Mining Frequent Closed Trees in Evolving Data Streams

Albert Bifet
Department of Computer Science, University of Waikato
and LARCA Research Group, Universitat Politècnica de Catalunya
abifet@cs.waikato.ac.nz

Ricard Gavaldà
LARCA Research Group, Departament de LSI
Universitat Politècnica de Catalunya
gavalda@lsi.upc.edu

October 9, 2009

Abstract

We propose new algorithms for adaptively mining closed rooted trees, both labeled and unlabeled, from data streams that change over time. Closed patterns are powerful representatives of frequent patterns, since they eliminate redundant information. Our approach is based on an advantageous representation of trees and a low-complexity notion of relaxed closed trees, as well as ideas from Galois Lattice Theory. More precisely, we present three closed tree mining algorithms in sequence: an incremental one, IncTreeMiner, a sliding-window based one, WinTreeMiner, and finally one that mines closed trees adaptively from data streams, AdaTreeMiner. By adaptive we mean here that it presents at all times the closed trees that are frequent in the current state of the data stream. To the best of our knowledge this is the first work on mining closed frequent trees in streaming data varying with time. We give a first experimental evaluation of the proposed algorithms.

1 Introduction

The frequent pattern discovery task has been intensely studied during the last decade. It is becoming harder every day, as the size of the pattern datasets is increasing and as we move to data that arrive sequentially instead of being available from the start. In the latter case, we have to accept that the distribution that generates data may vary over time, often in an unpredictable and drastic way, and correspondingly the algorithm must be ready to reconsider the patterns it outputs.

Tree Mining in particular is becoming an important field of research. A strong motivation is that XML patterns are tree patterns. XML is becoming a standard for information representation and exchange over the Internet; the amount of XML data is growing, and it will soon constitute one of the largest collections of human knowledge. Other applications of tree mining appear in chemical informatics [16], computer vision [23], text retrieval [29], bioinformatics [25] [17], and Web analysis [8] [31].

In this paper we take an approach to tree mining based on the notion of closure: we mine frequent closed tree patterns. Sharing some of the attractive features of frequency-based summarization of subpatterns, closure-based mining offers an alternative algorithmic view with both downsides and advantages; among the latter, there are the facts that, first, by imposing closure, the number of frequent patterns is heavily reduced and, second, the possibility appears of developing a mathematical foundation that connects it with lattice-theoretic approaches such as Formal Concept Analysis. A downside, however, is that, at the time of influencing the practice of Data Mining, their conceptual sophistication is higher than that of frequent pattern sets, which are, therefore, preferred often by non-experts.

Data streams model the situations where 1) data arrive in sequence, 2) at high speed, so we have little time to process each item, and 3) are so large that we may not be able to store all of what we see. Several applications naturally generate data streams, a prime example being log records or click-streams in web tracking and personalization. Typically, we assume that some features of the data stream, such as the distribution of the items it contains, change over time, and we want to keep an updated view of the current features. A frequent way to deal with continuous data streams that may evolve over time is to fix some window size W and try to keep updated information about the last W items seen, which can be seen a *sliding window* over the data stream; the window thus defines the portion of the data stream that is considered relevant at any given time. We call such an algorithm *sliding window based.*

We propose a general methodology to identify closed patterns in a data stream, using intersection of patterns and Galois Lattice Theory. We then specialize this methodology to trees, and develop three closed tree mining algorithms: IncTreeMiner, an incremental closed tree mining algorithm; WinTreeMiner, a sliding window closed tree mining algorithm; and finally AdaTreeMiner, an adaptive closed tree mining algorithm. By adaptive we mean here that it presents at all times the closed trees that are frequent in the current state of the data stream, that is, since the last noticeable distribution change occurred.

AdaTreeMiner is a new algorithm that can adaptively mine from data streams that change over time, with no need for the user to enter parameters describing the speed or nature of the change. In place or counters or accumulators, it uses as a black-box a recently proposed algorithm (`ADWIN`) [5] for detecting change and keeping updated statistics from a data stream with rigorous performance guarantees.

The rest of the paper is organized as follows. We discuss related work in

Section 2. Sections 3 and 4 give background and introduce our closure operator and its properties needed for our algorithms. Section 5 introduces the general mining framework and Section 6 shows how to adapt this framework to deal with distribution change. Section 7 shows its application to tree structures. Experimental results are given in Section 8, and some conclusions in Section 9.

2 Related Work

There is a large body of work done on itemset mining. An important part of the most recent work applies to data streams; see the survey [10] and the references there. We can divide these data stream methods in two different classes depending on whether they use a landmark window, containing all the examples seen so far, or a sliding window, as described before. Only a small part of these methods deal with frequent closed mining. Moment [11], CFI-Stream [19] and IncMine [9] are the state-of-art algorithms for mining frequent closed itemsets over a sliding window. CFI-Stream stores only closed itemsets in memory, but maintains all closed itemsets as it does not apply a minimum support threshold, with the corresponding memory penalty. Moment stores much more information besides the current frequent closed itemsets, but it has a minimum support threshold to reduce the quantity of patterns found. IncMine proposes a notion of semi-FCIs that consists in increasing the minimum support threshold for an itemset as it is retained longer in the window.

There have been subsequent efforts in moving towards closure-based mining on structured data, particularly sequences, trees, and graphs. One of the differences with closed itemset mining stems from the fact that uniqueness of set-theoretic intersection no longer holds: whereas the intersection of two sets is a set, the intersection of two sequences or two trees is not one sequence or one tree; an example will be given in Section 7.

This makes it nontrivial to justify the word “closed” in terms of a standard closure operator. Many papers resort to a support-based notion of closure of a tree or sequence [12]; others (like [1]) choose a variant of trees where a closure operator between trees can be actually defined (via least general generalization). In some cases, the trees are labeled, and strong conditions are imposed on the label patterns (such as nonrepeated labels in tree siblings [27] or nonrepeated labels at all in sequences [15]). Chi et al. proposed CMTreeMiner [12], the first algorithm to discover all closed and maximal frequent subtrees without first discovering all frequent subtrees. CMTreeMiner shares many features with CloseGraph [30], and works for labeled trees with the “induced” notion of subtree; see explanation in Section 7.

A lot of research work exists on XML pattern mining. Asai et al. [2] present StreamT, an online tree mining algorithm that uses a forgetting model and is able to maintain frequent trees over a sliding window, but it extracts all frequent trees, not just closed ones. Hsieh et al. [18] propose STMer, an alternative to StreamT to deal with frequent trees over data streams, but it is not sliding-window based as it never forgets any counts, so it is not adequate for streams that may change over time. In [14], Feng et al. present SOLARIA*, a frequent closed XML query pattern mining algorithm; it is not an incremental method. Li et al [22] present Incre-FXQPMiner, which mines frequent XML query patterns incrementally; however, it does not obtain the closed XML queries, neither it is sliding-window based.

Labeled trees are trees in which each vertex is given a unique label. Unlabeled trees are trees in which each vertex has no label, or there is a unique label for all vertices. A comprehensive introduction to the algorithms on unlabeled trees can be found in [28].

In [7] a XML tree classifier is presented that uses frequent closed and maximal trees as features; to extract them, it uses a preprocessor based on the approach presented in this paper.

To the best of our knowledge this is the first work dealing with mining frequent closed trees in streaming data that evolve with time.

3 Preliminaries

3.1 Patterns

Following Formal Concept Analysis usage, we are interested in (possibly infinite) sets of patterns, endowed with a partial order relation $\preceq$ among these patterns.

The set of all patterns will be denoted with $\mathcal{T}$, but actually all our developments will proceed in some finite subset of $\mathcal{T}$ which will act as our universe of discourse.

Given two patterns t and t', we say that t is a *subpattern* of t', or t' is a *super-pattern* of t, if $t \preceq t'$. Two patterns t, t' are said to be *comparable* if $t \preceq t'$ or $t' \preceq t$. Otherwise, they are incomparable. Also we write $t \prec t'$ if t is a proper subpattern of t' (that is, $t \preceq t'$ and $t \neq t'$).

The input to our data mining process is a dataset $\mathcal{D}$ of transactions, where each transaction $s \in \mathcal{D}$ consists of a transaction identifier, *tid*, and a pattern. The dataset is a finite set in the standard setting, and a potentially infinite sequence in the data stream setting. Tids are supposed to run sequentially from 1 to the size of $\mathcal{D}$. From that dataset, our universe of discourse $\mathcal{U}$ is the set of all patterns that appear as subpattern of some pattern in $\mathcal{D}$.

Following standard usage, we say that a transaction s *supports* a pattern t if t is a subpattern of the pattern in transaction s. The number of transactions in the dataset $\mathcal{D}$ that support t is called the *support* of the pattern t. A subpattern t is called *frequent* if its support is greater than or equal to a given threshold *min_supp*. The frequent subpattern mining problem is to find all frequent subpatterns in a given dataset. Any subpattern of a frequent pattern is also frequent and, therefore, any superpattern of a nonfrequent pattern is also nonfrequent (the *antimonotonicity* property).

We define a frequent pattern t to be *closed* (implicitly, w.r.t. to $\mathcal{D}$) if none of its proper superpatterns has the same support as it has in $\mathcal{D}$. Generally, there are much fewer frequent closed patterns than frequent ones. In fact, we can obtain all frequent subpatterns with their support from the set of frequent closed subpatterns with their supports. So, the set of frequent closed subpatterns maintains the same information as the set of all frequent subpatterns.

Other possible definitions of frequent closed pattern are the following:

- a frequent pattern is closed if it is in the intersection of all transactions that contain it.
- a frequent pattern is closed if no super-pattern is contained in exactly the same elements of $\mathcal{D}$ as itself.

Examples of pattern classes are itemsets, sequences, and trees [32]. Trees, viewed as patterns, are discussed in more detail in Section 7.

3.2 Relaxed support

Song et al.[26] introduced the concept of relaxed frequent itemset; we adapt it to general pattern mining. The support space of all subpatterns can be divided into $n = \lceil 1/\epsilon_r \rceil$ intervals, where ϵ_r is a user-specified relaxing factor, and each interval can be denoted by $\mathcal{I}_i = [l_i, u_i)$, where $l_i = (n-i) * \epsilon_r \geq 0$, $u_i = (n-i+1) * \epsilon_r \leq 1$ and $i \leq n$. Then a subpattern t is called a *relaxed closed subpattern* if and only if there exists no proper superpattern t' of t whose support belongs to the same interval $\mathcal{I}_i$.

Mining for relaxed (rather than strict) closed patterns may greatly reduce the number of closed subpatterns in data streams where approximation is acceptable.

We can define *relaxed support* as a mapping from all possible dataset supports to the set of relaxed intervals. We apply it to our mining algorithms replacing the calls to support values with calls to relaxed support values.

We introduce the concept of logarithmic relaxed frequent pattern, by defining $l_i = \lceil c^i \rceil$, $u_i = \lceil c^{i+1} - 1 \rceil$ for the value of c generating n intervals. Now, the mapping from supports to relaxed supports is a logarithmic function. It happens often that the number of patterns decreases roughly exponentially as we increase the support threshold. In this case, the notion of logarithmic support may be more appropiate than the linear one, to have roughly equally populated intervals.

4 Closure Operators on Patterns

In this section we develop our approach for closed pattern mining based on the use of closure operators. We obtain a notion of closed pattern using intersection and not only support, so antimonotonicity is not the only mathematical property exploited. Our approach relies on much richer mathematics, which, as usual, leads to more powerful algorithmics.

Definition 1. *Let $B = \{t_1, \ldots, t_n\} \in \mathcal{T}$. A pattern is a* common *subpattern of the patterns in B if it is subpattern of all the patterns in B. A subpattern is* maximal *in B if it is common, and it is not a subpattern of any other common subpattern of the patterns in B. The* intersection *of a set of patterns B, denoted $t_1 \cap \ldots \cap t_n$, is the set of all maximal subpatterns in B.*

Definition 2. *The* closure *of a pattern t for a dataset $\mathcal{D}$ denoted by $\Delta_{\mathcal{D}}(t)$ is the intersection of all transactions in $\mathcal{D}$ that contain it.*

We can relate the notion of closure to the notion of closed pattern based on support, as previously defined, as follows: a pattern t is closed if it is in its closure $\Delta_{\mathcal{D}}(t)$.

Proposition 1. *Adding a pattern transaction to a dataset of patterns $\mathcal{D}$ does not decrease the number of closed patterns for $\mathcal{D}$.*

Proof. All previously closed patterns remain closed: A closed pattern would become non-closed if one of its superpatterns reached the same support, but that is not possible because every time the support of a pattern increases, the support of all its subpatterns also increases. □

Proposition 2. *A pattern transaction t added to a dataset of patterns $\mathcal{D}$ will be a closed pattern for $\mathcal{D}$.*

Proof. The supports of superpatterns of pattern t are lower than or equal to the support of pattern t. A pattern transaction t added would be non-closed if one of its superpatterns reach the same support, but that is not possible because only t is added and none of its supperpatterns is added. □

Proposition 3. *Adding a transaction with pattern t to a dataset of patterns $\mathcal{D}$ where t is closed does not modify the number of closed patterns for $\mathcal{D}$.*

Proof. Let t be the pattern to be added and closed in $\mathcal{D}$, and s a subpattern of t. If s is closed then it will remain closed, as no one of its superpatterns will reach the same support. Suppose s is non-closed and let $t_1, \ldots, t_k$ be the closed superpatterns immediately above s. Since s is non-closed, we know that the support of s equals that of t_i, for some i. For every j, if $t_j \not\preceq t$, the support of s must be larger than that of t_j, because every transaction containing t is counted in the support of s but not in that of t_j. So we know that the index i above must correspond to some t_i with $t_i \preceq t$. Then adding t to $\mathcal{D}$ increases, in particular, the supports of t_i and s by exactly 1, their supports remain equal, and s remains non-closed. □

Proposition 4. *Deleting a pattern transaction from a dataset of patterns $\mathcal{D}$ does not increase the number of closed patterns for $\mathcal{D}$.*

Proof. All the previous non-closed patterns remain non-closed: A necessary condition for a non-closed pattern to become closed is that a superpattern with the same support modifies their support, but this is not possible because every time we decrease the support of a superpattern we also decrease the support of this pattern. □

Proposition 5. *Deleting a pattern transaction that is repeated in a dataset of patterns $\mathcal{D}$ does not modify the number of closed patterns for $\mathcal{D}$.*

Proof. By Proposition 2, transactions patterns added to $\mathcal{D}$ are closed. By Proposition 3, adding a transaction with a previously closed pattern to a dataset of patterns $\mathcal{D}$ does not modify the number of closed patterns for $\mathcal{D}$. So deleting from $\mathcal{D}$ a pattern that is closed and repeated cannot change the number of closed patterns either. □

Proposition 6. *Let $\mathcal{D}1$ and $\mathcal{D}2$ be two datasets of patterns. A pattern t is closed for $\mathcal{D}1 \cup \mathcal{D}2$ if and only if it is in the intersection of its closures $\Delta_{\mathcal{D}1}(t)$ and $\Delta_{\mathcal{D}2}(t)$.*

Proof. A pattern t is closed for $\mathcal{D}1 \cup \mathcal{D}2$ only if it is in closure $\Delta_{\mathcal{D}1\cup\mathcal{D}2}(t)$, the intersection of all transactions of $\mathcal{D}1$ and $\mathcal{D}2$ that contain it. The closure $\Delta_{\mathcal{D}1\cup\mathcal{D}2}(t)$ of pattern t is the intersection of all the transactions of $\mathcal{D}1 \cup \mathcal{D}2$ that contain it. Then $\Delta_{\mathcal{D}1\cup\mathcal{D}2}(t)$ is equal to the intersection of all the transactions of $\mathcal{D}1$ that contains it, and all the transactions of $\mathcal{D}2$ that contains it. Hence, $\Delta_{\mathcal{D}1\cup\mathcal{D}2}(t)$ is the intersection of the closure of t for $\mathcal{D}1$ and the closure of t for $\mathcal{D}2$.

□

We use Proposition 6 as a closure checking condition when adding a set of transactions to a dataset of patterns.

Corollary 1. *Let $\mathcal{D}1$ and $\mathcal{D}2$ be two datasets of patterns. A pattern t is closed for $\mathcal{D}1 \cup \mathcal{D}2$ if and only if*

- *t is a closed pattern for $\mathcal{D}1$, or*
- *t is a closed pattern for $\mathcal{D}2$, or*
- *t is a subpattern of a closed pattern in $\mathcal{D}1$ and a closed pattern in $\mathcal{D}2$ and it is in $\Delta_{\mathcal{D}_1\cup\mathcal{D}_2}(\{t\})$.*

In summary, the closure-based approach gives us elegant and algorithmically useful conditions that are for checking whether a pattern is closed.

5 Closed Pattern Mining

5.1 Incremental Closed Pattern Mining

In this subsection we propose a new method to do incremental closed pattern mining. Let D_1 be the transaction set seen so far, whose set of closed patterns $T1$ we have computed already. Suppose that a new batch of patterns D_2 arrives. We compute its set of closed patterns, $T2$, and then we update the closed pattern set to that of $D_1 \cup D_2$ using procedure CLOSED_SUBPATTERN_MINING_ADD, shown in Figure 1.

```
CLOSED_SUBPATTERN_MINING_ADD($T_1, T_2, min\_supp, T$)
    Input: Pattern sets $T1$ and $T2$, and $min\_supp$;
        $T1$ and $T2$ are the frequent closed patterns of some datasets $D1$, $D2$
    Output: The set $T$ of frequent closed patterns of dataset $D1 \cup D2$
    $T \leftarrow T1$
    for every $t$ in $T2$ in size-ascending order
        do if $t$ is closed in $T1$
            then $support_T(t) += support_{T2}(t)$
                for every $t'$ that is a subpattern of $t$
                    do if $t'$ is in $T1$
                        then if $t'$ support is not updated
                            then insert $t'$ into $T$
                                $support_T(t') += support_{T2}(t')$
                            else
                                skip processing $t'$ and all its subpatterns
        else insert $t$ into $T$
            for every $t'$ that is a subpattern of $t$
                do if $t'$ support is not updated
                    then if $t'$ is in $T1$
                        then $support_T(t') += support_{T2}(t')$
                    if $t'$ is closed
                        then insert $t'$ into $T$
                            $support_T(t') += support_{T2}(t')$
                    else skip processing $t'$ and all its subpatterns
    delete from $T$ patterns with support below $min\_supp$
    return $T$
```

Figure 1: The Closed Subpattern Mining Add algorithm

```
CLOSED_SUBPATTERN_MINING_DELETE($T1, T2, min\_supp, T$)
    Input: Pattern sets $T1$ and $T2$, and $min\_supp$;
        $T1$ and $T2$ are the frequent closed patterns of some datasets $D1$, $D2$
    Output: The set $T$ of frequent closed patterns of dataset $D1 \setminus D2$
 1  $T \leftarrow T1$
 2  for every $t$ in $T2$ in size-ascending order
 3      do for every $t'$ that can be obtained deleting nodes from $t$
 4          do if $t'$ support is not updated
 5              then if $t'$ is in $T1$
 6                  then if $t'$ is not closed
 7                      then delete $t'$ from $T$
 8                      else  $support_T(t') -= support_{T2}(t')$
 9              else  skip processing $t'$ and all its subpatterns
10  delete from $T$ patterns with support below $min\_supp$
11  return $T$
```

Figure 2: The Closed Subpattern Mining Delete algorithm

In words, let $T1$ be the existing set of closed patterns, and $T2$ those coming from the new batch D_2. For each closed pattern in $T2$, we check whether the pattern is closed in $T1$. If it is closed, we update its support and that of all its subpatterns, as justified by Proposition 3. If it is not closed, as it is closed for $T2$, we add it to the closed pattern set, as justified by Corollary 1, and we check for each of its subpatterns whether it is closed or not. In line 18, we use Proposition 6 to do the closure-checking. As we check all the subpatterns of $T2$ in size-ascending order, we know that all closed subpatterns of t have been checked before.

Note that this is a totally generic algorithm for pattern mining. The best (most efficient) data structure to do this task will depend on the kind of patterns. In general, a lattice is the default option, where each lattice node is a pattern with its support, a list of its closed superpatterns, and a list of its closed subpatterns.When merging the closed patterns, the parameter min_supp becomes important. If two transaction sets are merged, a pattern that is infrequent in both of them may become frequent in their union. We take care of that using $min\_supp' = \alpha \cdot min\_supp < min\_supp$, so that a pattern can be in the the union of the transaction sets not only if it is closed in only one of them.

5.2 Closed pattern mining over a sliding window

By adding a method to delete a set of transactions, we can adapt our method to use a sliding window of pattern transactions.

Figure 2 shows the pseucodode of CLOSED_SUBPATTERN_MINING_DELETE. We check for every t pattern in $T2$ in ascending order if its subpatterns are still closed or not after deleting some transactions. We can look for a closed superpattern with the same support.The lattice structure supports this operation well. We can delete a transaction one by one, or delete a batch of transactions of the sliding window. We delete transactions one by one to avoid recomputing the frequent closed patterns of each batch of transactions.

6 Adaptive closed pattern mining

In this section we present a new method for dealing with distribution change in pattern mining, using ADWIN [5], an algorithm for detecting change and dynamically adjusting the length of a data window. First we briefly review the ADWIN algorithm and then we describe our method combining the previous sliding window pattern mining algorithms and ADWIN.

6.1 The ADWIN algorithm

Recently, we proposed an algorithm termed ADWIN (for Adaptive Windowing) that solves in a well-specified way the problem of tracking the average of a stream of bits or real-valued items. ADWIN keeps a variable-length window of recently seen items, with the property that the window has at all times the maximal length statistically consistent with the hypothesis "there has been no change in the average value inside the window".

The inputs to ADWIN are a confidence value $\delta \in (0,1)$ and a (possibly infinite) sequence of real values $x_1, x_2, x_3, \ldots, x_t, \ldots$ The value of x_t is available only at time t. Each x_t is generated according to some distribution D_t, independently for every t. We denote with μ_t the expected value of x_t when it is drawn according to D_t. We assume that x_t is always in $[0,1]$; by an easy rescaling, we can handle any case in which we know an interval $[a,b]$ such that $a \leq x_t \leq b$ with probability 1. Nothing else is known about the sequence of distributions D_t; in particular, μ_t is unknown for all t.

Algorithm ADWIN uses a sliding window W with the most recently read x_i. Let $\hat{\mu}_W$ denote the (known) average of the elements in W, and μ_W the (unknown) average of μ_t for $t \in W$. We use $|W|$ to denote the length of a (sub)window W.

Algorithm ADWIN is presented in Figure 3. The idea of ADWIN method is simple: whenever two "large enough" subwindows of W exhibit "distinct enough" averages, one can conclude that the corresponding expected values are different, and the older portion of the window is dropped. The meaning of "large enough" and "distinct enough" can be made precise again by using the Hoeffding bound. The test eventually boils down to whether the average of the two subwindows is larger than a variable value ϵ_{cut} computed as follows

$$\begin{aligned} m &:= \frac{2}{1/|W_0| + 1/|W_1|} \\ \epsilon_{cut} &:= \sqrt{\frac{1}{2m} \cdot \ln \frac{4|W|}{\delta}} \, . \end{aligned}$$

```
ADWIN: ADAPTIVE WINDOWING ALGORITHM
1  Initialize Window $W$
2  for each $t > 0$
3       do $W \leftarrow W \cup \{x_t\}$ (i.e., add $x_t$ to the head of $W$)
4          repeat Drop elements from the tail of $W$
5            until $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| < \epsilon_{cut}$ holds
6              for every split of $W$ into $W = W_0 \cdot W_1$
7          output $\hat{\mu}_W$
```

Figure 3: Algorithm ADWIN.

where m is the harmonic mean of $|W_0|$ and $|W_1|$.

The main technical result in [5] about the performance of ADWIN is the following theorem, that provides bounds on the rate of false positives and false negatives for ADWIN:

Theorem 1. *With ϵ_{cut} defined as above, at every time step we have:*

1. (False positive rate bound). *If μ_t has remained constant within W, the probability that* ADWIN *shrinks the window at this step is at most δ.*
2. (False negative rate bound). *Suppose that for* some *partition of W in two parts W_0W_1 (where W_1 contains the most recent items) we have $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$. Then with probability $1 - \delta$* ADWIN *shrinks W to W_1, or shorter.*

This theorem justifies us in using ADWIN in two ways:

- as a *change detector*, since ADWIN shrinks its window if and only if there has been a significant change in recent times (with high probability)
- as an *estimator* for the current average of the sequence it is reading since, with high probability, older parts of the window with a significantly different average are automatically dropped.

ADWIN is parameter- and assumption-free in the sense that it automatically detects and adapts to the current rate of change. Its only parameter is a confidence bound δ, indicating how confident we want to be in the algorithm's output, inherent to all algorithms dealing with random processes. This is one reason for our choice, since many other simpler methods for detecting change (such as CUSUM, EWMA, and others) require some parameter from the user balancing reaction time and anticipated change rate.

Also important for our purposes, ADWIN does not maintain the window explicitly, but compresses it using a variant of the exponential histogram technique in [13]. In particular, it keeps a window of length W using only $O(\log W)$ memory rather than the $O(W)$ one expects from a naïve implementation. The processing time per item is also $O(\log W)$.

6.2 Closed pattern mining in the presence of distribution change

We propose two strategies to deal with distribution change:

1. Using a sliding window of transactions with an `ADWIN` estimator deciding the size of the window by monitoring the number of different closed trees. The number of closed trees is a good measure of the distribution change as it will always increase when we add new transactions with different frequent closed patterns, by Proposition 1.

2. Maintaining an `ADWIN` estimator for the support of each closed pattern in the lattice structure.

The second strategy has higher computational cost, but gives us a more precise control, on a one-to-one basis, of which trees are changing their distribution in the data stream and which ones are not. In both strategies we use CLOSED_SUBPATTERN_MINING_ADD to add transactions. In the first strategy we use CLOSED_SUBPATTERN_MINING_DELETE to delete transactions as we maintain a sliding window of transactions.

In the second strategy, we do not delete transactions. Instead, each `ADWIN` monitors the support of a closed pattern. When it detects a change, we can conclude reliably that the support of this pattern seems to be changing in the data stream in recent times. If the support decreases, the number of closed patterns may decrease and we have to delete the non-closed patterns from the lattice. We check whether it and all its subpatterns are still closed by trying to find a superpattern with the same support.

In order to obtain frequent closed patterns with a *min_supp* support, we will add to our algorithms a *min_supp* support checking condition, to delete and reduce the number of closed patterns in the lattice.

7 Closed tree mining

In this section we apply the general framework above specifically by considering rooted, unranked tree patterns. *Trees* are connected acyclic graphs, *rooted trees* are trees with a vertex singled out as the root, and *unranked* trees are trees with unbounded arity. We say that $t_1, \ldots, t_k$ are the *components* of tree t if t is made of a node (the root) joined to the roots of all the t_i's. We can distinguish betweeen the cases where the components at each node form a sequence (*ordered* trees) or just a set (*unordered* trees). We will deal with ordered and unordered, labeled and unlabeled trees.

An *induced subtree* of a tree t is any connected subgraph rooted at some node v of t whose vertices and edges are subsets of those of t. A *top-down subtree* of a tree t is an induced subtree of t which contains the root of t. An *embedded subtree* of a tree t is any tree rooted at some node v of t that preserves the ancestor-descendant relationship among vertices as in t. We will always use the notion of induced subtree. Formally, let t be a rooted tree with vertex set V

and edge set E, and t' a rooted tree with vertex set V' and edge set E'. Tree t' is an *induced subtree* (or simply a *subtree*) of t (written $t' \preceq t$) if and only if 1) $V' \subseteq V$, 2) $E' \subseteq E$, and 3) the labeling of V' (if present) is preserved in t. This notation can be extended to sets of trees $A \preceq B$: this means that for every $t \in A$, there is some $t' \in B$ for which $t \preceq t'$.

As mentioned in Section 2 the intersection of two trees is not necessarily one tree. Indeed: A common subtree of two trees is a tree that is subtree of both; it is a maximal common subtree if it is not a proper subtree of any other common subtree. Two unlabeled trees have always some maximal common subtree but, as is shown in Figure 4, this common subtree does not need to be unique.

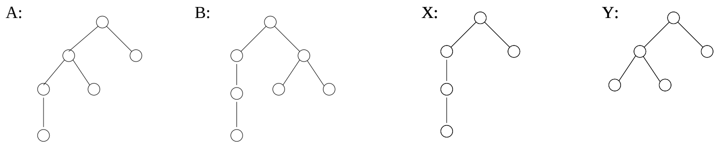
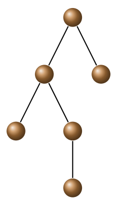


Figure 4: Trees X and Y are maximal common subtrees of A and B.

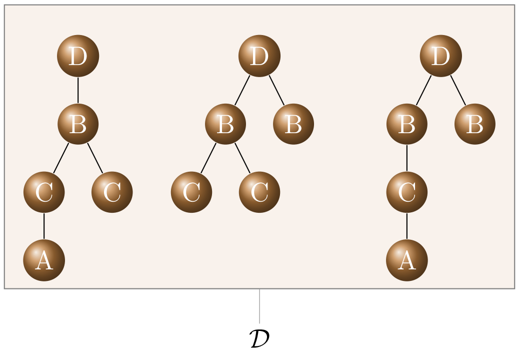
In fact, both trees X and Y in Figure 4 have the maximum number of nodes among the common subtrees of A and B. As per Definition 2, the *intersection* of a set of trees is the set of all maximal common subtrees of the trees in the set.

We represent unlabeled trees using level representations [20, 21]. The *level representation* of a tree is a sequence over a countably infinite alphabet, namely, the set of natural numbers. This encoding basically corresponds to a preorder traversal of t, where each number of the sequence represents the level or depth of the current node in the traversal. As an example, the level representation of the tree



is the natural sequence $(0, 1, 2, 2, 3, 1)$. Note that, for example, the subsequence $(1, 2, 2, 3)$ corresponds to the induced subtree rooted at the left son of the root. Formally, a *labeled level sequence* is a sequence $((x_1, l_1) \dots, (x_n, l_n))$ of pairs of level numbers and labels such that $x_1 = 0$ and each subsequent number x_{i+1} belongs to the range $1 \leq x_{i+1} \leq x_i + 1$. For every such sequence one can build a tree whose representation is that sequence, hence these sequences are in 1-to-1 correspondence with trees.

To represent labeled trees, we extend this notion to labeled level sequences [3, 24]: now every element of the sequence is a pair formed by a natural number, and a label, with the sequence formed by taking the first components of all



Tid	Trees
1	$((0,D),(1,B),(2,C),(3,A),(2,C))$
2	$((0,D),(1,B),(2,C),(2,C),(1,B))$
3	$((0,D),(1,B),(2,C),(3,A),(1,B))$

Figure 5: An example dataset

pairs satisfying the condition above. Given two such labeled sequences x, y, we denote with $|x|$ the number of pairs of x, with $x \cdot y$ the sequence obtained as concatenation of x and y, and with x^+ the sequence obtained adding 1 to the first components of each pair in x.

For example, $x = ((0,A),(1,B),(2,A),(3,B),(1,C))$ is a level sequence that satisfies $|x| = 6$ and $x = ((0,A)) \cdot ((0,B),(1,A),(2,B))^+ \cdot ((0,C))^+$. Note that, in general, the representation of t as a labelled sequence can be formed recursively from the representations of its subtrees using the pair-building, $\cdot$, and $^+$ operators.

The input to our data mining process is a dataset $\mathcal{D}$ of transactions, where each transaction $s \in \mathcal{D}$ consists of a transaction identifier, *tid*, and a rooted tree. In offline mining, $\mathcal{D}$ is a finite set, and in the datastream setting $\mathcal{D}$ is a potentially infinite sequence. Figure 5 shows a finite dataset example. The closed trees for the dataset of Figure 5 are shown as a Galois lattice in Figure 6.

7.1 Non-Incremental Closed Tree Mining

In [4] the authors presented an algorithm TreeNat for computing frequent and closed trees from a dataset of trees, in a non-incremental way. They represent the potential subtrees to be checked as frequent and closed on the dataset in such a way that extending them by one single node, in all possible ways, corresponds to a clear and simple operation on the representation. The completeness of the procedure is assured, that is, all trees can be obtained in this way. This allows them not having to extend trees that are found to be already nonfrequent.

The pseudocode of this method, Closed_Subtree_Mining, is presented

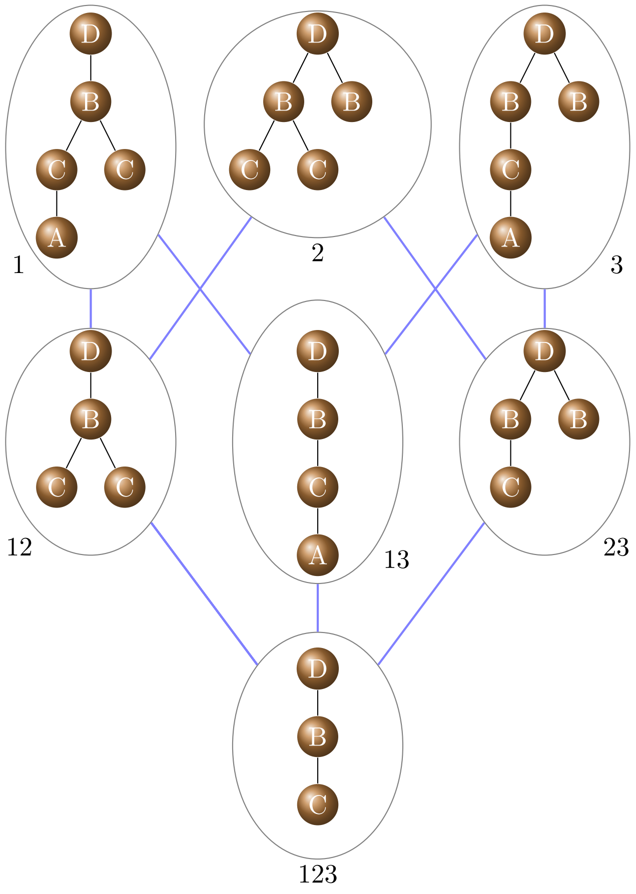


Figure 6: Example of Galois lattice of closed trees

```
CLOSED_SUBTREE_MINING($t, D, min\_supp, T$)
    Input: A tree representation $t$, a tree dataset $D$, and $min\_supp$
    Output: The frequent closed tree set $T$

 1  if $t \neq$ CANONICAL_REPRESENTATIVE($t$)
 2     then return $T$
 3  for every $t'$ that can be extended from $t$ in one step
 4        do if support($t'$) $\geq min\_supp$
 5              do $T \leftarrow$ CLOSED_SUBTREE_MINING($t', D,$
                   $min\_supp, T$)
 6        do if support($t'$) $=$ support($t$)
 7              then $t$ is not closed
 8  if t is closed
 9     then insert $t$ into $T$
10  return $T$
```

Figure 7: The Closed Subtree Mining algorithm

in Figures 7 and 8. Note that the first line of the algorithm is a canonical representative checking, a check that is used frequently in tree mining literature. In [4] the authors selected one of the ordered trees corresponding to a given unordered tree to act as a canonical representative: by convention, this canonical representative has larger trees always to the left of smaller ones.

The main difference of TreeNat with CMTreeMiner is that CMTreeMiner needs to store all occurrences of subtrees in the tree dataset to use its pruning methods, whereas TreeNat does not. The number of occurrences is high if the trees are big, or the number of labels is small. In this case, CMTreeMiner needs more memory and time to process them. Otherwise, if the size of the trees is small, or the number of labels is high then CMTreeMiner outperforms TreeNat, since it can use the power of its pruning methods.

```
CLOSED_MINING($D, min\_supp$)
    Input: A tree dataset $D$, and $min\_supp$
    Output: The set $T$ of closed trees of $D$ with support at least $min\_supp$

1  $t \leftarrow \bullet$
2  $T \leftarrow \emptyset$
3  $T \leftarrow$ CLOSED_SUBTREE_MINING($t, D, min\_supp, T$)
4  return $T$
```

Figure 8: The Closed Unordered Mining algorithm

7.2 Incremental Closed Tree Mining

Following the general framework for patterns presented in Section 5, and adapting it to the tree pattern case, it is easy to derive three unlabeled tree mining algorithms:

- IncTreeMiner-U, an incremental closed tree mining algorithm (this algorithm was called IncTreeNat in [6]),
- WinTreeMiner-U, a sliding window closed tree mining algorithm (this algorithm was called WinTreeNat in [6]),
- AdaTreeMiner-U an adaptive closed tree mining algorithm (this algorithm was called AdaTreeNat in [6])

And for labeled trees, we propose three labeled tree mining algorithms:

- IncTreeMiner-L, an incremental closed tree mining algorithm,
- WinTreeMiner-L, a sliding window closed tree mining algorithm
- AdaTreeMiner-L an adaptive closed tree mining algorithm

The batches are processed using the non-incremental algorithm explained in Subsection 7.1. The main difference between the labeled and unlabeled methods is the tree representation used, see [4]. We use the relaxed notion of closed tree described in Section 3.2 to speed up the mining process.

8 Experimental Evaluation

We tested our algorithms on synthetic and real data, comparing the results with CMTreeMiner [12].

All experiments were performed on a 2.0 GHz Intel Core Duo PC machine with 2 Gigabytes of main memory, running Ubuntu 7.10. As far as we know, CMTreeMiner is the state-of-art algorithm for mining induced frequent closed trees in databases of rooted trees. CMTreeMiner and our algorithms are implemented in C++. The main difference with our approach is that CMTreeMiner is not incremental and only works with induced subtrees, and our method works with both induced and top-down subtrees. In all experiments using `ADWIN`, its confidence parameter δ is set to 0.01.

8.1 Unlabeled Trees

On synthetic data, we use the same dataset as in [12] and [31] for rooted ordered trees restricting the number of distinct node labels to one. We call this dataset TN1, and is generated by the tree generation program of Zaki [31] available from his web page. This program generates a mother tree that simulates a master website browsing tree. Then it assigns probabilities of following its children nodes, including the option of backtracking to its parent, such that the sum of

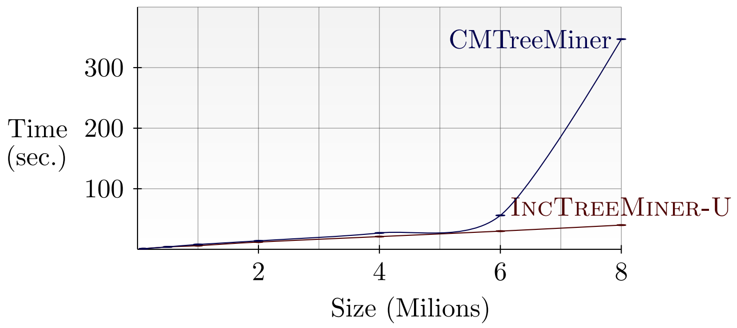


Figure 9: Execution time on the ordered unlabeled tree TN1 dataset

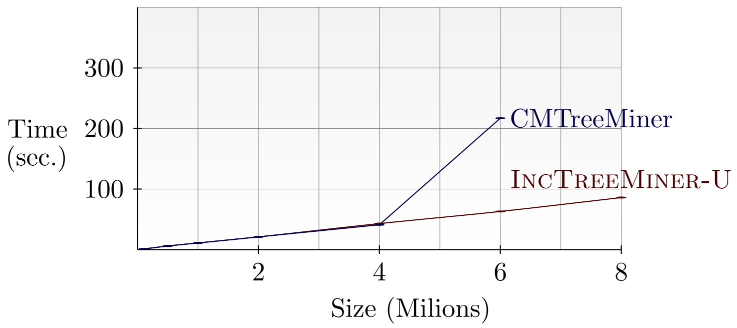


Figure 10: Time used on unordered unlabeled trees, TN1 dataset

all the probabilities is 1. Using the master tree, the dataset is generated creating subtrees by randomly picking subtrees according to these probabilities.

In the TN1 dataset, the parameters are the following: the number of distinct node labels is $N = 1$, the total number of nodes in the tree is $M = 10,000$, the maximal depth of the tree is $D = 10$, the maximum fanout is $F = 10$. The average number of nodes is 3.

The results of our experiments on synthetic data are shown in Figures 9, 10, 11, and 12. We used a batch size of $100,000$ for the IncTreeMiner-U method. We varied the dataset size from $100,000$ to 8 milion, and we observed that as the dataset size increases, IncTreeMiner-U time increases linearly, and CMTreeMiner does much worse than IncTreeMiner-U. At 4 milion samples, in the unordered case, CMTreeMiner needs to use swap memory. After 6 milion samples, CMTreeMiner runs out of main memory and it ends before outputting the closed trees. We observe that as the dataset size increases, CMTreeMiner can not mine datasets bigger than 6 milion trees: not being an incremental method, it must store the whole dataset in memory all the time *in addition*

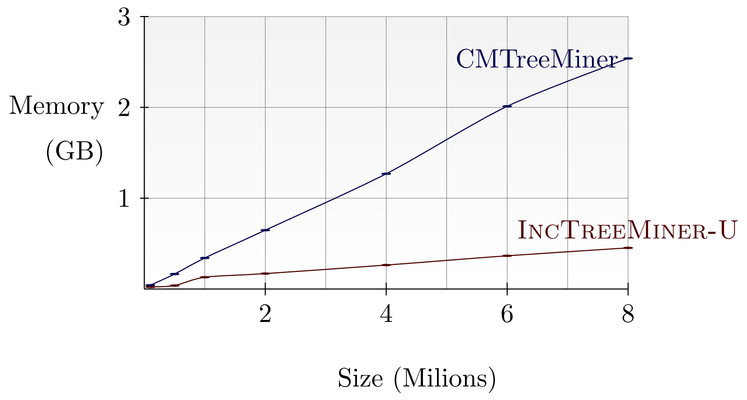


Figure 11: Memory used on ordered unlabeled tree TN1 dataset

to the lattice structure and the occurrences of the trees, in contrast with our algorithms.

Figure 13 shows the result of the second experiment: we take a TN1 dataset of 2 milion trees, and we introduce artificial distribution change changing the dataset trees from sample 500,000 to 1,000,000 and from 1,500,000 to 2,000,000, in order to force a small number of closed trees. We compare IncTreeMiner-U, WinTreeMiner-U with a sliding window of $500,000$ and $1,000,000$, and with AdaTreeMiner-U using ADWIN to monitor the size of the sliding window. We observe that AdaTreeMiner-U detects change faster, and it quickly revises the number of closed trees in its output. On the other hand, the other methods will not really detect change until they have flushed many old trees from their sliding windows, so they take longer to revise the number of closed trees in their output.

To compare the two adaptive methods, we perform a third experiment. We use a data stream of $200,000$ trees, with a static distribution of 20 closed trees on the first $100,000$ trees and 20 different closed trees on the last $100,000$ trees. The number of closed trees remains the same. Figure 14 shows the difference between the two methods. The first one, which monitors the number of closed trees, detects change at sample 111,480 and then it reduces the window size immediately. In the second method there are ADWINs monitoring each tree support; they notice the appearance of new closed trees quicker, but overall the number of closed trees decreases more slowly than in the first method.

Finally, we tested our algorithms on the CSLOGS Dataset, available from Zaki's web page [31]. It consists of web logs files collected over one month at the Department of Computer Science of Rensselaer Polytechnic Institute. The logs touched $13,361$ unique web pages, and the CSLOGS dataset contains $59,691$ trees. The average tree size is 12.

Figure 15 shows the number of closed trees detected on the CSLOGS dataset,

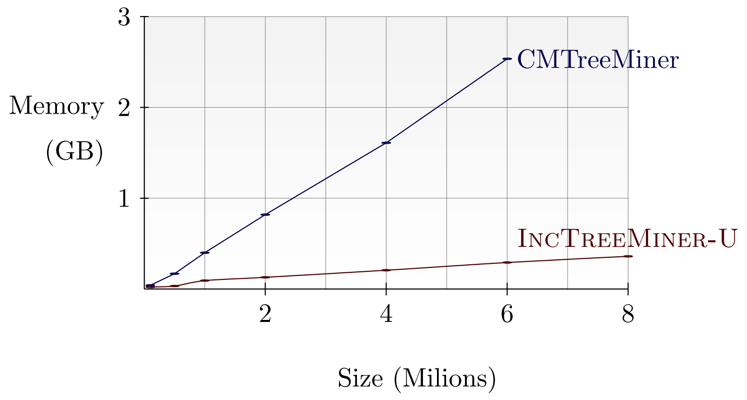


Figure 12: Memory used on unordered unlabeled trees, TN1 dataset

varying the number of relaxed intervals. We see that, in this dataset, support values are distributed in such a way that we find more closed trees using logarithmic relaxed support than using linear relaxed support, but not by a order of magnitude. This higher number of closed trees implies that we obtain more information using the same number of intervals. When the number of intervals is greater than 1,000 the number of intervals is 249, the number obtained using the strict (not relaxed) notion of support.

8.2 Labeled Trees

On synthetic labeled data, we use the same dataset as in [12] and [31] for rooted ordered trees. The synthetic dataset T8M are generated by the tree generation program of Zaki [31]. A mother tree is generated first with the following parameters: the number of distinct node labels $N = 100$, the total number of nodes in the tree $M = 10,000$, the maximal depth of the tree $D = 10$ and the maximum fanout $F = 10$. The dataset is then generated by creating subtrees of the mother tree. In our experiments, we set the total number of trees in the dataset to be from $T = 0$ to $T = 8,000,000$.

The results of our experiments on synthetic data are shown in Figures 16, 17, 18, and 19. We observe that as the dataset size increases, IncTreeMiner-L and CMTreeMiner times are similar and that IncTreeMiner-L uses much less memory than CMTreeMiner. CMTreeMiner can not mine datasets bigger than 8 milion trees: not being an incremental method, it must store the whole dataset in memory all the time *in addition* to the lattice structures, as we already observed for unlabeled trees.

In Figure 20 we compare WinTreeMiner-L with different window sizes to AdaTreeMiner-L on T8M dataset. We observe that the two versions of AdaTreeMiner-L outperforms WinTreeMiner-L for all window sizes.

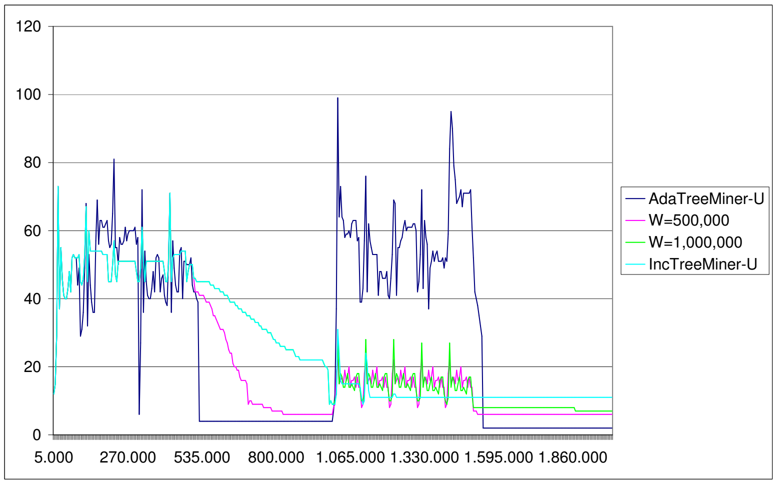


Figure 13: Number of closed trees detected with artificial distribution change introduced

9 Conclusions

We have presented the first efficient algorithms for mining ordered and unordered frequent closed trees on evolving data streams.

If the distribution of the tree dataset is stationary, the best method to use is IncTreeMiner, as we do not need to delete any past transaction. If the distribution may evolve, then a sliding window method is more appropiate. If we know which is the right size of the sliding window, then we can use WinTreeMiner, otherwise AdaTreeMiner would be a better choice, since it does not need the window size parameter. When compared with state-of-the-art, but nonincremental, algorithm CMTreeMiner, we clearly observe the advantage of our algorithm with respect to time and memory consumption, due to not having to store the whole dataset in main memory at all times.

Future work will be to do more experiments varying other tree parameters, and comparing it to other incremental methods as StreamT, if they are available.

10 Acknowledgments

Partially supported by the EU PASCAL2 Network of Excellence, and by the DGICYT MOISES-BAR, TIN2005-08832-C03-03, and MICINN SESAAME-BAR,

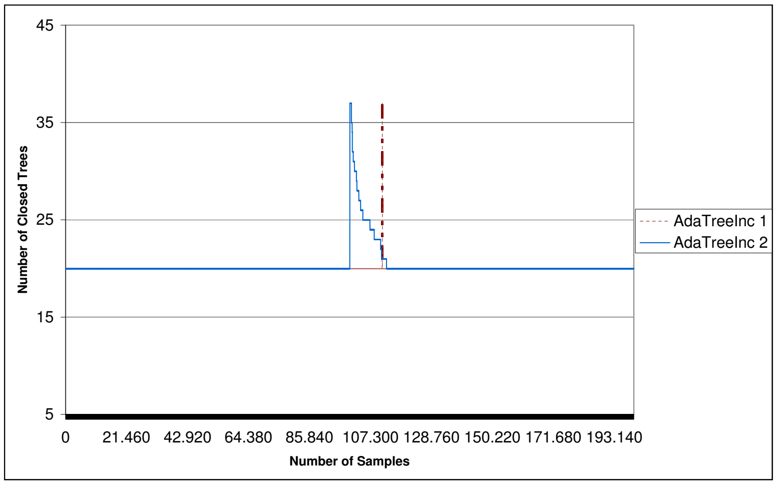


Figure 14: Number of closed trees maintaining the same number of closed datasets on input data

TIN2008-06582-C03-01/TIN, projects. Albert Bifet has been supported by a Formació d'Investigadors (FI) grant through the Grups de Recerca Consolidats (SGR) program of Generalitat de Catalunya.

The authors would like to thank José L. Balcázar for his unconditional support and inspiring ideas and discussions.

A preliminary version of this paper, dealing with unlabeled trees only, was presented in [6]; we are grateful to the reviewers of that conference and the reviewers of the present journal paper for extremely helpful comments.

References

[1] Hiroki Arimura and Takeaki Uno. An output-polynomial time algorithm for mining frequent closed attribute trees. In *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2005.

[2] Tatsuya Asai, Hiroki Arimura, Kenji Abe, Shinji Kawasoe, and Setsuo Arikawa. Online algorithms for mining semi-structured data stream. In

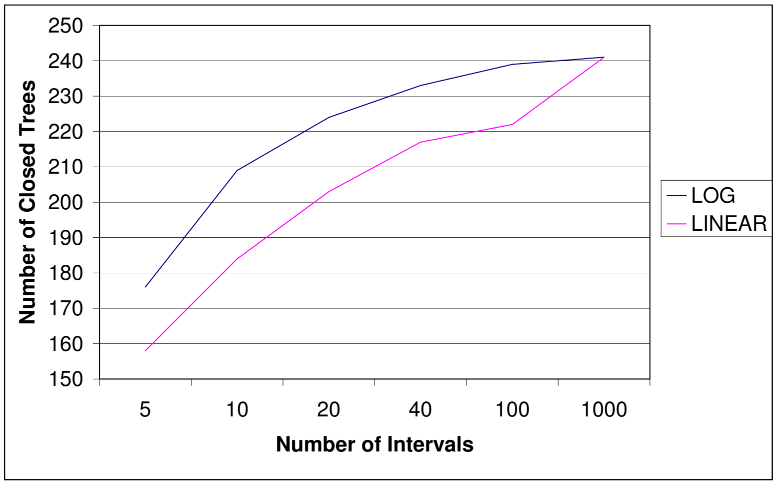


Figure 15: Number of closed trees detected on CSLOGS dataset varying Number of relaxed intervals

ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining, page 27, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[3] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin-Ichi Nakano. Discovering frequent substructures in large unordered trees. In *Discovery Science, 6th International Conference, DS 2003, Sapporo, Japan, October 17-19,2003, Proceedings*, volume 2843 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2003.

[4] José Luis Balcázar, Albert Bifet, and Antoni Lozano. Mining frequent closed rooted trees. *Accepted for publication in Machine Learning Journal*, 2009.

[5] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*. SIAM, 2007.

[6] Albert Bifet and Ricard Gavaldà. Mining adaptively frequent closed unlabeled rooted trees in data streams. In *14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

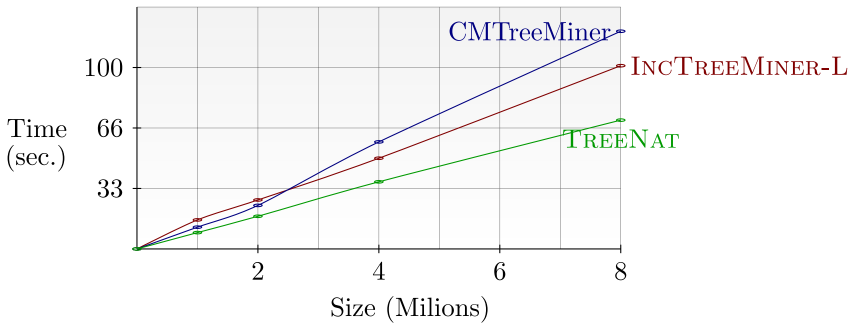


Figure 16: Execution time on ordered labeled tree T8M dataset

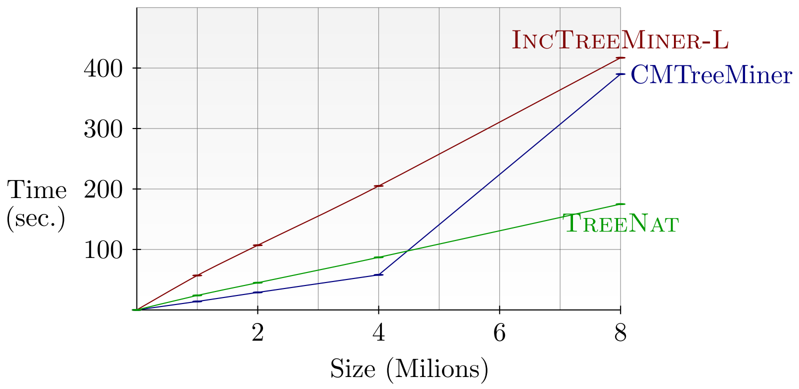


Figure 17: Time used on unordered labeled trees, TN1 dataset

[7] Albert Bifet and Ricard Gavaldà. Adaptive XML tree classification on evolving data streams. In *ECML-PKDD'09*, 2009.

[8] Soumen Chakrabarti. *Mining the Web: Analysis of Hypertext and Semi Structured Data*. Morgan Kaufmann, August 2002.

[9] James Cheng, Yiping Ke, and Wilfred Ng. Maintaining frequent closed itemsets over a sliding window. *Journal of Intelligent Information Systems*, 31(3):191–215, 2008.

[10] James Cheng, Yiping Ke, and Wilfred Ng. A survey on algorithms for mining frequent itemsets over data streams. *Knowledge and Information Systems*, 16(1):1–27, 2008.

[11] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the*

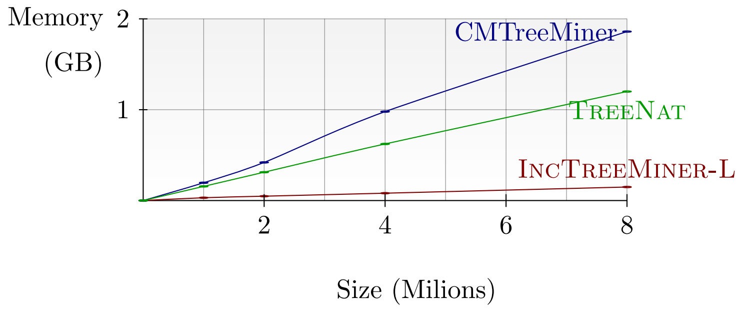


Figure 18: Memory used on ordered labeled tree T8M dataset

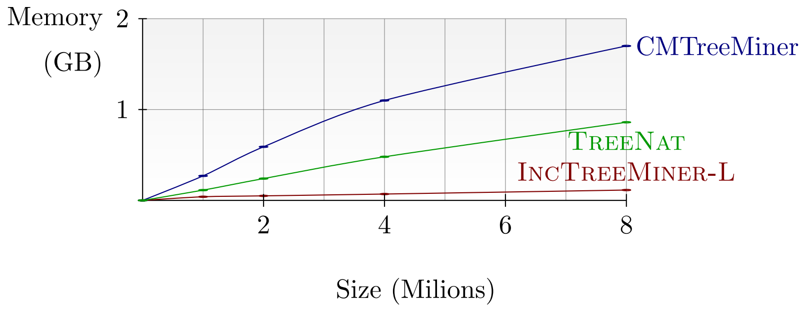


Figure 19: Memory used on unordered labeled trees on T8M dataset

2004 IEEE International Conference on Data Mining (ICDM'04), November 2004.

[12] Yun Chi, Yi Xia, Yirong Yang, and Richard Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *Fundamenta Informaticae*, XXI:1001–1038, 2001.

[13] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 14(1):27–45, 2002.

[14] Jianhua Feng, Qian Qian, Jianyong Wang, and Li-Zhu Zhou. Efficient mining of frequent closed XML query pattern. *J. Comput. Sci. Technol.*, 22(5):725–735, 2007.

[15] Gemma C. Garriga and José L. Balcázar. Coproduct transformations on lattices of closed partial orders. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 336–352. Springer, 2004.

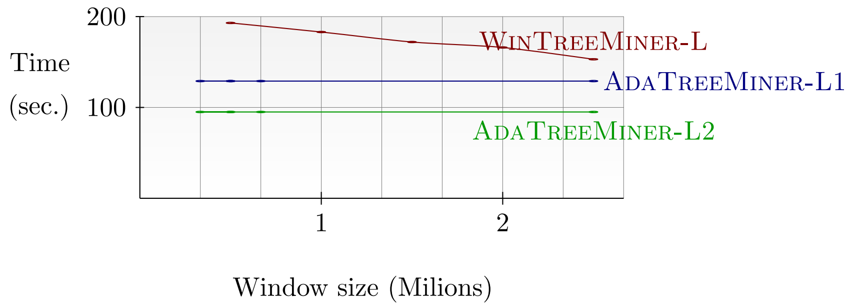


Figure 20: Time on ordered labeled trees on T8M dataset, varying window size

[16] Kosuke Hashimoto, Kiyoko Flora Aoki-Kinoshita, Nobuhisa Ueda, Minoru Kanehisa, and Hiroshi Mamitsuka. A new efficient probabilistic model for mining labeled ordered trees applied to glycobiology. *ACM Trans. Knowl. Discov. Data*, 2(1):1–30, 2008.

[17] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937, pages 177–190, Espoo, Finland, 1995. Springer-Verlag, Berlin.

[18] Mark Cheng-Enn Hsieh, Yi-Hung Wu, and Arbee L. P. Chen. Discovering frequent tree patterns over data streams. In *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*. SIAM, 2006.

[19] Nan Jiang and Le Gruenwald. CFI-Stream: mining closed frequent itemsets in data streams. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 592–597, 2006.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[21] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: The: Generating All Trees–History of Combinatorial Generation*. Addison-Wesley Professional, 2005.

[22] Hua-Fu Li, Man-Kwan Shan, and Suh-Yin Lee. Online mining of frequent query trees over XML data streams. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 959–960, 2006.

[23] Tyng-Luh Liu and Davi Geiger. Approximate tree matching and shape similarity. In *ICCV*, pages 456–462, 1999.

[24] Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[25] Dennis Shasha, Jason T. L. Wang, and Sen Zhang. Unordered tree mining with applications to phylogeny. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 708, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Guojie Song, Dongqing Yang, Bin Cui, Baihua Zheng, Yunfeng Liu, and Kunqing Xie. CLAIM: An efficient method for relaxed frequent closed itemsets mining over stream data. In *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, volume 4443 of *Lecture Notes in Computer Science*, pages 664–675. Springer, 2007.

[27] Alexandre Termier, Marie-Christine Rousset, and Michele Sebag. DRYADE: a new approach for discovering closed frequent trees in heterogeneous tree databases. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*, pages 543–546. IEEE Computer Society, 2004.

[28] Gabriel Valiente. *Algorithms on Trees and Graphs.* Springer-Verlag, Berlin, 2002.

[29] Sholom Weiss, Nitin Indurkhya, Tong Zhang, and Fred Damerau. *Text Mining: Predictive Methods for Analyzing Unstructured Information.* SpringerVerlag, 2004.

[30] Xifeng Yan and Jiawei Han. CloseGraph: mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 286–295, New York, NY, USA, 2003. ACM Press.

[31] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 71–80. ACM, 2002.

[32] Mohammed Javeed Zaki, Nagender Parimi, Nilanjana De, Feng Gao, Benjarath Phoophakdee, Joe Urban, Vineet Chaoji, Mohammad Al Hasan, and Saeed Salem. Towards generic pattern mining. In *Formal Concept Analysis, Third International Conference, ICFCA 2005, Lens, France, February 14-18, 2005, Proceedings*, volume 3403 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2005.