Frequent Pattern Mining in Attributed Trees

Claude Pasquier[1,2], Jérémy Sanhes[1], Frédéric Flouvat[1], and Nazha Selmaoui-Folcher[1]

[1] University of New Caledonia, PPME, BP R4, F-98851 Nouméa, New Caledonia
{jeremy.sanhes,frederic.flouvat,nazha.selmaoui}@univ-nc.nc
[2] Institute of Biology Valrose (IBV)
UNS - CNRS UMR7277 - INSERM U1091, F-06108 Nice cedex 2
claude.pasquier@unice.fr

Abstract. Frequent pattern mining is an important data mining task with a broad range of applications. Initially focused on the discovery of frequent itemsets, studies were extended to mine structural forms like sequences, trees or graphs. In this paper, we introduce a new data mining method that consists in mining new kind of patterns in a collection of attributed trees (atrees). Attributed trees are trees in which vertices are associated with itemsets. Mining this type of patterns (called asubtrees), which combines tree mining and itemset mining, requires the exploration of a huge search space. We present several new algorithms for attributed trees mining and show that their implementations can efficiently list frequent patterns in a database of several thousand of attributed trees.

Keywords: tree mining, frequent pattern mining, attributed tree.

1 Introduction

Frequent pattern mining is an important problem in data mining research. Initially focused on the discovery of frequent itemsets [1], studies were extended to mine structural forms like sequences [2], trees [7] or graphs [22]. While itemset mining seeks frequent combinations of items in a set of transactions, structural mining seeks frequent substructures. Most existing studies focus only on one kind of problem (itemset mining or structural mining). However, in order to represent richer information, it seems natural to consider itemsets that are organized in complex structures. In this paper, we introduce the problem of mining attributed trees that are tree structures in which each vertex is associated with an itemset.

In web log analysis, for example, it is common to represent user browsing in tree-like data where each page is identified with an unique id. However, one can more pertinently characterize browsed pages with lists of keywords associated with their content. This approach allows to capture the browsing habits of users even when the web site is reshuffled. Other applications can be imagined in various area such as retweet trees mining, spatio-temporal data mining, phylogenetic tree mining and XML document mining.

The key contributions of our work are the following: 1) We present the problem of mining ordered and unordered substructures in a collection of attributed trees.

J. Pei et al. (Eds.): PAKDD 2013, Part I, LNAI 7818, pp. 26–37, 2013.

2) We define canonical forms for attributed trees. 3) We propose a method for attributed trees enumeration that is based on two operations: itemset extension and tree extension. 4) We present an efficient algorithm IMIT for extracting frequent substructures in a set of attributed trees. 5) We perform extensive experiments on several synthetic datasets and a real weblogs dataset.

The rest of this paper is organised as follows. Section 2 presents basic concepts and defines the problem. Section 3 proposes a brief overview of related works, particularly few studies that mix itemset mining and structure mining. Section 4 describes the method including the search space exploration, the frequency computation and the candidates pruning method. Section 5 reports several applications of the algorithms to mine both synthetic and real datasets. Finally, section 6 concludes the paper and presents possible extensions of the current work.

2 Basic Concepts and Problem Statement

In this section, we give basic definitions and concepts and then introduce the problem of attributed tree mining.

2.1 Preliminaries

Let $\mathcal{I} = \{i_1, i_2, .., i_n\}$ be a set of items. An **itemset** is a set $\mathcal{P} \subseteq \mathcal{I}$. The size of an itemset is the number of items. The set $\mathcal{D}$ of itemsets presents in a database is denoted by $\{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$ where $\forall \mathcal{P} \in \mathcal{D}, \mathcal{P} \subseteq \mathcal{I}$. $\mathcal{D}$ is a transaction database.

A **tree** $S = (V, E)$ is a directed, acyclic and connected graph where V is a set of vertices (nodes) and $E = \{(u, v) | u, v \in V\}$ is a set of edges. A distinguished node $r \in V$ is considered as the root, and for any other node $x \in V$, there is a unique path from r to x. If there is a path from a vertex u to v in $S = (V, E)$, then u is an *ancestor* of v (v is a *descendant* of u). If $(u, v) \in E$ (i.e. u is an immediate ancestor of v), then u is the *parent* of v (v is a *child* of u). An ordered tree has a left-to-right ordering among the siblings. In this paper, unless otherwise specified, all trees we consider are unordered.

An **attributed tree**, or (**atree**) is a triple $T = (V, E, \lambda)$ where (V, E) is the underlying tree and $\lambda : V \rightarrow \mathcal{D}$ is a function which associates an itemset $\lambda(u) \in \mathcal{I}$ to each vertex $u \in V$. The size of an attributed tree is the number of items associated with its vertices.

In this paper, we use a string representation for an atree based on that defined for labeled trees by Zaki [24]. This representation is only intended to provide a readable form for atrees. The string representation for an atree T is generated by adding a representation of the nodes found in T in a depth-first preorder traversal of T and adding a special symbol \$ when a backtracking from a child to its direct parent occurs. In the paper, for simplicity, we omit the trailing \$s. A string representation of a node is generated by listing all the items present in the associated itemset in a lexicographical order. For example, the string representation of atree $T2$ from Fig. 1 is "*a c* \$ *cde ab* \$ *a*".

Attributed trees can be understood as itemsets organized in a tree structure. As such, **attributed tree inclusion** can be defined with respect to itemsets

inclusion or structural inclusion. For itemset inclusion, we say that atree T_1 is contained in another atree T_2 if both atrees have the same structure and for each vertex of T_1, the associated itemset is contained in the itemset of the coresponding vertex in T_2. More formally, $T_1 = (V_1, E_1, \lambda_1)$ is **contained in** $T_2 = (V_2, E_2, \lambda_2)$, and is denoted by $T_1 \sqsubset_I T_2$, *if* $V_1 = V_2$ and $E_1 = E_2$ and $\forall x \in V_1, \lambda_1(x) \subseteq \lambda_2(x)$. Structural inclusion is represented by the classical concept of subtree [5,7,12,17,19,23,24].

From the previous definition, we generalize the notion of asubtree in the following way. $T_1 = (V_1, E_1, \lambda_1)$ is a **asubtree** of a atree $T_2 = (V_2, E_2, \lambda_2)$ and is denoted $T_1 \sqsubset T_2$ *if* T_1 is an isomorphic asubtree of T_2, i.e. there exists a mapping $\varphi : V_1 \to V_2$ such that $T_1 \neq T_2$ and $(u, v) \in E_1$ if $(\varphi(u), \varphi(v)) \in E_2$ and $\forall x \in V_1, \lambda_1(x) \subseteq \lambda_2(\varphi(x))$. If T_1 is an asubtree of T_2, we say that T_2 is an asupertree of T_1. T_1 is called an **induced asubtree** of T_2 *iff* T_1 is an isomorphic asubtree of T_2 and φ preserves the parent-child relationships. T_1 is called an **embedded asubtree** of T_2 *iff* T_1 is an isomorphic asubtree of T_2 and φ preserves the ancestor-descendant relationships. $T_1 = (V_1, E_1, \lambda_1)$ is called a **gap-i asubtree** of $T_2 = (V_2, E_2, \lambda_2)$ *iff* T_1 is an isomorphic asubtree of T_2 and φ preserves the ancestor-descendant relationships with the following constraint: $\forall u \forall v \in E_1$ such that u is an ancestor of v and $d(\varphi(u), \varphi(v)) = 1$, $d(u, v) \leq i$ where $d(x, y)$ represents the number of edges between x and y in the atree.

Fig. 1 shows an example of an atree database composed of three different atrees with two (incomplete) sets of common asubtrees using a maximum gap of 0 and 1.

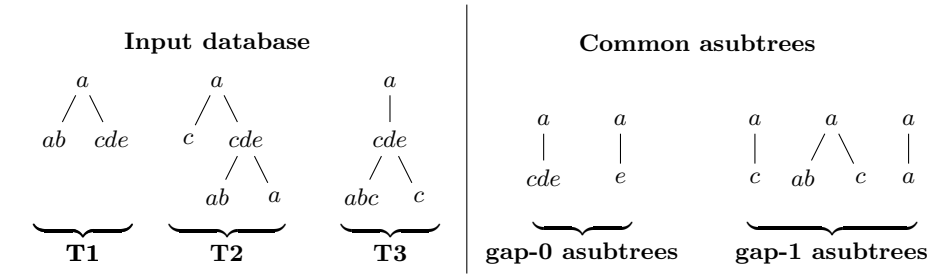

Fig. 1. Example of an atrees database with some common asubtrees

All tree mining algorithms dealing with unordered trees have to face the isomorphism problem. To avoid the redundant generation of equivalent solutions, one tree is chosen as the canonical form and other alternative forms are discarded [3,8,17,23,25]. In previous works, canonical forms are based on a lexicographical ordering on node's labels. In our work, we define an ordering based on node's associated itemsets. Given two itemsets $\mathcal{P}$ and $\mathcal{Q}$ ($\mathcal{P} \neq \mathcal{Q}$), we say that $\mathcal{P} < \mathcal{Q}$ iff 1) $\forall i \in [1, \min(|\mathcal{P}|, |\mathcal{Q}|)] : \mathcal{P}_i \leq \mathcal{Q}_i$ and 2) if $\forall i \in [1, \min(|\mathcal{P}|, |\mathcal{Q}|)] : \mathcal{P}_i = \mathcal{Q}_i$, then $|\mathcal{P}| > |\mathcal{Q}|$. From the definition above, an ordering, $\prec$, among atrees can be defined. From this, a **canonical form of isomorphic atrees** is easily determined using the method presented by Chi et al. [7].

The problem with frequent atrees mining is that the number of frequent patterns is often large. In real applications, generating all solutions can be very expensive or even impossible. Moreover, lots of these frequent atrees contain redundant information. In Fig. 1, for example, atree "$a\ e$" is present in all transactions but the pattern is already encoded in atree "$a\ cde$" because "$a\ e$" is contained in atree "$a\ cde$". This is the same for atree "$a\ a$" which is an asubtree of "$a\ ab\ \$\ c$".

Since the proposal of Manilla et al. [13] huge efforts have been made to design condensed representations that are able to summarize solutions in smaller sets. Set of closed patterns is an example of such a condensed representation [18]. We say that an atree T is a **closed atree** if none of its proper asupertrees has the same support as T. In this paper, we introduce another condensed representation which is defined with respect to the **contained in** relationship only. We say that an atree T is a **c-closed atree** (content closed) if it is not contained (as defined above) in another atree with the same support as T.

2.2 Problem Statement

Given a database $\mathcal{B}$ of atrees and an atree T, the **per-tree frequency** of T is defined as the number of atrees in $\mathcal{B}$ for which T is an asubtree. An atree is frequent if its per-tree support is greater than or equal to a minimum threshold value. The problem consists in enumerating all frequent patterns in a given forest of atrees.

3 Related Works

Most of the earlier frequent tree mining algorithms are derived from the well-known Apriori strategy [1]: a succession of candidates generation phase followed by a support counting phase in which infrequent candidates are filtered out. Two strategies are possible for candidate generation: extension and join. With extension, a new candidate tree is generated by adding a node to a frequent tree [3,17]. With join, a new candidate is created by combining two frequent trees [12,25]. Combination of the two principles has also been studied [8].

Extension principle is a simple method suitable to mine implied trees because the number of nodes that can be used to extend a given subtree is often lower than the number of frequent subtrees.

Other tree mining algorithms are derived from FP-growth approach [11]. These algorithms, which adopt the divide-and-conquer pattern-growth principle avoid the costly process of candidate generation. However, pattern-growth approach cannot be extended simply to tackle the frequent tree pattern mining problem. Existing implementations are limited in the type of trees they can handle: induced unordered trees with no duplicate labels in each node's childs [23], ordered trees [21] or embedded ordered trees [26] are some kind of trees that were successfully mined with pattern-growth approach.

Finding condensed representations of frequent patterns is a natural extension of pattern mining. For itemset mining, the notion of closure is formally defined

[18]. Several works explored this topic in the context of tree mining and proposed mining methods as well as various implementations [9,19,20]. To the best of our knowledge, no method has been proposed for the general case of attributed trees.

Recently we saw growing interest in mining itemsets organized in structures. Miyoshi et al. [14] consider labeled graphs with quantitative attributes associated with vertices. This kind of structure allows to solve the problem by combining a "classical" subgraph mining algorithm for the labeled graph, and an existing itemset mining algorithm for quantitative itemsets in each vertex. Mining attributed subgraphs independently of labels of vertices is impossible with this approach. Several studies [10,15,16] deal with attributed graphs but are looking for frequent subgraphs sharing common sets of attributes. Our work differs from these studies in the sense that itemsets associated with the vertices of a given frequent substructures are not necessarily identical.

4 Mining Frequent Atrees

We are mainly interested in identifying induced ordered and unordered asubtrees. Depending on applications, some patterns including gaps in the ancestor-descendant relationship can also be considered. However, in order to collect only interesting patterns, the gap used should remain small. Otherwise, the relationship between a node and its descendants is not really tangible. Although we focus on induced asubtrees, we designed a general method that is able to mine asubtrees with any gap value, including embedded asubtrees. However, because of the primary objective, our method works better for induced asubtree mining and performances decrease as gap parameter increases.

4.1 Atrees Enumeration

Using the operator $\prec$, it is possible to construct a candidate tree Q representing the complete search space [4] in the following way. The root node of the tree is at the top level and labeled with $\emptyset$. Recursively, for each leaf node $n \in Q$, children n' are added such that $n \prec n'$. Children of a node $n \in Q$, are generated either by tree extension or by itemset extension.

Tree Extension. For tree extension, we use a variation of the well-known rightmost path extension method [3,17]. Let T be an atree of size k. T can be extended to generate new atrees in two different ways. In the first way, a new child N is added to the rightmost node of T (right node extension). In the second way, a new sibling N is added to a node in the rightmost path of T (right path extension) [6].

In the classical approach, N represents every valid node from the input database. In our approach, new nodes N are created from every valid node Q from the input database. In fact, each node Q, associated with an itemset of size k, generates a set of k nodes $\mathcal{N} = \{N_1, .., N_k\}$ used for tree extension. Each N_i is associated with an itemset of size 1; the only item being the ith item of $\lambda(Q)$.

For example, in Fig. 1, the nodes that can be used for right node extension of pattern "$a\ cde$" are "ab", "a" (from atree $T2$), "abc" and "c" (from atree $T3$). From node "abc", three extensions are generated ("a", "b" and "c") while node "ab" generates "a" and "b". Nodes "a" and "c" generate extensions "a" and "c" respectively. Three different candidates are then obtained by adding each of these extension to the candidate pattern: "$a\ cde\ a$", "$a\ cde\ b$" and "$a\ cde\ c$"

For ordered trees, this method of candidate generation has been shown to be complete as well as non-redundant [3]. However, for unordered trees, it might generate redundant patterns in the form of isomorphic trees. Duplicate candidates are detected and discarded before the candidate extension process by performing a canonical check.

Itemset Extension. For itemset extension, we use a variation of the method presented by Ayres et al. [4]. With this variation, a new item I is added to the itemset associated with the rightmost node of the candidate atree T. Items used for itemset extension are derived from the itemset associated with this node in the input database. The constraint is that the new item must be greater than any item associated with the rightmost node of T.

4.2 Frequency Computation

We organize our data in a structure storing all information needed for the mining process. Our structure is an extension of the vertical representation of trees introduced by Zaki [24,25]. Briefly, each candidate asubtree is associated with its pattern and several data allowing to pinpoint all its occurrences in the database. The first candidates, composed of a unique node associated with one item, are generated by scanning the input database. Using only this unique structure, it is easy to compute the number of occurrences of each pattern. In addition, this same structure is sufficient to generate all possible extensions of a given pattern. When a pattern of size k is processed, all occurrences are extended with tree extension and itemset extension methods described before to generate new $(k+1)$-candidates that are themselves stored in the structure.

4.3 Search Space Exploration

Several techniques can be used to prune the search tree.

Candidate Pruning. The same rules specified by Agrawal and Srikant twenty years ago [1], can be applied to the case of atrees: i) any sub-pattern of a frequent pattern is frequent, and ii) any super-pattern of a non frequent pattern is non frequent. As the frequency count is an anti-monotonic function (extending a pattern cannot lead to a new pattern with a greater frequency), is it possible to stop the exploration of a branch when the frequency of a candidate is less than the minimum support. For example, in Fig. 1, during the mining of atrees, when we examine pattern "$a\ c\ a$" and found that its frequency is lower than the minimum support, we do not generate candidates obtained by extending "$a\ c\ a$" (e.g. "$a\ c\ ab$", "$a\ c\ a\ \$\ b$", "$a\ c\ a\ \$\ \$\ c$").

In addition, in the case of unordered tree mining, extension of a candidate is stopped if it is not in canonical form.

C-closed Atrees Enumeration. By enumerating only atrees that are not **contained** in another atree with the same support, the search space can be considerably reduced. Enumerating c-closed atrees involves the storage of every frequent pattern found with their associated per-tree frequency and their total number of occurrences in the database (the **occurrence-match frequency**).

Let T be a candidate atree currently processed, $\mathcal{T}$ be the set of all previously identified frequent atrees and $\mathcal{X}$ be the set of candidates generated by extension of T. We distinguish two subsets of $\mathcal{X}$. $\mathcal{X}_I$ is the set of atrees generated by itemset extension of T and $\mathcal{X}_T$ is composed of tree extensions of T. We define two functions: f_t which gives the per-tree frequency of an atree and f_o which returns its occurence-match frequency.

We say that T is a c-closed atree if $\nexists T' \in \mathcal{T} U \mathcal{X}_I$ such that $T \sqsubset_I T'$ and $f_t(T') = f_t(T)$. However, finding an itemset extension of T with the same per-tree frequency as T does not allows to stop the exploration of other candidates in $\mathcal{X}$. The following additional conditions must also be satisfied: $\exists T' \in \mathcal{T} U \mathcal{X}_I$: $T \sqsubset_I T'$ and $f_o(T') = f_o(T)$.

In Fig. 1, for example, the first candidate to be examined is "a" with a per-tree frequency of 3. By itemset extension, we build $\mathcal{X}_I = \{$"ab", "ac"$\}$. Candidate "ab" has a per-tree frequency of 3, therefore, candidate "a" is not c-closed as "a" $\sqsubset_I$ "ab". However, pattern "a" appears 7 times in the database while the total occurrence of candidate "ab" is 3. The 4 times where "a" occurs in an itemset which does not contain "b" may lead to the generation of other patterns that are c-closed. This is the case in Fig. 1 where a right node extension of pattern "a" generates candidate "$a\ e$" with a per-tree frequency of 3.

Closed Atrees Enumeration. We say that T is a a closed atree if $\nexists T' \in \mathcal{T} U \mathcal{X}$ such that $T \sqsubset T'$ and $f_t(T') = f_t(T)$. The extension of T can be stopped if $\exists T' \in \mathcal{T} U \mathcal{X} : T \sqsubset T'$ and $f_o(T') = f_o(T)$. In addition, one has also to remove non closed trees from $\mathcal{T}$, i.e. all atrees that are asubtrees of T with a same per-tree frequency. The check for closure requires to perform several subtree isomorphism checks that are costly operations.

4.4 Mining Algorithms

Fig. 2 shows the high level structure of the IMIT algorithm. First, a set with all asubtree of size 1 is built by scanning the input database. Then, a loop allows to process every candidate in the set. The function *GetFirst* return the smallest candidate in the set according to the $\prec$ operator. The processing involves a canonical test and a frequency test. A frequent candidate which is in canonical form is added to the list of solutions and all of its extensions are added to the list of candidates. The processing of a candidate finishes by removing it from the candidates' list.

```
IMIT(𝒟, minSup)
 1: 𝒞 ← {all asubtrees of size 1 in 𝒟}
 2: while 𝒞 ≠ ∅ do
 3:    T ← getFirst(𝒞)
 4:    if isCanonical(T) and f_t(T) ≥ minSup then
 5:       𝒯 ← 𝒯 ∪ {T}
 6:       𝒞 ← 𝒞 ∪ 𝒳
 7:    end if
 8:    𝒞 ← 𝒞 \ {T}
 9: end while
10: printSolutions(𝒯)
```

Fig. 2. IMIT Algorithm

This algorithm is sufficient to enumerate all solutions but it has a huge search space. To limit the redundancies in the set of solutions, we developed IMIT_CLOSED, an algorithm extracting closed asubtrees (Fig. 3). As illustrated in section 5, the algorithm is costly and is not usable to mine large input databases.

We designed IMIT_CONTENT_CLOSED, a third algorithm extracting c-closed asubtrees. This new algorithm (not shown in this paper) can be easily deduced from the IMIT_CLOSED algorithm (Fig. 3) by replacing $\sqsubset$ by $\sqsubset_I$, replacing $\mathcal{X}$ by $\mathcal{X}_I$ in line 5 and removing line 11 to 13. The use of $\sqsubset_I$ instead of $\sqsubset$ allows to only perform itemsets inclusion tests that are less costly than subtree isomorphism checks. Lines 11 to 13 remove from the set of solutions those that are asubtree of the current candidate. This test is not needed for the

```
IMIT_CLOSED(𝒟, minSup)
 1: 𝒞 ← {all asubtrees of size 1 in 𝒟}
 2: while 𝒞 ≠ ∅ do
 3:    T ← getFirst(𝒞)
 4:    if isCanonical(T) and f_t(T) ≥ minSup then
 5:       if ∄T' ∈ 𝒯 ∪ 𝒳 : T ⊏ T' and f_t(T') = f_t(T) then
 6:          𝒯 ← 𝒯 ∪ {T}
 7:       end if
 8:       if ∄T' ∈ 𝒯 : T ⊏ T' and f_o(T') = f_o(T) then
 9:          𝒞 ← 𝒞 ∪ 𝒳
10:       end if
11:       for all T' ∈ 𝒯 such that T' ⊏ T and f_o(T') = f_o(T') do
12:          𝒯 ← 𝒯 \ {T'}
13:       end for
14:    end if
15:    𝒞 ← 𝒞 \ {T}
16: end while
17: printSolutions(𝒯)
```

Fig. 3. IMIT_CLOSED Algorithm

extraction of c-closed patterns. Experiments show that this third algorithm is the best compromise between non redundancy of solutions and execution time.

5 Experimental Results

All algorithms are implemented in C++ using STL. Experiments were performed on a computer running Ubuntu 12.04 LTS and based on a Intel©Core^{TM}i5-2400 @ 3.10GHz with 8 Gb main memory. All timings are based on total execution time, including all preprocessing and results output.

5.1 Synthetic Datasets

We modified the synthetic data generation program proposed by Zaki [24] in order to be able to generate atrees with different size of itemsets. We added two new parameters controlling the minimum and maximum itemset's size. This allows to generate atree with fixed itemset's size or with a size randomly chosen in a range.

We used the default parameters as in [24] except for the number of subtrees T that is set to 10,000. We build five datasets by varying the size of itemsets. In T10K, all vertices are associated with itemsets of size 1. This allows us to compare our implementation with SLEUTH [25]. In T10K-3 and T10K-5, vertices are associated with itemsets of size 3 and 5 respectively. In T10K-1/10, vertices are associated with itemsets of size randomly selected between 1 and 10, while in T10K-1/20, itemsets' size vary from 1 to 20.

5.2 Web Logs Datasets

We built a dataset on logs given by our university following the method described By Zaki [24]. However, instead of labeling nodes with URLs of the browsed pages, we associated them with itemsets representing keywords of their content. The dataset is composed of 126,396 attributed trees with itemsets of size 10 (10 keywords by page).

5.3 Performance Evaluation

Fig. 4 shows the execution time for mining c-closed sets of induced unordered patterns using IMIT_CONTENT_CLOSED on our five synthetic datasets. For comparison, we added in the figure the execution time of SLEUTH, a reference implementation of equivalence class extension paradigm [25], on the T10K dataset. IMIT_CONTENT_CLOSED is about two times slower than SLEUTH for all support values except the smallest ones where SLEUTH is penalized by the cost of joining millions of frequent patterns.

Although IMIT_CONTENT_CLOSED is slower than SLEUTH, these results are satisfactory because our algorithm is designed to mine attributed trees.

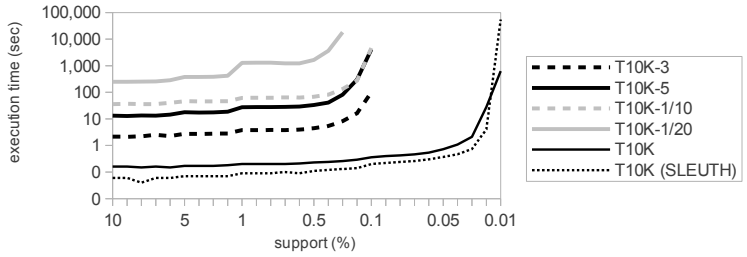

Fig. 4. Execution time of IMIT_CONTENT_CLOSED for mining c-closed sets of induced unordered patterns on 5 synthetic datasets

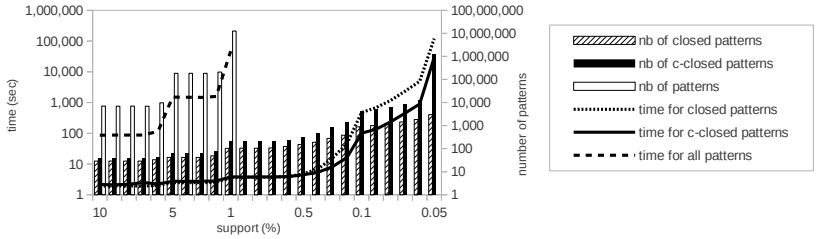

Fig. 5. Induced unordered mining with 3 versions of IMIT on T10K-3 dataset

As such, it is normal to perform worst on mining labeled trees than dedicated implementations. The memory footprint of our algorithm is twice as SLEUTH's one.

The figure also shows that mining attributed trees is extremely more computing intensive than mining labeled trees; and the difference is largely underestimated because only c-closed patterns were mined. Mining all patterns generates a huge number of solutions and takes a long time. To give an idea, mining the T10K-3 dataset with a minimum support of 1% outputs 12 millions patterns in 15 hours (Fig. 5). Mining c-closed atrees allows to reduce both the number of patterns and the execution time. Thus, at 1% minimum support, 200 c-closed patterns are found in 4 seconds.

As shown in the same figure, the search for closed patterns allows to reduce further the number of patterns. At 1% minimum support, for example, the number of patterns drops to 103. However, because of the costly subtree isomorphism checks, in return, performances collapse when patterns become numerous. The result is that the difference in computation time increases as the minimum support decreases.

Fig. 6 show the execution time and number of c-closed patterns in the weblogs dataset. This dataset is much larger than synthetic datasets used before and its mining cannot be performed with a minimum support of less than 10% in a reasonable amount of time. Mining the weblog dataset with a minimum support of 6% lasts 6 hours and returns 360 patterns.

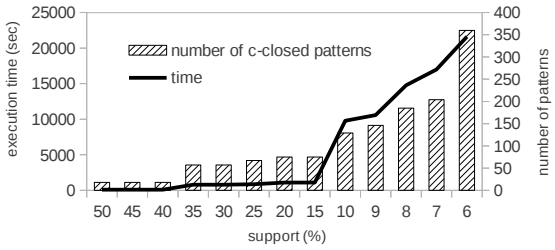

Fig. 6. Performances of IMIT_CONTENT_CLOSED for mining c-closed sets of induced unordered patterns on weblogs datasets

6 Conclusion and Perspectives

In this paper, we introduce the problem of mining attributed trees. We investigate methods enumerating all frequent patterns or only closed ones, but these methods proved inefficient because of, in the first case, the huge number of patterns returned, and in the second case, the cost of subtree isomorphism checks. Finally, we propose a condensed representation of frequent atrees that is defined with respect to itemset inclusion. This representation allows to drastically reduce both the number of patterns and the execution time. We evaluate the efficiency of the proposed algorithm, IMIT_CONTENT_CLOSED, and show that it successfully extract frequent patterns in large datasets. One future work is to extend the proposed algorithm to effectively mine frequent closed patterns. Another future work consists in developing similar methods for mining more complex structures such as attributed graphs.

Acknowledgments. This work was funded by French contract ANR-2010-COSI-012 FOSTER.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. SIGMOD Rec. 22(2), 207–216 (1993)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: ICDE, pp. 3–14 (1995)
3. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: SDM (2002)
4. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: KDD, pp. 429–435 (2002)
5. Balcázar, J.L., Bifet, A., Lozano, A.: Mining frequent closed rooted trees. Mach. Learn. 78(1-2), 1–33 (2010)
6. Chehreghani, M.H.: Efficiently mining unordered trees. In: ICDM, pp. 111–120 (2011)
7. Chi, Y., Muntz, R.R., Nijssen, S., Kok, J.N.: Frequent subtree mining - an overview. Fundam. Inf. 66(1-2), 161–198 (2004)

8. Chi, Y., Yang, Y., Muntz, R.R.: Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: SSDBM, pp. 11–20 (2004)
9. Chi, Y., Yang, Y., Xia, Y., Muntz, R.R.: Cmtreeminer: Mining both closed and maximal frequent subtrees. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 63–73. Springer, Heidelberg (2004)
10. Fukuzaki, M., Seki, M., Kashima, H., Sese, J.: Finding itemset-sharing patterns in a large itemset-associated graph. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part II. LNCS (LNAI), vol. 6119, pp. 147–159. Springer, Heidelberg (2010)
11. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. 29(2), 1–12 (2000)
12. Hido, S., Kawano, H.: Amiot: Induced ordered tree mining in tree-structured databases. In: ICDM, pp. 170–177 (2005)
13. Mannila, H., Toivonen, H.: Multiple uses of frequent sets and condensed representations. In: KDD, pp. 189–194 (2005)
14. Miyoshi, Y., Ozaki, T., Ohkawa, T.: Frequent pattern discovery from a single graph with quantitative itemsets. In: ICDM Workshops, pp. 527–532 (2009)
15. Moser, F., Colak, R., Rafiey, A., Ester, M.: Mining cohesive patterns from graphs with feature vectors. In: SDM, pp. 593–604 (2009)
16. Mougel, P.-N., Rigotti, C., Gandrillon, O.: Finding collections of k-clique percolated components in attributed graphs. In: Tan, P.-N., Chawla, S., Ho, C.K., Bailey, J. (eds.) PAKDD 2012, Part II. LNCS (LNAI), vol. 7302, pp. 181–192. Springer, Heidelberg (2012)
17. Nijssen, S., Kok, J.N.: Efficient discovery of frequent unordered trees. In: First International Workshop on Mining Graphs, Trees and Sequences (MGTS) (2003)
18. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 398–416. Springer, Heidelberg (1998)
19. Termier, A., Rousset, M.C., Sebag, M.: Dryade: A new approach for discovering closed frequent trees in heterogeneous tree databases. In: ICDM, pp. 543–546 (2004)
20. Termier, A., Rousset, M.C., Sebag, M., Ohara, K., Washio, T., Motoda, H.: Dryade-parent, an efficient and robust closed attribute tree mining algorithm. IEEE Trans. on Knowl. and Data Eng. 20(3), 300–320 (2008)
21. Wang, C., Hong, M., Pei, J., Zhou, H., Wang, W., Shi, B.: Efficient pattern-growth methods for frequent tree pattern mining. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 441–451. Springer, Heidelberg (2004)
22. Washio, T., Motoda, H.: State of the art of graph-based data mining. SIGKDD Explor. Newsl. 5(1), 59–68 (2003)
23. Xiao, Y., Yao, J.F., Li, Z., Dunham, M.H.: Efficient data mining for maximal frequent subtrees. In: ICDM, pp. 379–386 (2003)
24. Zaki, M.J.: Efficiently mining frequent trees in a forest. In: KDD, pp. 71–80 (2002)
25. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundam. Inf. 66(1-2), 33–52 (2004)
26. Zou, L., Lu, Y., Zhang, H., Hu, R.: Prefixtreeespan: a pattern growth algorithm for mining embedded subtrees. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 499–505. Springer, Heidelberg (2006)