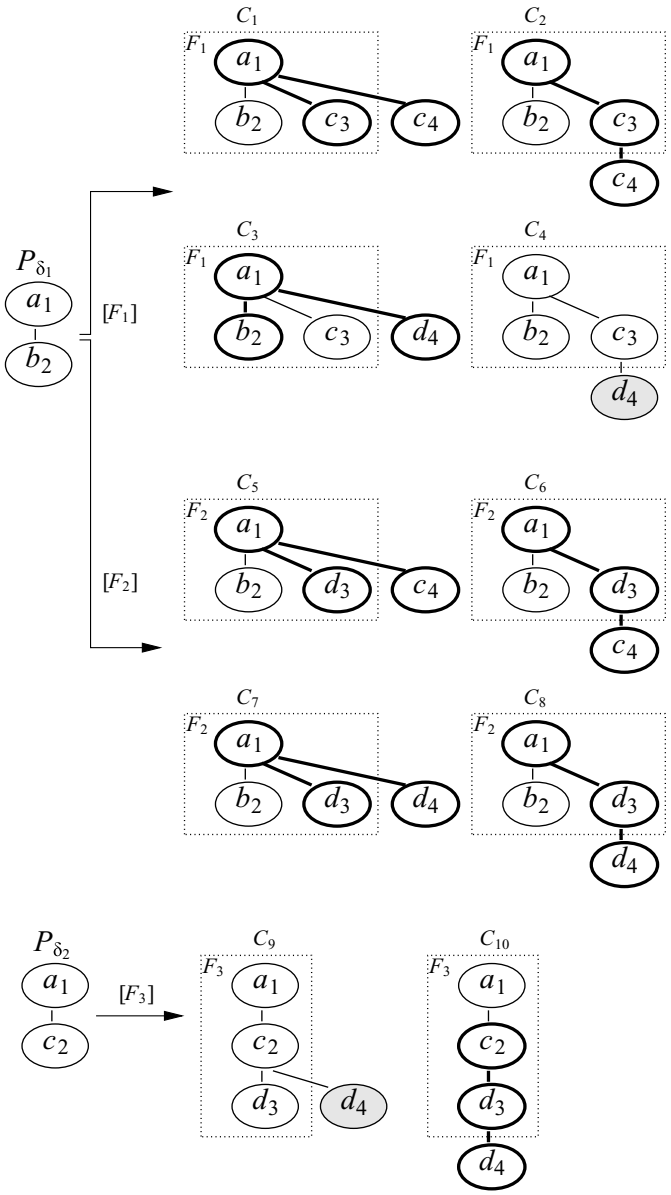


Fig. 6. Candidate trees generated by the product operator applied on the class $[P_{\delta_1}]$ and $[P_{\delta_2}]$.

The equivalence class which has F_3 as common pivot is obtained by applying the product operator on the pairs from $[P_{\delta_2}]$.

$$[F_3] \sim \{F_3, \{(2,d),(3,d)\}\}$$

The candidates $C_1, C_2, \ldots, C_{10}$ given by Fig. 6 result from the combination of the pivot on each equivalence class $[F_1]$, $[F_2]$ and $[F_3]$ with each pair from the list R in these classes. Thus, the candidate tree C_1 is generated by $C_1 = F_1 \oplus (1,c)$, and $C_2 = F_1 \oplus (3,c)$, and so on.

However, using only one relation of equivalence in the candidate generation process produces many

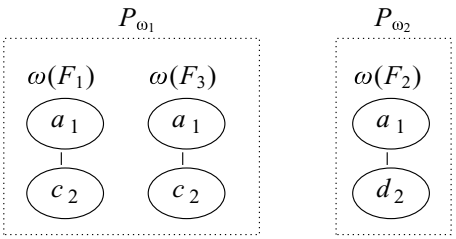


Fig. 7. Left pivot trees.

non-valid candidates (*which one knows they will not be frequent*). *We can see that only the candidates* C_4 *and* C_9 *in Fig. 6 are valid because inside the others candidates, there is a subtree of size 3* (*in bold*) *which does not belong to 3-frequent subtrees set* $\mathcal{F}_3$ (*cf. Fig. 4*). *Then, by the property of anti-monotonicity, we know a priori that these trees are not frequent. We thus propose to rely on several relations of equivalence in order to reduce the number of candidates generated while preserving a complete approach. We use at the same time the right, left and root relations of equivalence. We thereby propose a strategy of generation of candidates based on three equivalence classes allowing to handle a subset of candidates more restricted but complete.*

4.3. Left equivalence classes

Given two k-trees $F, F' \in \mathcal{F}_k$, we define a function $\omega(F) = F \ominus lml(F)$ to get the pivot subtree P_ω and we define a left equivalence relationship $\mathcal{P}_\omega$ as

$$F \mathcal{P}_\omega F' \iff \omega(F) = \omega(F')$$

Trees F and F' belong to the same left equivalence class if and only if they share the same common structure to the position $k-1$. The left equivalence class of a tree F is defined by:

$$\mathcal{P}_\omega(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\omega F'\}$$

As for the right equivalence classes, every tree $F \in \mathcal{F}$ belongs to one and only one left equivalence class and can be represented by its left pivot $P_\omega = F \ominus lml(F)$ and a pair (p, l) corresponding to the position p and the label l of its left most leaf $lml(F)$. We can thus represent a left equivalence class $[\mathcal{P}_\omega]$ by a pivot P_ω and a list $L = \{L[1], \ldots, L[|L|]$ where $L[i]_1$ ($i \in [1, |L|]$) corresponds to the position $lml(F)$ and $L[i]_2$ corresponds to the label of $lml(F)$. We thus have:

$$[P_\omega] \sim \{P_\omega, L\}$$

Example 3. *By applying the function* ω *on the trees from Fig. 4 we get the pivots* $\omega(F_1) = F_1 \ominus rml(F_1)$, $\omega(F_2) = F_2 \ominus rml(F_2)$ *and* $\omega(F_3) = F_3 \ominus rml(F_3)$ *illustrated by Fig. 7. As* $\omega(F_1) = \omega(F_3)$, *it is a common pivot identified by* P_{ω_1}. *The pivot* $\delta(F_3)$ *is denoted by* P_{ω_2}. *These pivot trees allow us to group together the trees belonging to* $\mathcal{F}_3$ *in the following equivalence classes:*

$$[P_{\omega_1}] \sim \{P_{\omega_1}, \{(1, b), (2, d)\}\}$$
$$[P_{\omega_2}] \sim \{P_{\omega_2}, \{(1, b)\}\}$$

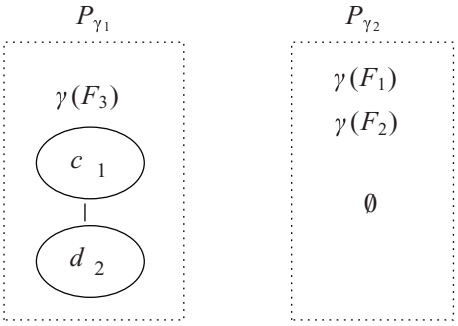


Fig. 8. Root *pivot* trees.

4.4. Root equivalence classes

Let $\gamma(F)$ be a function which removes the root from tree F of size k and returns the pivot tree P_γ of size $k-1$. $\gamma(F)$ is defined by:

$$\gamma(F) = \begin{cases} F \ominus root(F), \text{if } |children(root(F))| = 1 \\ \emptyset, \text{otherwise} \end{cases}$$

Note that if we allow $\gamma(F)$ to be applied on the trees which root has a degree greater then 1, then this function will return a forest of trees (where each tree size will be lower than $(k-1)$. In order to avoid the operations between the trees and the forests, we set that $\gamma(F) = \emptyset$ if $|children(root(F))| \neq 1$.

Given two k-root trees $F, F' \in \mathcal{F}_k$, we define the root equivalence relation $\mathcal{P}_\gamma$ as follows:

$$F \mathcal{P}_\gamma F' \iff \gamma(F) = \gamma(F')$$

The root equivalence class is defined by:

$$\mathcal{P}_\gamma(F) = \{F' \in \mathcal{F}_k | F \mathcal{P}_\gamma F'\}$$

An equivalence class which pivot tree is P_γ is denoted by $[P_\gamma]$. Each tree $F \in \mathcal{F}_k$ belongs to one and only one equivalence class $[P_\gamma]$. A root equivalence class $[P_\gamma]$ can then be represented by a pivot tree P_γ and a list of pairs $O = \{O[1], \ldots, O[|O|]\}$. Each pair $O[i]$ corresponds to the root which has been removed from a tree F. As done above, a root equivalence class $[P_\gamma]$ can be given by its root pivot P_γ together with a list of pairs (position,label). We thus have

$$[P_\gamma] \sim \{P_\gamma, O\}$$

Example 4. *Figure 8 illustrates the pivot trees $\gamma(F_1) = \gamma(F_2) = \emptyset$ and $\gamma(F_3) = F_3 \ominus root(F_3)$ obtained from $\mathcal{F}_3$ (cf. Fig. 4). As a consequence, we can build the equivalence classes:*

$$[P_{\gamma_1}] \sim \{P_{\gamma_1}, \{(-1, c)\}\}$$
$$[P_{\gamma_2}] \sim \{\emptyset, \{\emptyset\}\}$$

Note that we associate $[P_{\gamma_2}]$ to $\emptyset$ as we cannot use the trees from this class to any other one.

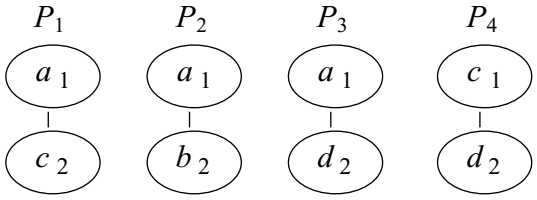


Fig. 9. *Pivot* trees.

4.5. Grouping the equivalence classes

To obtain an equivalence class with a common tree P, we regroup the equivalence classes $[P_\delta]$, $[P_\omega]$ and $[P_\gamma]$ in a new one denoted by $\mathcal{P}$, defined by:

$$\mathcal{P}(F) = \{F' \in \mathcal{F}_k | F\mathcal{P}_\delta F' \vee F\mathcal{P}_\omega F' \vee F\mathcal{P}_\gamma F'\}$$

and represented by:

$$[P] \sim \{P, R, L, O\}$$

where P is a pivot tree, R (resp. L) is a list containing the right most leaves (resp. left most leaves) removed from the frequent trees by function $\delta(F)$ (resp. $\omega(F)$) and O is a list containing all the roots removed by the function $\gamma(F)$. The list of equivalence classes $[P]$ are filled in following the criteria below:

- if $F\mathcal{P}_\delta F'$ then the pair (*position, label*) is added to list R
- if $F\mathcal{P}_\omega F'$ then the pair (*position,label*) is added to list L
- if $F\mathcal{P}_\gamma F'$ then the pair (*position, label*) is added to list O

The Algorithm 1 describes how to obtain the equivalence classes from the set of frequent subtrees of size k. In this algorithm, we use a hash table (line 1), denoted by H, in order to generate a pivot tree only once. From line 3 to line 13, the instructions are meant to deal with each frequent tree from F. Firstly, we verify whether the degree of the root is equal to 1 in order to avoid the creation of a forest. If yes, then the root of F is removed. Then, we use the resulting pivot S_1 as hash key in H and the root is added to the list O in the equivalence class. Secondly, the left most leaf is removed from F, resulting in pivot S_2, which is used as hash key while the removed leaf is added in L. Note that a pair (position, label) is added to list L. Thirdly, the pivot S_3 is obtained by removing the right most leaf, which is added to R. Finally, the *pivot* method is called to build the candidate trees.

Example 5. *Let us consider the frequent trees from Fig. 4 to build the classes having a common pivot P. In the first tree F_1, the root cannot be removed. So the leaf with number 2 is removed, leading to the pivot P_1 illustrated by Fig. 9. This tree will serve as a basis to build the class $[P_1]$ and the pair $(1, b)$ is added to its list L such that: $[P_1] \sim \{P_1, R = \emptyset, L = \{(1, b)\}, O = \emptyset\}$. Coming to F_1, by removing leaf 3, we get a new pivot identified by P_2 in Fig. 9. On the contrary to the previous case, we add the pair $(1, c)$ to the list R such that: $[P_2] \sim \{P_2, R = \{(1, c)\}, L = \emptyset, O = \emptyset\}$. The trees in F_2 and F_3 are treated in the same way to obtain the following classes:*

$$[P_1] \sim \{P_1, R = \{(2, d)\}, L = \{(1, b), (2, d)\}, O = \emptyset\}$$
$$[P_2] \sim \{P_2, R = \{(1, c), (1, d)\}, L = \emptyset, O = \emptyset\}$$

Data: $\mathcal{F}_k$, *a set of* (k) *frequent subtrees*

Result: C_{k+1} , $(k+1)$*-candidates*

1 map H;

2 *//Generating the equivalence classes*;

3 **foreach** $F \in \mathcal{F}_k$ **do**

4 *//removes the root*;

5 **if** $children(root(F)) = 1$ **then**

6 $S_1 \leftarrow (F \ominus root(F))$;

7 $H[S_1].O+ = (-1, root(F))$;

8 *//removes the left most leaf*;

9 $S_2 \leftarrow (F \ominus lml(F))$;

10 $H[S_2].L+ = (parent(lml(F)), lml(F))$;

11 *//removes the right most leaf*;

12 $S_3 \leftarrow (F \ominus rml(F))$;

13 $H[S_3].R+ = (parent(rml(F)), rml(F))$;

14 *//Generating candidates*;

15 $\mathcal{C} \leftarrow$ PIVOT(H);

16 **return** C;

Algorithm 1: GenCandPivot

$$[P_3] \sim \{P_3, R = \emptyset, L = \{(1, b)\}, O = \emptyset\}$$
$$[P_4] \sim \{P_4, R = \emptyset, L = \emptyset, O = \{(-1, a)\}\}$$

4.6. The pivot algorithm

Candidates of size $k+1$, denoted by $\mathcal{C}_{k+1}$, are generated from the combination of elements coming from the equivalence classes built from the k frequent trees and that share a common pivot of size $k-1$ (cf. Algorithm 2).

First (lines 2–5), pairs from right and root classes are combined to generate the following set of candidates $\mathcal{C}^{\gamma}_{k+1}$:

$$\mathcal{C}^{\gamma}_{k+1} = \{O[i] \oplus P \oplus R[j] \colon i \in [1, |O|] \text{ and } j \in [1, |R|]$$

Data: $[P]_{k-1}$, $(k-1)$ *equivalence class*

Result: C_{k+1} , $(k+1)$*-candidates*

1 $C \leftarrow \emptyset$;

2 **foreach** $p \in [P]_{k-1}$ **do**

3 **for** $i \leftarrow 0$ **to** $p.O.size()$ **do**

4 **for** $j \leftarrow 0$ **to** $p.R.size()$ **do**

5 $C+ = p.O[i] \oplus P \oplus P.R[j]$;

6 **for** $i \leftarrow 0$ **to** $p.L.size()$ **do**

7 **for** $j \leftarrow 0$ **to** $p.R.size()$ **do**

8 $C+ = p.L[i] \oplus P \oplus P.R[j]$;

9 **return** C;

Algorithm 2: Pivot: Generating candidates

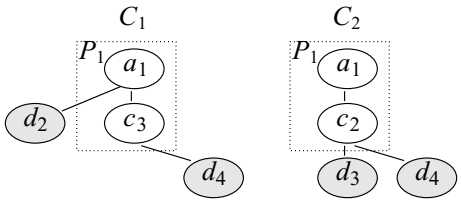


Fig. 10. *Candidates* obtained by *pivot*.

Then (lines 6–8), we consider the pivot P. All the combinations between one element from the right list and one from the left list are combined to generate the next set of candidates $\mathcal{C}^{\omega}_{k+1}$

$$\mathcal{C}^{\omega}_{k+1} = \{L[i] \oplus P \oplus R[j] : i \in [1, |L|] \text{ and } j \in [1, |R|]$$

Finally, the set of all candidates is generated from the union of two subsets of candidates generated from breadth (left + right) and depth (root + right):

$$\mathcal{C}_{k+1} = \mathcal{C}^{\omega}_{k+1} \bigcup \mathcal{C}^{\gamma}_{k+1}$$

Example 6. *Figure 10 illustrates the candidates obtained by combining the elements from the classes* $[P_1], \ldots, [P_4]$*. With class* $[P_1]$*, the candidate* C_1 *is generated with the first element from the left list* L *combined with the single element from* R*:* $C_1 = (1, b) \oplus P_1 \oplus (2, d)$*. The second candidate is obtained by combining the second element from* L *and the element from* R*:* $C_1 = (2, d) \oplus P_1 \oplus (2, d)$*. As the*

root list O is empty, it is not possible to build candidates from R combined with O. For $[P_1], [P_2]$ $[P_3]$, no candidate is generated (*either R is empty or L and O are empty*).

Note that using the Zaki methods, 10 candidates are generated with a single equivalence class whereas Pivot generated only 2 *of them* (*cf. Fig. 6*).

We demonstrate here that all the frequent subtrees are discovered and generated as candidates.

Theorem 1. *The PIVOT method allows us to extract all frequent subtrees.*

Proof 1. *Let us consider that the counting algorithm is correct. We have to prove that each frequent subtree has been generated as a candidate: $\forall f_k \in \mathcal{F}, f_k \in \mathcal{C}_k$.*

This property is demonstrated recursively by considering the size k of the frequent subtree.

k $= 1$ *The frequent subtrees of size 1* (*nodes*) *are discovered by a scan of the database, as performed by any other method. It is thus complete.*

k $= 2$ *The frequent subtrees of size 2 are discovered by combining in all possible manners the frequent nodes, which is thus complete.*

k > 2. *Let us assume that all the frequent trees of size $t \leqslant k$ have been extracted. Let us demonstrate that every tree of size $k+1$ is generated as a candidate.*

Let us consider $f_{k+1} \in \mathcal{F}_{k+1}$, every subtree of f_{k+1} is thus frequent.

Moreover, it is easy to show that for every tree t having at least 3 nodes, first deleting the right most leaf and then the left most leaf or, on the contrary, first deleting the left most leaf and then the right most leaf is the same.

Similarly, if the tree is a root tree, first deleting the root and then the right most leaf or first deleting the right most leaf and then the root is the same:

$$(t \ominus lml(t)) \ominus (rml(t \ominus lml(t)))$$
$$= (t \ominus rml(t)) \ominus (lml(t \ominus rml(t)))$$

$$(t \ominus lml(t)) \ominus (root(t \ominus lml(t)))$$
$$= (t \ominus root(t)) \ominus (rml(t \ominus root(t)))$$

Thus, the right pivot of the left pivot of a tree t is the same as the left pivot of the right pivot.

As a consequence, these trees are combined by our approach as the trees are obtained by computing the cartesian product of the trees from the equivalence classes Γ and Δ from the one hand and Ω and Δ from the other hand.

In this framework, two cases can occur: a breadth-expansion or a depth-expansion:

If *there exists $f \in f_{k+1}$ of size k such that $depth(f) = depth(f_{k+1})$. Then we denote by $f_k^1 = f_{k+1} - rml(f_{k+1})$ and $f_k^2 = f_{k+1} - lml(f_{k+1})$ the two frequent trees found by deleting the right most leaf and left most leaf. As mentioned previously, the pivot of the left equivalence class of f_k^1 ($f_k^1 \ominus lml(f_k^1)$) is equal to the pivot of the right equivalence class of f_k^2 ($f_k^2 \ominus rml(f_k^2)$).*
Thus f_k^1 et f_k^2 are combined by our method, leading to f_{k+1} by a breadth-expansion.

Otherwise *there must exist a tree f of depth $depth(f_{k+1}) - 1$ such that f is frequent as every subtree is frequent. We consider $f_k^3 = f_{k+1} - root(f_{k+1})$ and $f_k^4 = f_{k+1} - rml(f_{k+1})$ with $depth(f_k^3) = depth(f_{k+1}) - 1$.*
The right pivot droit of f_k^3 is equal to the root pivot of f_k^4. There thus exists a right equivalence class at step $k-1$ with the same pivot as on root equivalence class, leading to the generation of f_{k+1} par depth-expansion when the cartesian product will be computed.

□

Table 1
Parameters to build the synthetic dataset

Parameters	Values
Maximal depth of a tree	5
Maximal number of branches for a node	5
Maximal number of labels	50
Number of frequent trees seeded	10
Probability for a node to be a parent	0.4

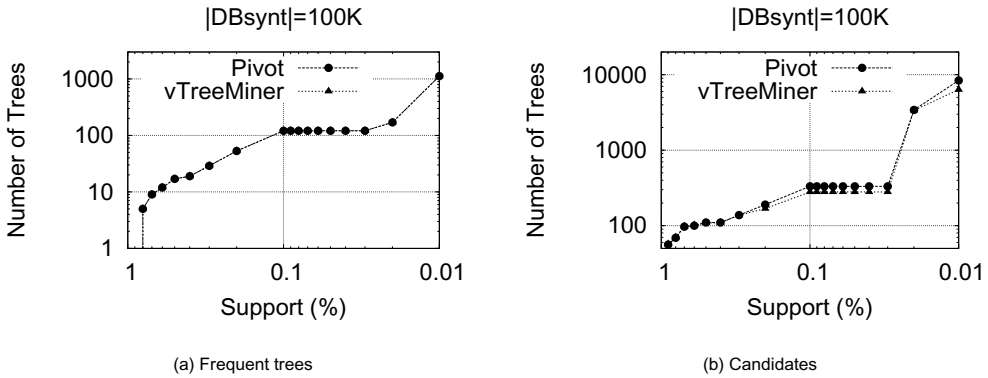


Fig. 11. Number of trees on synthetic data (embedded inclusion).

5. Experiments

The experiments were performed on a Pentium 1400 Mhz, 256 Mb RAM and 1024 Kb cache memory, running Linux 2.6 with C++ and STL data structures compiled with gcc 4.0.

5.1. Synthetic data

vTreeMiner has been chosen as a reference as it is based on *Apriori*, using the source code delivered by Zaki [15]. As our source code and this one have been developed under the same programming language, we were able to use the same function *gettimeofday*() to measure runtimes and the function *mallinfo*() to measure the memory consumption.

We generated a database using method proposed in [12]. The parameters used to generate these datasets are given in Table 1.

When considering *embedded* inclusion, meaning that *ancestor-descendants* are taken in account, we use the *vTreeMiner* algorithm. The support has been defined from 0.9 to 0.01.

Figure 11(a) reports the number of frequent trees found using *Pivot* and *vTreeMiner* algorithms. The generation of candidates are very close between *vTreeMiner* and *Pivot*, (cf. Fig. 11(b)). The scopes used by *vTreeMiner* help it to get better runtimes (cf. Fig. 12(a)). However, it is more memory-consuming (cf. Fig. 12(b)).

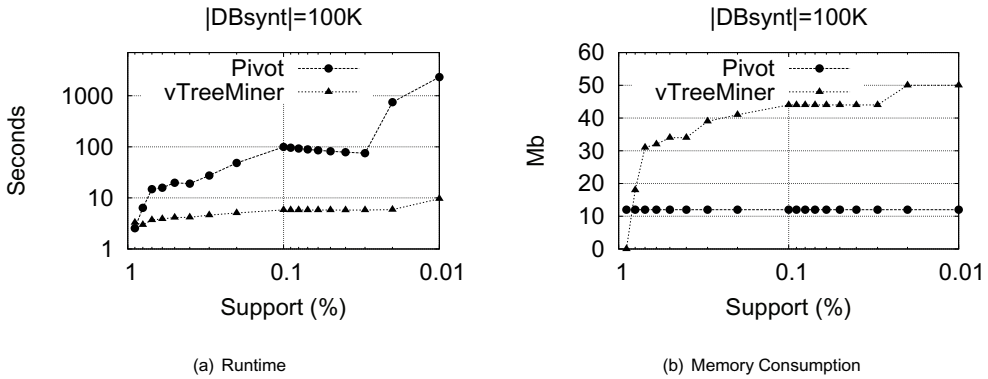


Fig. 12. Runtime and memory.

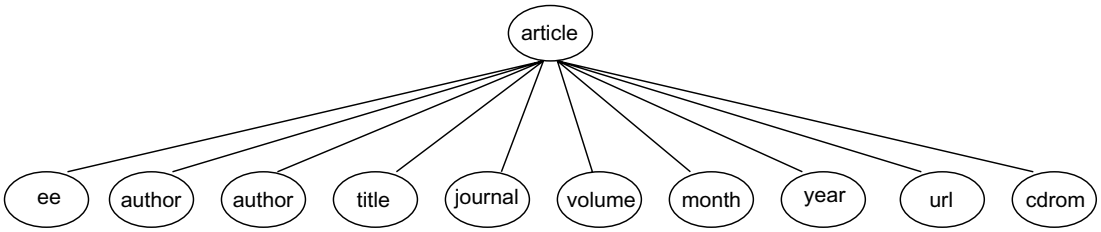


Fig. 13. Form of the DBLP trees.

5.2. Real data

We report here the results obtained from the DBLP[1] database available from [9]. This database offers bibliographic information on major computer science journals and proceedings, containing 328,458 references. Each entry of the database is a tree describing a paper published in a DBLP linked conference or journal. Figure 13 shows the typical form of such trees.

The trees are obtained from the references. For instance, the reference below is transformed into the tree shown by Fig. 14:

```
<article key="journals/siamdm/CanteautCD00">
<author>Anne Canteaut</author>
<author>Pascale Charpin</author>
<author>Hans Dobbertin</author>
<title>Weight Divisibility of Cyclic Codes, Highly Nonlinear
       Functions on F<sub>2<sup><i>m</i></sup></sub>,
       and Crosscorrelation of Maximum-Length Sequences.</title>
<pages>105-138</pages>
<year>2000</year>
<volume>13</volume>
<journal>SIAM Journal on Discrete Mathematics</journal>
<number>1</number>
```

[1]Digital Bibliography & Library Project.

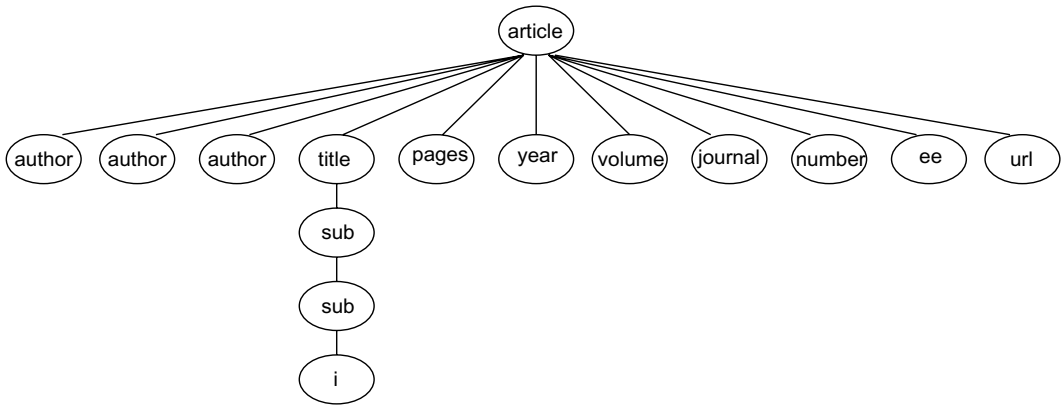


Fig. 14. Sample DBLP tree.

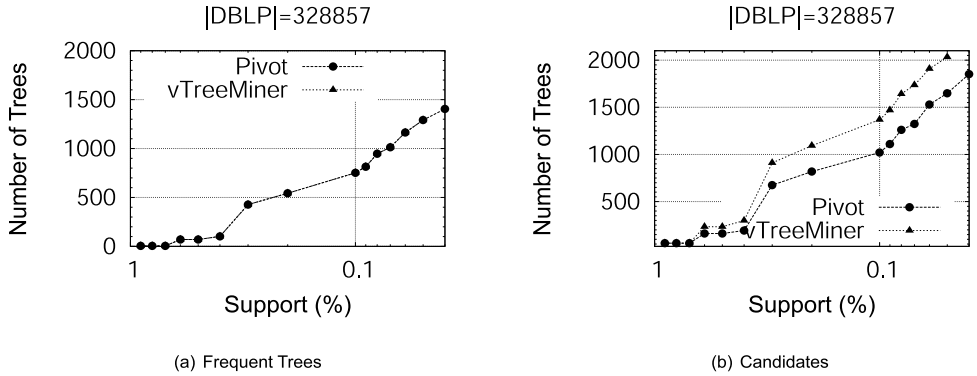


Fig. 15. Number of Trees in the DBLP database, with embedded inclusion.

```
<ee>http://epubs.siam.org/sam-bin/dbq/article/35005</ee>
<url>db/journals/siamdm/siamdm13.html#CanteautCD00</url>
</article>
```

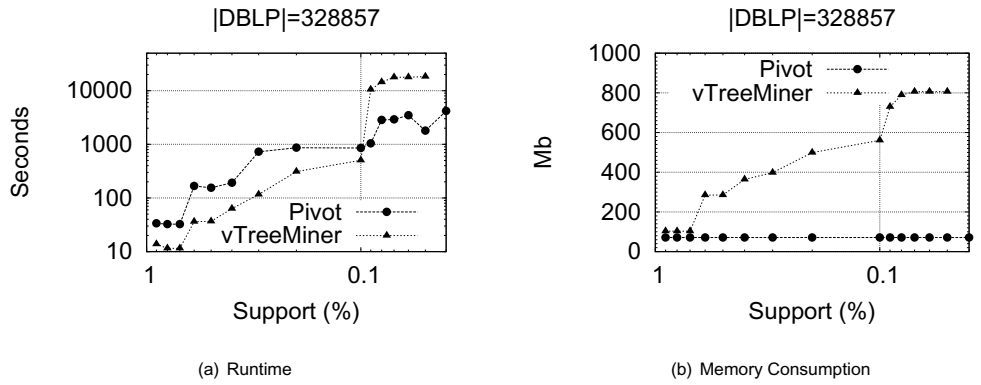
We compare in this section the results obtained by using *Pivot* and *vTreeMiner* algorithms.

Figure 15(a) shows the validity of our algorithms as they all extract the same frequent trees. Figure 15(b) shows the evolution of the number of candidates depending on the support (ranging from 0.9 to 0.01). As shown here, *Pivot* generates less candidates than *vTreeMiner*.

Figure 16(a) reports experiments conducted in order to evaluate runtime executions. As we can notice, Pivot clearly outperforms vTreeMiner. Furthermore, we can observe that for support values lower than 0.09, the execution time of vTreeMiner increases very quickly. The Fig. 16(b) shows the memory consumption of both algorithms. As expected, according to the data representation, Pivot is less memory-consuming.

As can be seen from the experiments reported above, Pivot is very efficient regarding the memory consumpation and can even outperform existing methos regarding runtime. More specifically, when our approach is applied on wide trees, then it outperforms existing methods, while it is not adapted to deep trees.

Regarding real data (DBLP) that are based on quite wide trees, vTreeMiner runs faster than Pivot for minimal supports greater than 0.1. However, it should be noted that this is inverted when supports



(a) Runtime (b) Memory Consumption

Fig. 16. Experiments on the DBLP database.

become lower than 0.1, and that vTreeMiner cannot compute the frequent subtrees when the support is lower than 0.03 as the memory is fully occupied and fails.

6. Conclusion

Tree mining is an active but complex data mining topic having numerous applications (e.g. phylogenic, extraction of mediator schema). Many methods have been proposed in the literature, usually based on level wise methods. In this framework, the generation of candidates is one of the key points to design scalable algorithms. However, most of the existing tree mining approaches consider simple candidate generations, by building k-node candidates from $(k-1)$-node frequent subtrees which they extend by adding a single node to the right most branch.

In this paper, we thus introduce an original and efficient method for generating candidates in the context of tree mining. We propose to merge two $(k-1)$-node frequent subtrees that share a *pivot* (i.e. a $(k-2)$ node tree) for building a k-node candidate, in the same way as itemsets are merged in the APriori framework.

By this way, a large number of candidates can be immediately discarded, which saves memory. We propose a new method Pivot combining several equivalence relations in order to decrease the number of candidates being generated. We rely on the right equivalence relation, but also on a left equivalence relation and a root equivalence relation. This method is proven to be complete and efficient (regarding experiments reported here).

Acknowledgments

The authors would like to thank M. J. Zaki (Department of Computer Science, Rensselaer Polytechnic Institute Troy, New York), for the source codes of vTreeMiner and hTreeMiner he provided to us.

References

[1] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in: *Proc 20th Int Conf Very Large Data Bases, VLDB*, 1994, pp. 487–499.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto and S. Arikawa, Efficient substructure discovery from large semistructured data, in: *SIAM Int Conf on Data Mining*, 2002.
[3] T. Asai, H. Arimura, T. Uno and S. Nakano, Discovering frequent substructures in large unordered trees, in: *DOI Technical Report DOI-TR 216, Department of Informatics, Kyushu University*, 2003.
[4] Y. Chi, S. Nijssen, R. Muntz and J. Kok. Frequent subtree mining – an overview, *Fundamenta Informaticae* **66**(1–2) (2005), 161–198.
[5] Y. Chi, Y. Xia, Y. Yang and R. Muntz, Mining closed and maximal frequent subtrees from databases of labeled rooted trees, *IEEE Transactions on Knowledge and Data Engineering* **17**(2) (2005), 190–202.
[6] J.D. Knijf, Frequent tree mining with selection constraints, in: *Proc of the 3rd Int Workshop on Mining Graphs, Trees and Sequences* (*MGTS*), 2005.
[7] A. Laurent, P. Poncelet and M. Teisseire, Fuzzy data mining for the semantic web: Building xml mediator schemas, in: *Fuzzy Logic and the Semantic Web*, E. Sanchez, ed., Capturing Intelligence, Elsevier, 2006.
[8] F.D.R. Lopez, A. Laurent, P. Poncelet and M. Teisseire, Rsf – a new tree mining approach with an efficient data structure, in: *Proceedings of the joint Conference: 4th Conference of the European Society for Fuzzy Logic and Technology* (*EUSFLAT 2005*), 2005.
[9] D. of Computer Science & Engineering University of Washington. Xmldata repository, 2002. http://www.cs.washington.edu/research/xmldatasets/.
[10] J. Pei and J. Han, Constrained frequent pattern mining: A pattern-growth view, *ACM SIGKDD* **2**(2), 2002.
[11] S. Tatikonda, S. Parthasarathy and T. Kurc, Trips and tides: new algorithms for tree mining, in: *Proc 15th ACM Int Conf on Information and Knowledge Management* (*CIKM*), 2006.
[12] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder, a first step towards XML data mining, in: *IEEE Conference on Data Mining* (*ICDM*), 2002, pp. 450–457.
[13] X. Yan and J. Han, gspan: Graph-based substructure pattern mining, in: *Proc of the IEEE International Conference on Data Mining* (*ICDM*), 2002, pp. 721–724.
[14] M. Zaki, Efficiently mining frequent trees in a forest, in: *ACM-SIGKDD'02*, 2002.
[15] M.J. Zaki, Treeminer code, http://www.cs.rpi.edu/~zaki/software/.
[16] M.J. Zaki, Efficiently mining frequent embedded unordered trees, *Fundamenta Informaticae* **65**(1–2) (2005), 33–52.
[17] M.J. Zaki, Efficiently mining frequent trees in a forest: Algorithms and applications, *IEEE Transaction on Knowledge and Data Engineering* **17**(8) (2005), 1021–1035.