Mach Learn (2010) 78: 1–33
DOI 10.1007/s10994-009-5123-9

Mining frequent closed rooted trees

José L. Balcázar · Albert Bifet · Antoni Lozano

Received: 17 March 2007 / Revised: 29 May 2009 / Accepted: 3 June 2009 / Published online: 23 July 2009

Abstract Many knowledge representation mechanisms are based on tree-like structures, thus symbolizing the fact that certain pieces of information are related in one sense or another. There exists a well-studied process of closure-based data mining in the itemset framework: we consider the extension of this process into trees. We focus mostly on the case where labels on the nodes are nonexistent or unreliable, and discuss algorithms for closure-based mining that only rely on the root of the tree and the link structure. We provide a notion of intersection that leads to a deeper understanding of the notion of support-based closure, in terms of an actual closure operator. We describe combinatorial characterizations and some properties of ordered trees, discuss their applicability to unordered trees, and rely on them to design efficient algorithms for mining frequent closed subtrees both in the ordered and the unordered settings. Empirical validations and comparisons with alternative algorithms are provided.

Keywords Tree mining · Closure operator · Unordered trees · Labeled trees

1 Introduction

Undisputably tree-structured representations are a key idea pervading all of Computer Science; many link-based structures may be studied formally by means of trees. From the B+ indices that make our commercial Database Management Systems useful, through search-tree or heap data structures or Tree Automata, up to the decision tree structures in Artificial Intelligence and Decision Theory, or the parsing structures in Compiler Design,

Editor: Thomas Gärtner.

J.L. Balcázar · A. Bifet (✉) · A. Lozano
Department de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, C. Jordi Girona, 1-3, 08034 Barcelona, Spain
e-mail: abifet@lsi.upc.edu

J.L. Balcázar
e-mail: balqui@lsi.upc.edu

A. Lozano
e-mail: antoni@lsi.upc.edu

in Natural Language Processing, or in the now-ubiquitous XML, trees often represent an optimal compromise between the conceptual simplicity and processing efficiency of strings and the harder but much richer knowledge representations based on graphs. Mining frequent trees is becoming an important task, with broad applications including chemical informatics (Hashimoto et al. 2008), computer vision (Liu and Geiger 1999), text retrieval (Weiss et al. 2004), bioinformatics (Shasha et al. 2004; Hein et al. 1995), and Web analysis (Chakrabarti 2002; Zaki 2002). A wealth of variations of the basic notions, both of the structures themselves (binary, bounded-rank, unranked, ordered, unordered) or of their relationships (like induced or embedded top-down or bottom-up subtree relations) have been proposed for study and motivated applications.

Closure-based mining on purely relational data, that is, itemset mining, is, by now, well-established, and there are interesting algorithmic developments. Sharing some of the attractive features of frequency-based summarization of subsets, it offers an alternative view with several advantages; among them, there are the facts that, first, by imposing closure, the number of frequent sets is heavily reduced, and, second, the possibility appears of developing a mathematical foundation that connects closure-based mining with lattice-theoretic approaches like 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 sets, which are, therefore, preferred often by non-experts. Thus, there have been subsequent efforts in moving towards closure-based mining on structured data. We provide now some definitions and, then, a discussion of existing work.

1.1 Preliminary definitions

Our *trees* will be rooted, unranked trees (that is, with nodes of unbounded arity), and we will consider two kinds of trees: *ordered trees*, in which the children of any node form a sequence of siblings, and *unordered trees*, in which they form a set of siblings. Note that this difference is not intrinsic, but, rather, lies in the way we look at the trees (more precisely, in the specifics of the implementation of some abstract data type primitives such as deciding subtree relations—see below). The set of all trees will be denoted with $\mathcal{T}$. 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. In the unordered case, the components form a set, not a sequence; therefore, permuting them does not give a different tree. In our drawings of unordered trees, we follow the convention that deeper, larger trees are drawn at the left of smaller trees.

A *bottom-up subtree* of a tree t is any connected subgraph rooted at some node v of t which contains exactly the descendants of v in t. The *level* or *level* of a node is the length of the path from the root to that node (the root has level 0). A bottom-up subtree of a tree t is at level d if its root is at level d in t.

An *induced subtree* of a tree t is any connected subgraph rooted at some node v of t such that its vertices and edges are subsets of those of t. An *embedded subtree* of a tree t is any connected subgraph rooted at some node v of t that does not break the ancestor-descendant relationship among the vertices of t. Formally, let s 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 s is an *induced subtree* of t if and only if (1) $V' \subseteq V$, (2) $E' \subseteq E$, and (3) the labeling of V' is preserved in t. Tree s is an *embedded subtree* of t if and only if (1) $V' \subseteq V$, (2) $(v_1, v_2) \in E'$ (here, v_1 is the parent of v_2 in s) if and only if v_1 is an ancestor of v_2 in t, and (3) the labeling of V' is preserved in t.

In order to compare link-based structures, we will also be interested in a notion of subtree where the root is preserved. In the unordered case, a tree t' is a *top-down subtree* (or simply

a *subtree*) of a tree t (written $t' \preceq t$) if t' is a connected subgraph of t which contains the root of t. Note that the ordering of the children is not relevant. In the ordered case, the order of the existing children of each node must be additionally preserved. All along this paper, the main place where it is relevant whether we are using ordered or unordered trees is what is the choice of the implementation of the test for the subtree notion.

Given a finite dataset $\mathcal{D}$ of transactions, where each transaction $s \in \mathcal{D}$ is an unlabeled rooted tree, we say that a transaction s *supports* a tree t if the tree t is a subtree of the transaction s. The number of transactions in the dataset $\mathcal{D}$ that support t is called the *support* of the tree t. A tree t is called *frequent* if its support is greater than or equal to a given threshold *min_sup*. The frequent tree mining problem is to find all frequent trees in a given dataset. Any subtree of a frequent tree is also frequent and, therefore, any supertree of a nonfrequent tree is also nonfrequent.

We define a frequent tree t to be *closed* if none of its proper supertrees has the same support as it has. Generally, there are much fewer closed trees than frequent ones. In fact, we can obtain all frequent subtrees with their support from the set of closed frequent subtrees with their supports, as explained later on: whereas this is immediate for itemsets, in the case of trees we will need to employ some care because a frequent tree may be a subtree of several incomparable frequent closed trees.

1.2 Related work

There exist already work about closure-based mining on structured data, particularly sequences (Yan et al. 2003; Balcázar and Garriga 2005), trees (Chi et al. 2001b; Termier et al. 2004) and graphs (Yan and Han 2003; Yan et al. 2005). One of the differences with closed itemset mining stems from the fact that the set theoretic intersection no longer applies, and whereas the intersection of sets is a set, the intersection of two sequences or two trees is not one sequence or one tree. 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 closedness of a tree or sequence (Chi et al. 2001b, see below); others (like Arimura 2005) 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, Termier et al. 2004 or nonrepeated labels at all in sequences, Garriga and Balcázar 2004).

Yan and Han (2002, 2003) proposed two algorithms for mining frequent and closed graphs. The first one is called gSpan (graph-based Substructure pattern mining) and discovers frequent graph substructures without candidate generation; gSpan builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan adopts the depth-first search strategy to mine frequent connected subgraphs. The second one is called CloseGraph and discovers closed graph patterns. CloseGraph is based on gSpan, and is based on the development of two pruning methods: equivalent occurrence and early termination. The early termination method is similar to the early termination by equivalence of projected databases method in CloSpan (Yan et al. 2003), an algorithm for mining closed sequential patterns in large datasets published by the Illimine team. However, in graphs there are some cases where early termination may fail and miss some patterns. By detecting and eliminating these cases, CloseGraph guarantees the completeness and soundness of the closed graph patterns discovered.

1.2.1 Frequent tree mining algorithms

In the case of trees, only labeled tree mining methods are considered in the literature. There are four broad kinds of subtrees: bottom-up subtrees, top-down subtrees, induced subtrees, and embedded subtrees. Bottom-up subtree mining is the simplest subtree mining. Algorithms for embedded labeled frequent trees are the following:

- Rooted Ordered Trees
  - **TreeMiner** (Zaki 2002): This algorithm developed by Zaki, uses vertical representations for support counting, and follows the combined depth-first/breadth traversal idea to discover all embedded ordered subtrees.
- Rooted Unordered Trees
  - **SLEUTH** (Zaki 2005): This method presented by Zaki, extends TreeMiner to the unordered case using two different methods for generating canonical candidates: the class-based extension and the canonical extension.

Algorithms for induced labeled frequent trees include:

- Rooted Ordered Trees
  - **FREQT** (Asai et al. 2002). Asai et al. developed FREQT. It uses an extension approach based on the rightmost path. FREQT uses an occurrence list base approach to determine the support of trees.
- Rooted Unordered Trees
  - **uFreqt** (Nijssen and Kok 2003): Nijssen et al. extended FREQT to the unordered case. Their method solves in the worst case, a maximum bipartite matching problem when counting tree supports.
  - **uNot** (Asai et al. 2003): Asai et al. presented uNot in order to extend FREQT. It uses an occurrence list based approach which is similar to Zaki's TreeMiner.
  - **HybridTreeMiner** (Chi et al. 2004): Chi et al. proposed HybridTreeMiner, a method that generates candidates using both joins and extensions. It uses the combined depth-first/breadth-first traversal approach.
  - **PathJoin** (Xiao et al. 2003): Xiao et al. developed PathJoin, assuming that no two siblings are identically labeled. It presents the *maximal* frequent subtrees. A *maximal* frequent subtree is a frequent subtree none of whose proper supertrees are frequent.

All the labeled frequent tree mining methods proposed in the literature are occurrence based and solve these two problems:

- the computation of a tree inclusion relation
- the enumeration of all trees in a non-redundant way

A comprehensive introduction to the algorithms on unlabeled trees can be found in (Valiente 2002) and a survey of works on frequent subtree mining can be found in (Chi et al. 2001a).

1.2.2 Closed tree mining algorithms

Our main interest is related to closed trees since they, if appropriately organized as shown below, give the same information as the set of all frequent trees in less space.

Chi et al. proposed CMTreeMiner (Chi et al. 2001b), the first algorithm to discover all closed and maximal frequent labeled induced subtrees without first discovering all frequent subtrees. CMTreeMiner shares many features with CloseGraph, and uses two pruning techniques: the *left-blanket* and *right-blanket* pruning. The *blanket* of a tree is defined as the set

of immediate supertrees that are frequent, where an *immediate supertree* of a tree t is a tree that has one more vertex than t. The *left-blanket* of a tree t is the blanket where the vertex added is not in the right-most path of t (the path from the root to the rightmost vertex of t). The *right-blanket* of a tree t is the blanket where the vertex added is in the right-most path of t. Their method is as follows: it computes, for each candidate tree, the set of trees that are occurrence-matched with its blanket's trees. If this set is not empty, they apply two pruning techniques using the left-blanket and right-blanket. If it is empty, then they check if the set of trees that are transaction-matched but not occurrence matched with its blanket's trees is also empty. If this is the case, there is no supertree with the same support and then the tree is closed. CMTreeMiner is a labeled tree method and it was not designed for unlabeled trees. As the authors of CMTreeMiner say in their paper (Chi et al. 2001b): *"Therefore, if the number of distinct labels decrease dramatically (so different occurrences for the same pattern increase dramatically), the memory usage of CMTreeMiner is expected to increase and its performance is expected to deteriorate."*

Arimura and Uno proposed CLOATT (Arimura 2005) considering closed mining in attribute trees, which is a subclass of labeled ordered trees and can also be regarded as a fragment of description logic with functional roles only. These attribute trees are defined using a relaxed tree inclusion.

Termier et al. proposed DRYADEPARENT (Termier et al. 2008) as a closed frequent attribute tree mining method comparable to CMTreeMiner. Attribute trees are trees such that two sibling nodes cannot have the same label. They extend to induced subtrees their previous algorithm DRYADE (Termier et al. 2004).

The DRYADE and DRYADEPARENT algorithm are based on the computation of tiles (closed frequent attribute trees of depth 1) in the data and on an efficient hooking strategy that reconstructs the closed frequent trees from these tiles. Whereas CMTreeMiner uses a classical generate-and-test strategy to build candidate trees edge by edge, the hooking strategy of DRYADEPARENT finds a complete depth level at each iteration and does not need tree mapping tests. The authors claim that their experiments have shown that DRYADEPARENT is faster than CMTreeMiner in most settings and that the performances of DRYADEPARENT are robust with respect to the structure of the closed frequent trees to find, whereas the performances of CMTreeMiner are biased toward trees having most of their edges on their rightmost branch.

As attribute trees are trees such that two sibling nodes cannot have the same label, DRYADEPARENT is not a method appropriate for dealing with unlabeled trees.

1.3 Contributions of this paper

Our focus in this paper is on unlabeled rooted trees and top-down subtrees, although we will discuss briefly the labeled and induced case too. Thus our relevant information is the root and the link structure. Our motivation arose from the analysis of web navigation patterns, where we only looked at the sets of pages visited in each single session, structured in a tree-like form and desiring to use, on purpose, no information beyond the links, as a way of exploring the potential limitations of this source of information; this study was to be combined and complemented with a development of a novel distributed, focused crawler that would rely on the closures found among the navigation patterns to approximate the local users' interests. Unfortunately this complementary part of the project is currently postponed, but the closure-based analysis of trees led already to the developments described here. We start discussing the properties of the intersection operator as a foundation to a closure operator in Sect. 3, along the lines of (Ganter and Wille 1999), (Balcázar and Garriga 2005), (Garriga 2006), or

(Baixeries and Balcázar 2003) for unstructured or otherwise structured datasets; we study algorithms to compute intersections in Sect. 2. Preliminary versions of these results were announced at (Balcázar et al. 2006). A representation of ordered trees is studied in Sect. 4, including an efficient algorithm to test the subtree relation, which is subsequently used to design algorithms for mining closed ordered trees in Sect. 5. Section 6 extends the analysis to unordered trees and Sect. 7 to induced subtrees and labeled trees. Section 8 discusses an experimental comparison and other potential applications. Part of the results of Sects. 5, 6, and 8 appear in preliminary, quite incomplete form in (Balcázar et al. 2007), although early, similar but weaker results appear also in (Balcázar et al. 2007).

2 Basic algorithmics and mathematical properties

This section discusses, mainly, to what extent the intuitions about trees can be formalized in mathematical and algorithmic terms. As such, it is aimed just at building up intuition and background understanding, and making sure that our later sections on tree mining algorithms rest on solid foundations: they connect with these properties but make little explicit use of them.

Given two trees, a common subtree is a tree that is subtree of both; it is a maximal common subtree if it is not a subtree of any other common subtree. Two trees have always some maximal common subtree but, as is shown in Fig. 1, this common subtree does not need to be unique. This figure also serves the purpose of further illustrating the notion of unordered subtree.

In fact, both trees X and Y in Fig. 1 have the maximum number of nodes among the common subtrees of A and B.

From here on, the *intersection* of a set of trees is the set of all maximal common subtrees of the trees in the set. Sometimes, the one-node tree will be represented with the symbol •, and the two-node tree by •–•.

2.1 Facts from combinatorics on trees

The number of trees with n nodes is known to be $\Theta(\rho^n n^{-3/2})$, where $\rho = 0.3383218569$ (Plotkin and Rosenthal 1994). We provide a more modest lower bound based on an easy way to count the number of unordered binary trees; this will be enough to show in a few lines an exponential lower bound on the number of trees with n nodes.

Define B_n as the number of unordered binary trees with n nodes, and set $B_0 = 1$ for convenience. Clearly, a root without children (tree •) is the only binary tree with one node,

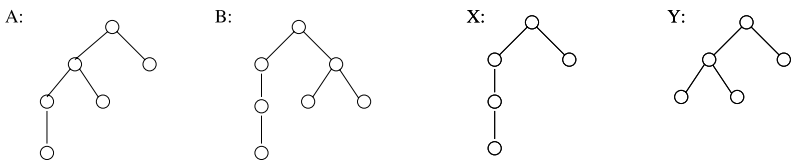
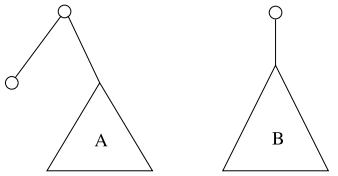


Fig. 1 Trees X and Y are maximal common subtrees of A and B

so $B_1 = 1$, while a root with just one child which is a leaf (tree •—•) is the only binary tree with two nodes, so $B_2 = 1$. Now note that each of the trees



has n nodes if A is a subtree with $n-2$ nodes and B is a subtree with $n-1$ nodes. Moreover, since these two kinds of trees form disjoint subclasses of the trees with n nodes, it holds that $B_n \geq B_{n-1} + B_{n-2}$ for all $n \geq 3$, thus showing that B_n is bigger than the n-th Fibonacci number F_n (note that the initial values also satisfy the inequality, since $F_0 = 0$ and $F_1 = F_2 = 1$). Since it is well-known that $F_{n+2} \geq \phi^n$, where $\phi > 1.618$ is the golden number, we have the lower bound

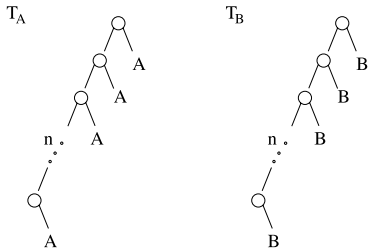
$$\phi^{n-2} \leq F_n \leq B_n$$

which is also a lower bound for the total number of trees (both ordered and unordered) with n nodes.

2.2 Number of subtrees

We can easily observe, using the trees A, B, X, and Y above, that two trees can have an exponential number of maximal common subtrees.

Recall that the aforementioned trees have the property that X and Y are two maximal common subtrees of A and B. Now, consider the pair of trees constructed in the following way using copies of A and B. First, take a path of length $n-1$ (thus having n nodes which include the root and the unique leaf) and "attach" to each node a whole copy of A. Call this tree T_A. Then, do the same with a fresh path of the same length, with copies of B hanging from their nodes, and call this tree T_B. Graphically:



All the trees constructed similarly with copies of X or Y attached to each node of the main path (instead of A or B) are maximal common subtrees of T_A and T_B. The fact that the copies are at different depths assures that all the 2^n possibilities correspond to different subtrees. Therefore, the number of different maximal common subtrees of T_A and T_B is at least 2^n (which is exponential in the input since the sum of the sizes of T_A and T_B is $15n$). Any algorithm for computing maximal common subtrees has, therefore, a worst case exponential cost due to the size of the output.

We must note, though, that experiments suggest that intersection sets of cardinality beyond 1 hardly ever arise unless looked for. In order to find how often two trees have intersection sets of cardinality beyond 1, we set up an empirical validation using the tree generation program of Zaki (2002) to generate a random set of trees. 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 all the probabilities is 1. Using the master tree, the dataset is generated selecting subtrees according to these probabilities.

Using Zaki's tree generator program we generate sets of 100 random trees of sizes from 5 to 50 and then we run our frequent tree mining algorithm with minimum support 2. Our program doesn't find any two trees with the same transactions list in any run of the algorithm. This fact suggests that, as all the intersections came up to a single tree, the exponential blow-up of the intersection sets is extremely infrequent.

2.3 Finding the intersection of trees recursively

Computing a potentially large intersection of a set of trees is not a trivial task, given that there is no ordering among the components: a maximal element of the intersection may arise through mapping smaller components of one of the trees into larger ones of the other. Therefore, the degree of branching along the exploration is high. We propose a natural recursive algorithm to compute intersections.

The basic idea is to exploit the recursive structure of the problem by considering all the ways to match the components of the two input trees. Suppose we are given the trees t and r, whose components are $t_1, \ldots, t_k$ and $r_1, \ldots, r_n$, respectively. If $k \leq n$, then clearly $(t_1, r_1), \ldots, (t_k, r_k)$ is one of those matchings. Then, we recursively compute the maximal common subtrees of each pair (t_i, r_i) and "cross" them with the subtrees of the previously computed pairs, thus giving a set of maximal common subtrees of t and r for this particular identity matching. The algorithm explores all the (exponentially many) matchings and, finally, eliminates repetitions and trees which are not maximal (by using recursion again).

We do not specify the data structure used to encode the trees. The only condition needed is that every component t' of a tree t can be accessed with an index which indicates the lexicographical position of its encoding $\langle t' \rangle$ with respect to the encodings of the other components; this will be $\textsc{Component}(t, i)$. The other procedures are as follows:

- $\#\textsc{Components}(t)$ computes the number of components of t, that is, the arity of the root of t.
- $\textsc{Matchings}(n_1, n_2)$ computes the set of perfect matchings of the graph K_{n_1,n_2}, that is, of the complete bipartite graph with partition classes $\{1, \ldots, n_1\}$ and $\{1, \ldots, n_2\}$ (each class represents the components of one of the trees). For example, $\textsc{Matchings}(2, 3) = \{\{(1, 1), (2, 2)\}, \{(1, 1), (2, 3)\}, \{(1, 2), (2, 1)\}, \{(1, 2), (2, 3)\}, \{(1, 3), (2, 1)\}, \{(1, 3), (2, 2)\}\}$.
- $\textsc{Cross}(l_1, l_2)$ returns a list of trees constructed in the following way: for each tree t_1 in l_1 and for each tree t_2 in l_2 make a copy of t_1 and add t_2 to it as a new component.
- $\textsc{Max Subtrees}(S_1, S_2)$ returns the list of trees containing every tree in S_1 that is not a subtree of another tree in S_2 and every tree in S_2 that is not a subtree of another tree in S_1, thus leaving only the maximal subtrees. This procedure is shown in Fig. 3. There is a further analysis of it in the next subsection.

The fact that, as has been shown, two trees may have an exponential number of maximal common subtrees necessarily makes any algorithm for computing all maximal subtrees inefficient. However, there is still space for some improvement.

```
RECURSIVE INTERSECTION(r, t)
    Input: A tree r, a tree t.
    Output: A set of trees, intersection of r and t.

 1  if (r = •) or (t = •)
 2     then S ← {•}
 3  elseif (r = •—•) or (t = •—•)
 4     then S ← {•—•}
 5     else  S ← {}
 6           n_r ← #COMPONENTS(r)
 7           n_t ← #COMPONENTS(t)
 8           for each m in MATCHINGS(n_r, n_t)
 9                do mTrees ← {•}
10                   for each (i, j) in m
11                        do c_r ← COMPONENT(r, i)
12                           c_t ← COMPONENT(t, j)
13                           cTrees ← RECURSIVE INTERSECTION(c_r, c_t)
14                           mTrees ← CROSS(mTrees, cTrees)
15                   S ← MAX SUBTREES(S, mTrees)
16  return S
```

Fig. 2 Algorithm RECURSIVE INTERSECTION

Fig. 3 Algorithm MAX SUBTREES

```
MAX SUBTREES(S_1, S_2)
1  for each r in S_1
2        do for each t in S_2
3             if r is a subtree of t
4                then mark r
5             elseif t is a subtree of r
6                then mark t
7  return sublist of nonmarked trees in S_1 ∪ S_2
```

2.4 Finding the intersection by dynamic programming

In the above algorithm, recursion can be replaced by a table of precomputed answers for the components of the input trees. This way we avoid repeated recursive calls for the same trees, and speed up the computation. Suppose we are given two trees r and t. In the first place, we compute all the trees that can appear in the recursive queries of RECURSIVE INTERSECTION(r, t). This is done in the following procedure:

– SUBCOMPONENTS(t) returns a list containing t if $t = \bullet$; otherwise, if t has the components $t_1, \ldots, t_k$, then, it returns a list containing t and the trees in SUBCOMPONENTS(t_i) for every t_i, ordered increasingly by number of nodes.

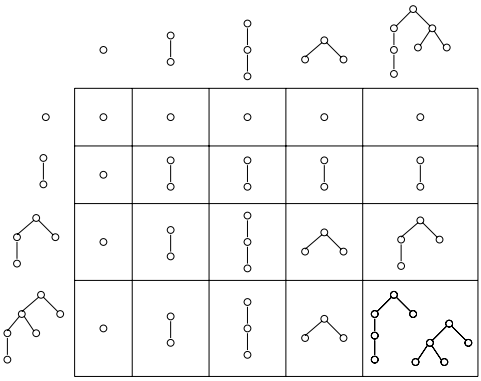
The new algorithm shown in Fig. 4 constructs a dictionary D accessed by pairs of trees (t_1, t_2) when the input trees are nontrivial (different from $\bullet$ and $\bullet\!-\!\bullet$, which are treated separately). Inside the main loops, the trees which are used as keys for accessing the dictionary are taken from the lists SUBCOMPONENTS(r) and SUBCOMPONENTS(t), where r and t are the input trees.

Note that the fact that the number of trees in SUBCOMPONENTS(t) is linear in the number of nodes of t assures a quadratic size for D. The entries of the dictionary are computed by increasing order of the number of nodes; this way, the information needed to compute an entry has already been computed in previous steps.

```
DYNAMIC PROGRAMMING INTERSECTION(r, t)
 1  for each s_r in SUBCOMPONENTS(r)
 2       do for each s_t in SUBCOMPONENTS(t)
 3            do if (s_r = •) or (s_t = •)
 4                 then D[s_r, s_t] ← {•}
 5               elseif (s_r = •—•) or (s_t = •—•)
 6                 then D[s_r, s_t] ← {•—•}
 7                 else D[s_r, s_t] ← {}
 8                      ns_r ← #COMPONENTS(s_r)
 9                      ns_t ← #COMPONENTS(s_t)
10                      for each m in MATCHINGS(ns_r, ns_t)
11                           do mTrees ← {•}
12                              for each (i, j) in m
13                                   do cs_r ← COMPONENT(s_r, i)
14                                      cs_t ← COMPONENT(s_t, j)
15                                      cTrees ← D[cs_r, cs_t]
16                                      mTrees ← CROSS(mTrees, cTrees)
17                              D[s_r, s_t] ← MAX SUBTREES(D[s_r, s_t], mTrees)
18  return D[r, t]
```

Fig. 4 Algorithm DYNAMIC PROGRAMMING INTERSECTION

Fig. 5 Table with all partial results computed



The procedure MAX SUBTREES, which appears in the penultimate step of the two intersection algorithms presented, was presented in Sect. 2.3. The key point in the procedure MAX SUBTREES is the identification of subtrees made in steps 3 and 5 of Fig. 3. This is discussed in depth below, but let us advance that, in the unordered case, it can be decided whether $t_1 \preceq t_2$ in time $O(n_1 n_2^{1.5})$ (Valiente 2002), where n_1 and n_2 are the number of nodes of t_1 and t_2, respectively.

Finally, the table in Fig. 5 shows an example of the intersections stored in the dictionary by the algorithm DYNAMIC PROGRAMMING INTERSECTION with trees A and B of Fig. 1 as input.

3 Closure operator on trees

Now we attempt at formalizing a closure operator for substantiating the work on closed trees, with no resort to the labelings: we focus on the case where the given dataset consists of unlabeled, rooted trees; thus, our only relevant information is the identity of the root

and the link structure. In order to have the same advantages as with frequent closed itemset mining, we want to be able to obtain all frequent subtrees, with their support, from the set of closed frequent subtrees with their supports. We propose a notion of Galois connection with the associated closure operator, in such a way that we can characterize support-based notions of closure with a mathematical operator.

For a notion of closed (sets of) trees to make sense, we expect to be given as data a finite set (actually, a list) of transactions, each of which consisting of its transaction identifier (tid) and a tree. Transaction identifiers are assumed to run sequentially from 1 to N, the size of the dataset. We denote $\mathcal{D} \subset \mathcal{T}$ the dataset. General usage would lead to the following notion of closed tree:

Definition 1 A tree t is closed for $\mathcal{D}$ if no tree $t' \neq t$ exists with the same support such that $t \preceq t'$.

We aim at clarifying the properties of closed trees, providing a more detailed justification of the term "closed" through a closure operator obtained from a Galois connection, along the lines of (Ganter and Wille 1999), (Balcázar and Garriga 2005), (Garriga 2006), or (Baixeries and Balcázar 2003) for unstructured or otherwise structured datasets. However, given that the intersection of a set of trees is not a single tree but yet another set of trees, we will find that the notion of "closed" is to be applied to subsets of the transaction list, and that the notion of a "closed tree" t is not exactly coincident with the singleton $\{t\}$ being closed.

To see that the task is not fully trivial, note first that $t \preceq t'$ implies that t is a subtree of all the transactions where t' is a subtree, so that the support of t is, at least, that of t'. Existence of a larger t' with the same support would mean that t does not gather all the possible information about the transactions in which it appears, since t' also appears in the same transactions and gives more information (is more specific). A closed tree is maximally specific for the transactions in which it appears. However, note that the example of the trees A and B given above provides two trees X and Y with the same support, and yet mutually incomparable. This is, in a sense, a problem. Indeed, for itemsets, and several other structures, the closure operator "maximizes the available information" by a process that would correspond to the following: given tree t, find the largest supertree of t which appears in all the transactions where t appears. But doing it that way, in the case of trees, does not maximize the information: there can be different, incomparable trees supported by the same set of transactions. Maximizing the information requires us to find them all.

There is a way forward, that can be casted into two alternative forms, equally simple and essentially equivalent. We can consider each subtree of some tree in the input dataset as an atomic item, and translate each transaction into an itemset on these items (all subtrees of the transaction tree). Then we can apply the standard Galois connection for itemsets, where closed sets would be sets of items, that is, sets of trees. The alternative we describe can be seen also as an implementation of this idea, where the difference is almost cosmetic, and we mention below yet another simple variant that we have chosen for our implementations, and that is easier to describe starting from the tree-based form we give now.

3.1 Galois connection

A Galois connection is provided by two functions, relating two partial orders in a certain way. Here our partial orders are plain power sets of the transactions, on the one hand, and of the corresponding subtrees, in the other. On the basis of the binary relation $t \preceq t'$, the following definition and proposition are rather standard.

Definition 2 The Galois connection pair:

- For finite $A \subseteq \mathcal{D}$, $\sigma(A) = \{t \in \mathcal{T} \mid \forall t' \in A\,(t \preceq t')\}$.
- For finite $B \subset \mathcal{T}$, not necessarily in $\mathcal{D}$, $\tau_{\mathcal{D}}(B) = \{t' \in \mathcal{D} \mid \forall t \in B\,(t \preceq t')\}$.

The use of finite parts of the infinite set $\mathcal{T}$ should not obscure the fact that the image of the second function is empty except for finitely many sets B; in fact, we could use, instead of $\mathcal{T}$, the set of all trees that are subtrees of some tree in $\mathcal{D}$, with exactly the same effect overall. There are many ways to argue that such a pair is a Galois connection. One of the most useful ones is as follows.

Proposition 1 *For all finite $A \subseteq \mathcal{D}$ and $B \subset \mathcal{T}$, the following holds*:

$$A \subseteq \tau_{\mathcal{D}}(B) \iff B \subseteq \sigma(A).$$

This fact follows immediately since, by definition, each of the two sides is equivalent to $\forall t \in B\,\forall t' \in A\,(t \preceq t')$.

It is well-known that the compositions (in either order) of the two functions that define a Galois connection constitute closure operators, that is, are monotonic, extensive, and idempotent (with respect, in our case, to set inclusion).

Corollary 1 *The composition $\tau_{\mathcal{D}} \circ \sigma$ is a closure operator on the subsets of $\mathcal{D}$. The converse composition $\Gamma_{\mathcal{D}} = \sigma \circ \tau_{\mathcal{D}}$ is also a closure operator.*

$\Gamma_{\mathcal{D}}$ operates on subsets of $\mathcal{T}$; more precisely, again, on subsets of the set of all trees that appear as subtrees somewhere in $\mathcal{D}$. Thus, we have now both a concept of "closed set of transactions" of $\mathcal{D}$, and a concept of "closed sets of trees", and they are in bijective correspondence through both sides of the Galois connection. However, the notion of closure based on support, as previously defined, corresponds to single trees, and it is worth clarifying the connection between them, naturally considering the closure of the singleton set containing a given tree, $\Gamma_{\mathcal{D}}(\{t\})$, assumed nonempty, that is, assuming that t indeed appears as subtree somewhere along the dataset. We point out the following easy-to-check properties:

1. $t \in \Gamma_{\mathcal{D}}(\{t\})$.
2. $t' \in \Gamma_{\mathcal{D}}(\{t\})$ if and only if $\forall s \in \mathcal{D}(t \preceq s \Rightarrow t' \preceq s)$.
3. t may be, or may not be, maximal in $\Gamma_{\mathcal{D}}(\{t\})$ (maximality is formalized as: $\forall t' \in \Gamma_{\mathcal{D}}(\{t\})[t \preceq t' \Rightarrow t = t']$). In fact, t is maximal in $\Gamma_{\mathcal{D}}(\{t\})$ if and only if $\forall t'(\forall s \in \mathcal{D}[t \preceq s \Rightarrow t' \preceq s] \wedge t \preceq t' \Rightarrow t = t')$.

The definition of closed tree can be phrased in a similar manner as follows: t is closed for $\mathcal{D}$ if and only if: $\forall t'(t \preceq t' \wedge \mathrm{supp}(t) = \mathrm{supp}(t') \Rightarrow t = t')$.

Theorem 1 *A tree t is closed for $\mathcal{D}$ if and only if it is maximal in $\Gamma_{\mathcal{D}}(\{t\})$.*

Proof Suppose t is maximal in $\Gamma_{\mathcal{D}}(\{t\})$, and let $t \preceq t'$ with $\mathrm{supp}(t) = \mathrm{supp}(t')$. The data trees s that count for the support of t' must count as well for the support of t, because $t' \preceq s$ implies $t \preceq t' \preceq s$. The equality of the supports then implies that they are the same set, that is, $\forall s \in \mathcal{D}(t \preceq s \iff t' \preceq s)$, and then, by the third property above, maximality implies $t = t'$. Thus t is closed.

Conversely, suppose t is closed and let $t' \in \Gamma_{\mathcal{D}}(\{t\})$ with $t \preceq t'$. Again, then $\mathrm{supp}(t') \leq \mathrm{supp}(t)$; but, from $t' \in \Gamma_{\mathcal{D}}(\{t\})$ we have, as in the second property above, $(t \preceq s \Rightarrow t' \preceq s)$

for all $s \in \mathcal{D}$, that is, $\text{supp}(t) \leq \text{supp}(t')$. Hence, equality holds, and from the fact that t is closed, with $t \preceq t'$ and $\text{supp}(t) = \text{supp}(t')$, we infer $t = t'$. Thus, t is maximal in $\Gamma_{\mathcal{D}}(\{t\})$. □

Now we can continue the argument as follows. Suppose t is maximal in some closed set B of trees. From $t \in B$, by monotonicity and idempotency, together with aforementioned properties, we obtain $t \in \Gamma_{\mathcal{D}}(\{t\}) \subseteq \Gamma_{\mathcal{D}}(B) = B$; being maximal in the larger set implies being maximal in the smaller one, so that t is maximal in $\Gamma_{\mathcal{D}}(\{t\})$ as well. Hence, we have argued the following alternative, somewhat simpler, characterization:

Corollary 2 *A tree is closed for $\mathcal{D}$ if and only if it is maximal in some closed set of $\Gamma_{\mathcal{D}}$.*

A simple observation here is that each closed set is uniquely defined through its maximal elements. In fact, our implementations chose to avoid duplicate calculations and redundant information by just storing the maximal trees of each closed set. We could have defined the Galois connection so that it would provide us "irredundant" sets of trees by keeping only maximal ones; the property of maximality would be then simplified into $t \in \Gamma_{\mathcal{D}}(\{t\})$, which would not be guaranteed anymore (cf. the notion of stable sequences in Balcázar and Garriga 2005). The formal details of the validation of the Galois connection property would differ slightly (in particular, the ordering would not be simply a mere subset relationship) but the essentials would be identical, so that we refrain from developing that approach here. We would obtain a development somewhat closer to (Balcázar and Garriga 2005) than our current development is, but there would be no indisputable advantages.

Now, given any set t, its support is the same as that of $\Gamma_{\mathcal{D}}(\{t\})$; knowing the closed sets of trees and their supports gives us all the supports of all the subtrees. As indicated, this includes all the closed trees, but has more information regarding their joint membership in closed sets of trees. We can compute the support of arbitrary frequent trees in the following manner, that has been suggested to us by an anonymous reviewer of this paper: assume that we have the supports of all closed frequent trees, and that we are given a tree t; if it is frequent and closed, we know its support, otherwise we find the smallest closed frequent supertrees of t. Here we depart from the itemset case, because there is no unicity: there may be several noncomparable minimal frequent closed supertrees, but the support of t is the largest support appearing among these supertrees, due to the antimonotonicity of support.

For further illustration, we exhibit here, additionally, a toy example of the closure lattice for a simple dataset consisting of six trees, thus providing additional hints on our notion of intersection; these trees were not made up for the example, but were instead obtained through six different (rather arbitrary) random seeds of the synthetic tree generator of Zaki (2002).

Figure 6 depicts the closed sets obtained. It is interesting to note that all the intersections came up to a single tree, a fact that suggests that the exponential blow-up of the intersection sets, which is possible as explained in Sect. 2.2, appears infrequently enough.

Of course, the common intersection of the whole dataset is (at least) a "pole" whose length is the minimal height of the data trees.

4 Level representations

The development so far is independent of the way in which the trees are represented. The reduction of a tree representation to a (frequently augmented) sequential representation has always been a source of ideas, already discussed in depth in Knuth (1997, 2005). We use

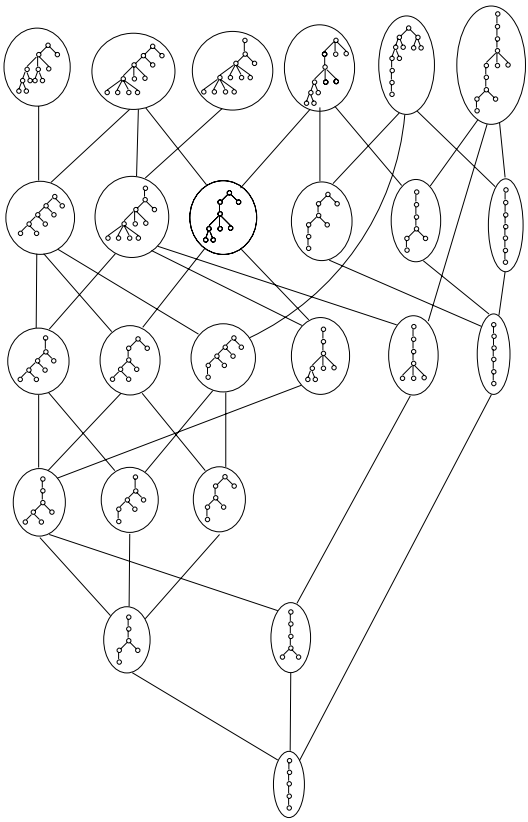


Fig. 6 Lattice of closed trees for the six input trees in the *top row*

here a specific data structure (Nakano and Uno 2003; Beyer and Mitchell Hedetniemi 1980; Asai et al. 2003; Nijssen and Kok 2003) to implement trees that leads to a particularly streamlined implementation of the closure-based mining algorithms.

We will represent each tree as a sequence over a countably infinite alphabet, namely, the set of natural numbers; we will concentrate on a specific language, whose strings exhibit a very constrained growth pattern. Some simple operations on strings of natural numbers are:

Definition 3 Given two sequences of natural numbers x, y, we represent by

- $|x|$ the length of x.
- $x \cdot y$ the sequence obtained as concatenation of x and y.
- $x + i$ the sequence obtained adding i to each component of x; we represent by x^+ the sequence $x + 1$.

We will apply to our sequences the common terminology for strings: the term *subsequence* will be used in the same sense as *substring*; in the same way, we will also refer to *prefixes* and *suffixes*. Also, we can apply lexicographical comparisons to our sequences.

The language we are interested in is formed by sequences which never "jump up": each value either decreases with respect to the previous one, or stays equal, or increases by only one unit. This kind of sequences will be used to describe trees.

Definition 4 A *level sequence* or *depth sequence* is a sequence $(x_1, \ldots, x_n)$ of natural numbers 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 example, $x = (0, 1, 2, 3, 1, 2)$ is a level sequence that satisfies $|x| = 6$ or $x = (0) \cdot (0, 1, 2)^+ \cdot (0, 1)^+$. Now, we are ready to represent trees by means of level sequences.

Definition 5 We define a function $\langle \cdot \rangle$ from the set of ordered trees to the set of level sequences as follows. Let t be an ordered tree. If t is a single node, then $\langle t \rangle = (0)$. Otherwise, if t is composed of the trees $t_1, \ldots, t_k$ joined to a common root r (where the ordering $t_1, \ldots, t_k$ is the same of the children of r), then

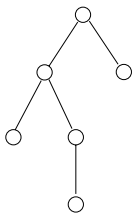
$$\langle t \rangle = (0) \cdot \langle t_1 \rangle^+ \cdot \langle t_2 \rangle^+ \cdot \ldots \cdot \langle t_k \rangle^+.$$

Here we will say that $\langle t \rangle$ is the level representation of t.

Note the role of the previous definition:

Proposition 2 *Level sequences are exactly the sequences of the form* $\langle t \rangle$ *for ordered, unranked trees* t.

That is, our encoding is a bijection between the ordered trees and the level sequences. This encoding $\langle t \rangle$ basically corresponds to a preorder traversal of t, where each number of the sequence represents the level of the current node in the traversal. As an example, the level representation of the tree



the level sequence $(0, 1, 2, 2, 3, 1)$. Note that, for example, the subsequence $(1, 2, 2, 3)$ corresponds to the bottom-up subtree rooted at the left son of the root (recall that our subsequences are adjacent). We can state this fact in general.

Proposition 3 *Let* $x = \langle t \rangle$, *where* t *is an ordered tree. Then,* t *has a bottom-up subtree* r *at level* $d > 0$ *if and only if* $\langle r \rangle + d$ *is a subsequence of* x.

Proof We prove it by induction on d. If $d = 1$, then $\langle r \rangle + d = \langle r \rangle^+$ and the property holds by the recursive definition of level representation.

For the induction step, let $d > 1$. To show one direction, suppose that r is a bottom-up subtree of t at level d. Then, r must be a bottom-up subtree of one of the bottom-up subtrees corresponding to the children of the root of t. Let t' be the bottom-up subtree at level 1 that contains r. Since r is at level $d - 1$ in t', the induction hypothesis states that $\langle r \rangle + d - 1$ is a subsequence of $\langle t' \rangle$. But $\langle t' \rangle^+$ is also, by definition, a subsequence of x. Combining both facts, we get that $\langle r \rangle + d$ is a subsequence of x, as desired. The argument also works in the contrary direction, and we get the equivalence. □

Fig. 7 The Ordered Subtree test algorithm

```
ORDERED_SUBTREE(st, t)
    Input: A tree st, a tree t.
    Output: true if st is a subtree of t.

1   pos_st = 1
2   pos_t = 1
3   while pos_st ≤ SIZE(st) and pos_t ≤ SIZE(t)
4         do if level (st, pos_st) > level (t, pos_t)
5               then while level (st, pos_st) ≠ level (t, pos_t)
6                         do pos_st = pos_st − 1
7            if level (st, pos_st) = level (t, pos_t)
8               then pos_st = pos_st + 1
9            pos_t = pos_t + 1
10  return pos_st > SIZE(st)
```

4.1 Subtree testing in ordered trees

Top-down subtree testing of two ordered trees can be obtained by performing a simultaneous preorder traversal of the two trees (Valiente 2002). This algorithm is shown in Fig. 7. There, pos_t traverses sequentially the level representation of tree t and pos_{st} similarly traverses the purported subtree st. The natural number found in the level representation of t at position pos_t is exactly level(t, pos_t).

Suppose we are given the trees st and t, and we would like to know if st is a subtree of t. Our method begins visiting the first node in tree t and the first node in tree st. While we are not visiting the end of any tree,

- If the level of tree t node is greater than the level of tree st node then we visit the next node in tree t.
- If the level of tree st node is greater than the level of tree t node then we backtrack to the last node in tree st that has the same level as *tree* node.
- If the level of the two nodes are equal then we visit the next node in tree t and the next node in tree st.

If we reach the end of tree st, then st is a subtree of tree t.

The running time of the algorithm is clearly quadratic since for each node of tree t, it may visit all nodes in tree st. An incremental version of this algorithm follows easily, as it is explained in next section.

5 Mining frequent ordered trees

In the rest of the paper, our goal will be to obtain a frequent closed tree mining algorithm for ordered and unordered trees. First, we present in this section a basic method for mining frequent ordered trees. We will extend it to unordered trees and frequent closed trees in the next section.

We begin showing a method for mining frequent ordered trees. Our approach here is similar to gSpan (Yan and Han 2002): we represent the potential frequent subtrees to be checked 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, we argue that all trees can be obtained in this way. This allows us to avoid extending trees that are found to be already nonfrequent.

We show now that our representation allows us to traverse the whole subtree space by an operation of extension by a single node, in a simple way.

Definition 6 Let x and y be two level sequences. We say that y is a *one-step extension* of x (in symbols, $x \vdash^1 y$) if x is a prefix of y and $|y| = |x| + 1$. We say that y is an *extension* of x (in symbols, $x \vdash y$) if x is a prefix of y.

Note that $x \vdash^1 y$ holds if and only if $y = x \cdot (i)$, where $1 \leq i \leq j + 1$, and j is the last element of x. Note also that a series of one-step extensions from (0) to a level sequence x

$$(0) \vdash^1 x_1 \vdash^1 \cdots \vdash^1 x_{k-1} \vdash^1 x$$

always exists and must be unique, since the x_i's can only be the prefixes of x. Therefore, we have:

Proposition 4 *For every level sequence x, there is a unique way to extend* (0) *into x.*

For this section we could directly use gSpan, since our structures can be handled by that algorithm. However, our goal is the improved algorithm described in the next section, to be applied when the ordering in the subtrees is irrelevant for the application, that is, mining closed unordered trees.

Indeed, level representations allow us to check only canonical representatives for the unordered case, thus saving the computation of support for all (except one) of the ordered variations of the same unordered tree. Figures 8 and 9 show the gSpan-based algorithm, which is as follows: beginning with a tree of single node, it calls recursively the FREQUENT_ORDERED_SUBTREE_MINING algorithm doing one-step extensions and checking that they are still frequent. Correctness and completeness follow from Propositions 2 and 4 by standard arguments.

Since we represent trees by level representations, we can speed up these algorithms, using an incremental version of the subtree ordered test algorithm explained in Sect. 4.1, reusing

Fig. 8 The Frequent Ordered Mining algorithm

```
FREQUENT_ORDERED_MINING(D, min_sup)
    Input: A tree dataset D, and min_sup.
    Output: The frequent tree set T.

1  t ← •
2  T ← ∅
3  T ← FREQUENT_ORDERED_SUBTREE_MINING(t, D, min_sup, T)
4  return T
```

```
FREQUENT_ORDERED_SUBTREE_MINING(t, D, min_sup, T)
    Input: A tree t, a tree dataset D, min_sup, and the frequent tree set T so far.
    Output: The frequent tree set T, updated from t.

1  insert t into T
2  for every t' that can be extended from t in one step
3        do if support(t') ≥ min_sup
4              then T ← FREQUENT_ORDERED_SUBTREE_MINING(t', D, min_sup, T)
5  return T
```

Fig. 9 The Frequent Ordered Subtree Mining algorithm

Fig. 10 The Incremental Ordered Subtree test algorithm

```
INCREMENTAL_ORDERED_SUBTREE(st, t, pos_st, pos_t)
    Input: A tree st, a tree t, and positions pos_st, pos_t
        such that the st prefix of length pos_st − 1 is a
        subtree of the t prefix of length pos_t.
    Output: true if st is a subtree of t.

1   while pos_st ≤ SIZE(st) and pos_t ≤ SIZE(t)
2       do if level (st, pos_st) > level (t, pos_t)
3             then while level (st, pos_st) ≠ level (t, pos_t)
4                     do pos_st = pos_st − 1
5          if level (st, pos_st) = level (t, pos_t)
6             then pos_st = pos_st + 1
7          pos_t = pos_t + 1
8   return pos_st > SIZE(st)
```

the node positions we reach at the end of the algorithm. If $st1$ is a tree extended from st in one step adding a node, we can start ORDERED_SUBTREE$(st1, t)$ proceeding from where ORDERED_SUBTREE(st, t) ended. So, we only need to store and reuse the positions pos_t and pos_{st} at the end of the algorithm. This incremental method is shown in Fig. 10. Note that ORDERED_SUBTREE can be seen as a call to INCREMENTAL_ORDERED_SUBTREE with pos_{st} and pos_t initialized to zero.

6 Unordered subtrees

In unordered trees, the children of a given node form sets of siblings instead of sequences of siblings. Therefore, ordered trees that only differ in permutations of the ordering of siblings are to be considered the same unordered tree.

6.1 Subtree testing in unordered trees

We can test if an unordered tree r is a subtree of an unordered tree t by reducing the problem to maximum bipartite matching. Figure 11 shows this algorithm.

Suppose we are given the trees r and t, whose components are $r_1, \ldots, r_n$ and $t_1, \ldots, t_k$, respectively. If $n > k$ or r has more nodes than t, then r cannot be a subtree of t. We recursively build a bipartite graph where the vertices represent the child trees of the trees and the edges the relationship "is subtree" between vertices. The function BIPARTITEMATCHING returns true if it exists a solution for this maximum bipartite matching problem. It takes time $O(n_r n_t^{1.5})$ (Valiente 2002), where n_r and n_t are the number of nodes of r and t, respectively. If BIPARTITEMATCHING returns true then we conclude that r is a subtree of t.

To speed up this algorithm, we store the computation results of the algorithm in a dictionary D, and we try to reuse these computations at the beginning of the algorithm.

6.2 Mining frequent closed subtrees in the unordered case

The main result of this subsection is a precise mathematical characterization of the level representations that correspond to canonical variants of unordered trees. Luccio et al. (2004, 2001) showed that a canonical representation based on the preorder traversal can be obtained in linear time. Nijssen and Kok (2003), Chi et al. (2005) and Asai et al. (2003) defined similar canonical representations.

```
UNORDERED_SUBTREE(r, t)
    Input: A tree r, a tree t.
    Output: true if r is a subtree of t.

 1  if D(r, t) exists
 2     then Return D(r, t)
 3  if (SIZE(r) > SIZE(t) or #COMPONENTS(r) > #COMPONENTS(t))
 4     then Return false
 5  if (r = •)
 6     then Return true
 7  graph ← {}
 8  for each s_r in SUBCOMPONENTS(r)
 9        do for each s_t in SUBCOMPONENTS(t)
10              do if (UNORDERED_SUBTREE(s_r, s_t))
11                    then insert(graph, edge(s_r, s_t))
12  if BIPARTITEMATCHING(graph)
13     then D(r, t) ← true
14     else  D(r, t) ← false
15  return D(r, t)
```

Fig. 11 The Unordered Subtree test algorithm

We select 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. More precisely,

Definition 7 Let t be an unordered tree, and let $t_1, \ldots, t_n$ be all the ordered trees obtained from t by ordering in all possible ways all the sets of siblings of t. The *canonical representative* of t is the ordered tree t_0 whose level representation is maximal (according to lexicographic ordering) among the level representations of the trees t_i, that is, such that

$$\langle t_0 \rangle = \max\{\langle t_i \rangle \mid 1 \leq i \leq n\}.$$

We can use, actually, the same algorithm as in the previous section to mine unordered trees; however, much work is unnecessarily spent in checking repeatedly ordered trees that correspond to the same unordered tree as one already checked. A naive solution is to compare each tree to be checked with the ones already checked, but in fact this is an inefficient process, since all ways of mapping siblings among them must be tested.

A far superior solution would be obtained if we could count frequency only for canonical representatives. We prove next how this can be done: the use of level representations allows us to decide whether a given (level representation of a) tree is canonical, by using an intrinsic characterization, stated in terms of the level representation itself.

Theorem 2 *A level sequence x corresponds to a canonical representative if and only if for any level sequences y, z and any $d \geq 0$ such that $(y + d) \cdot (z + d)$ is a subsequence of x, it holds that $y \geq z$ in lexicographical order.*

Proof Suppose that x corresponds to a canonical representative and that $(y + d) \cdot (z + d)$ is a subsequence of x for some level sequences y, z and $d \geq 0$. In this case, both $y + d$ and $z + d$ are subsequences of x and, by Proposition 3, $\langle y \rangle$ and $\langle z \rangle$ are two subtrees of $\langle x \rangle$. Since their respective level representations, y and z, appear consecutively in x, the two subtrees must be siblings. Now, if $y < z$, the reordering of siblings y and z would lead to a bigger

level representation of the same unordered tree, and x would not correspond to a canonical representative. Therefore, $y \geq z$ in lexicographical order.

For the other direction, suppose that x does not correspond to a canonical representative. Then, the ordered tree t represented by x would have two sibling subtrees r and s (appearing consecutively in t, say r before s) that, if exchanged, would lead to a lexicographically bigger representation. Let $y = \langle r \rangle$ and $z = \langle s \rangle$. If r and s are at level d in t, then $(y + d) \cdot (z + d)$ would be a subsequence of $x = \langle t \rangle$ (again by Proposition 3). Then, it must hold that $y < z$ in lexicographical order. □

Corollary 3 *Let a level sequence x correspond to a canonical representative. Then its extension $x \cdot (i)$ corresponds to a canonical representative if and only if, for any level sequences y, z and any $d \geq 0$ such that $(y + d) \cdot (z + d)$ is a suffix of $x \cdot (i)$, it holds that $y \geq z$ in lexicographical order.*

Proof Suppose that x corresponds to a canonical representative, and let i be such that $x \cdot (i)$ is a level sequence. At this point, we can apply Theorem 2 to $x \cdot (i)$: it is a canonical representative if and only if all subsequences of the form $(y + d) \cdot (z + d)$ (for appropriate y, z, and d) satisfy that $y \geq z$. But such subsequences $(y + d) \cdot (z + d)$ can now be divided into two kinds: the ones that are subsequences of x and the ones that are suffixes of $x \cdot (i)$.

A new application of Theorem 2 to x assures that the required property must hold for subsequences of the first kind. So, we can characterize the property that $x \cdot (i)$ corresponds to a canonical representative just using the subsequences of the second kind (that is, suffixes) as said in the statement. □

We build an incremental canonical checking algorithm, using the result of Corollary 3. The algorithm is as follows: each time we add a node of level d to a tree t, we check for all levels less than d that the last two child subtrees are correctly ordered. As it is an incremental algorithm, and the tree that we are extending is canonical, we can assume that child subtrees are ordered, so we only have to check the last two ones.

6.3 Closure-based mining

In this section, we propose TREENAT, a new algorithm to mine frequent closed trees. Figure 12 illustrates the framework.

Figure 13 shows the pseudocode of CLOSED_UNORDERED_SUBTREE_MINING. It is similar to UNORDERED_SUBTREE_MINING, adding a checking of closure in lines 10–13. Correctness and completeness follow from Propositions 2 and 4, and Corollary 3.

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 our method does not. That means that with a small number of labels, CMTreeMiner will need to store a huge number of occurrences, and it will take much more time and memory than our method, that doesn't need to store all that information. Also, with unlabeled trees, if the tree size is big, CMTreeMiner needs more time and memory to store all possible occurrences. For example, an unlabeled tree of size 2 in a tree of size n has $n - 1$ occurrences. But when the number of labels is big, or the size of the unlabeled trees is small, CMTreeMiner will be fast because the number of occurrences is small and it can use the power of its pruning methods. Dealing with unordered trees, CMTreeMiner doesn't use bipartite matching as we do for subtree testing. However, it uses canonical forms and the storing of all occurrences.

Fig. 12 The Closed Unordered Mining algorithm

```
CLOSED_UNORDERED_MINING(D, min_sup)
    Input: A tree dataset D, and min_sup.
    Output: The closed tree set T.

1  t ← •
2  T ← ∅
3  T ← CLOSED_UNORDERED_SUBTREE_MINING(t, D, min_sup, T)
4  return T
```

```
CLOSED_UNORDERED_SUBTREE_MINING(t, D, min_sup, T)
    Input: A tree t, a tree dataset D, min_sup, and the closed frequent tree set T so far.
    Output: The closed frequent tree set T, updated from t.

1  if t ≠ CANONICAL_REPRESENTATIVE(t)
2     then return T
3  t_is_closed ← TRUE
4  for every t' that can be extended from t in one step
5        do if support(t') ≥ min_sup
6              then T ← CLOSED_UNORDERED_SUBTREE_MINING(t', D, min_sup, T)
7        do if support(t') = support(t)
8              then t_is_closed ← FALSE
9  if t_is_closed = TRUE
10    then insert t into T
11 if (∃t'' ∈ T | t'' is subtree of t, support(t) =support(t''))
12    then delete t'' from T
13 return T
```

Fig. 13 The Closed Unordered Subtree Mining algorithm

7 Induced subtrees and labeled trees

Our method can be extended easily to deal with induced subtrees and labeled trees in order to compare it with CMTreeMiner in Sect. 8, working with the same kind of trees and subtrees.

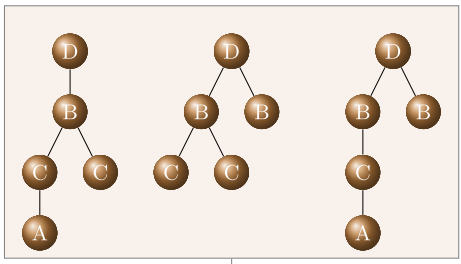
7.1 Induced subtrees

In order to adapt our algorithms to all induced subtrees, not only rooted, we need to change the subtree testing procedure with a slight variation. We build a new procedure for checking if a tree r is an induced subtree of t using the previous procedure SUBTREE(r, t) (ORDERED_SUBTREE(r, t) for ordered trees or UNORDERED_SUBTREE(r, t) for unordered trees) that checks whether a tree r is a top-down subtree of tree t. It is as follows: for every node n in tree t we consider the top-down subtree t' of tree t rooted at node n. If there is at least one node that SUBTREE (r, t') returns true, then r is an induced subtree of t, otherwise not. Applying this slight variation to both ordered and unordered trees, we are able to mine induced subtrees as CMTreeMiner.

7.2 Labeled trees

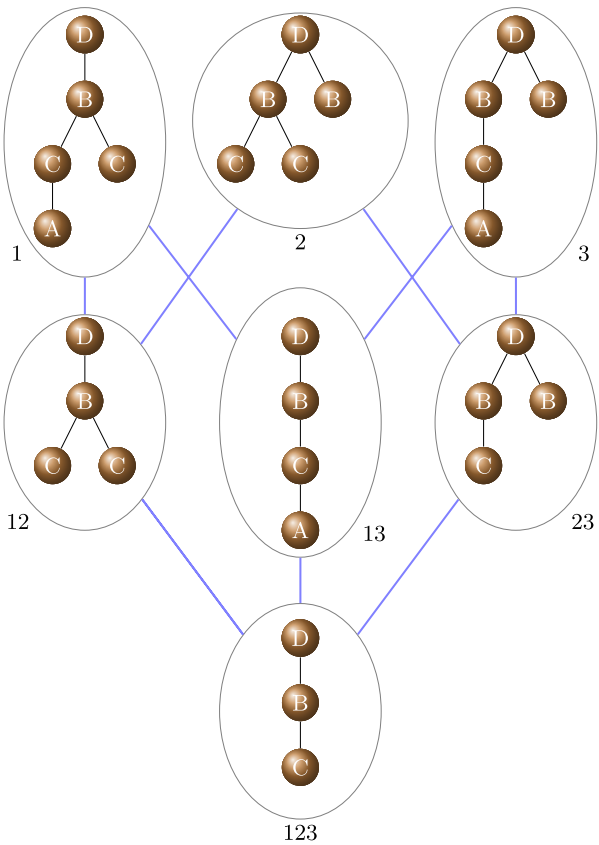
We need to use a new tree representation to deal with labels in the nodes of the trees. We represent each labeled tree using labeled level sequences (Asai et al. 2003; Nijssen and Kok 2003), a labeled extension of the level representations explained earlier.

Fig. 14 A dataset example



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

Fig. 15 Example of Galois Lattice of Closed trees



Definition 8 A *labeled level sequence* is a sequence $((x_1, l_1), \ldots, (x_n, l_n))$ of pairs of natural numbers and labels such that $x_1 = 0$ and each subsequent number x_{i+1} belongs to the range $1 \leq x_{i+1} \leq x_i + 1$.

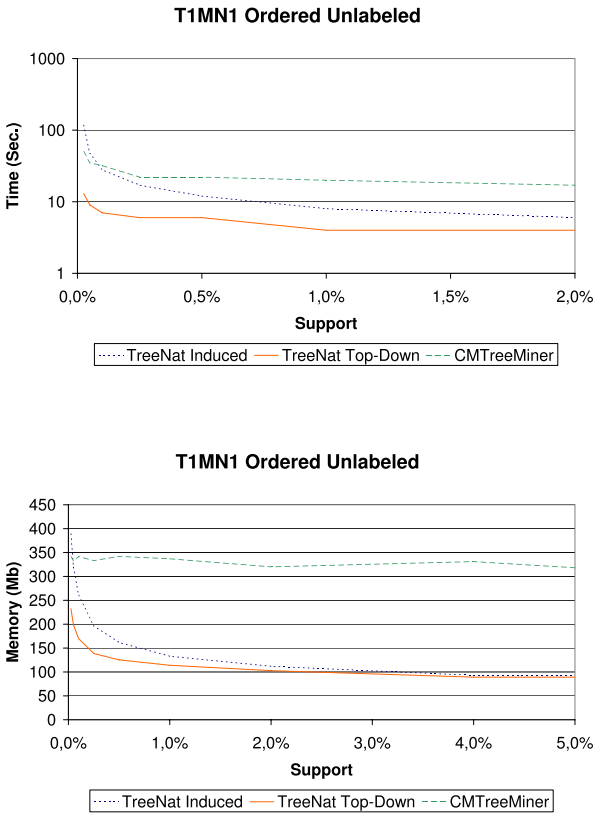


Fig. 16 Synthetic data experimental results on Ordered Trees: Support versus Running Time and Memory

For example, $x = ((0, A), (1, B), (2, A), (3, B), (1, C))$ is a level sequence that satisfies $|x| = 6$ or $x = ((0, A)) \cdot ((0, B), (1, A), (2, B))^+ \cdot ((0, C))^+$. Now, we are ready to represent trees by means of level sequences (see also Chi et al. 2004).

Definition 9 We define a function $\langle \cdot \rangle$ from the set of ordered trees to the set of labeled level sequences as follows. Let t be an ordered tree. If t is a single node, then $\langle t \rangle = ((0, l_0))$. Otherwise, if t is composed of the trees $t_1, \ldots, t_k$ joined to a common root r (where the ordering $t_1, \ldots, t_k$ is the same of the children of r), then

$$\langle t \rangle = ((0, l_0)) \cdot \langle t_1 \rangle^+ \cdot \langle t_2 \rangle^+ \cdot \ldots \cdot \langle t_k \rangle^+.$$

Here we will say that $\langle t \rangle$ is the labeled level representation of t.

This encoding is a bijection between the ordered trees and the labeled level sequences. This encoding $\langle t \rangle$ basically corresponds to a preorder traversal of t, where each natural number of the node sequence represents the level of the current node in the traversal.

Figure 14 shows a finite dataset example using labeled level sequences.

The closed trees for the dataset of Fig. 14 are shown in the Galois lattice of Fig. 15.

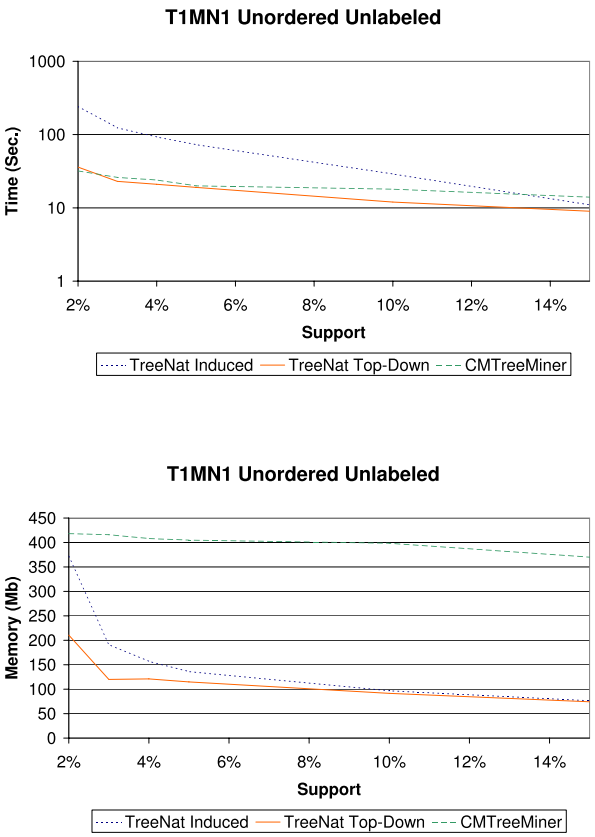


Fig. 17 Synthetic data experimental results on Unordered Trees: Support versus Running Time and Memory

8 Applications

We tested our algorithms on synthetic and real data, and compared the results with CMTreeMiner (Chi et al. 2001b).

All experiments were performed on a 2.0 GHz Intel Core Duo PC machine with 2 Gigabyte main memory, running Ubuntu 7.10. As far as we know, CMTreeMiner is the state-of-art algorithm for mining induced closed frequent trees in databases of rooted trees.

8.1 Datasets for mining closed frequent trees

We present the datasets used in this section for empirical evaluation of our closed frequent tree mining methods. GAZELLE is a new unlabeled tree dataset. The other datasets are the most used ones in frequent tree mining literature.

– ZAKI Synthetic Datasets. Datasets generated by the tree generator of Zaki (2002). 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 all the probabilities is 1. Using the master tree, the dataset is generated selecting subtrees according to these probabilities. It was used in CMTreeMiner (Chi et al. 2001b) empirical evaluation.

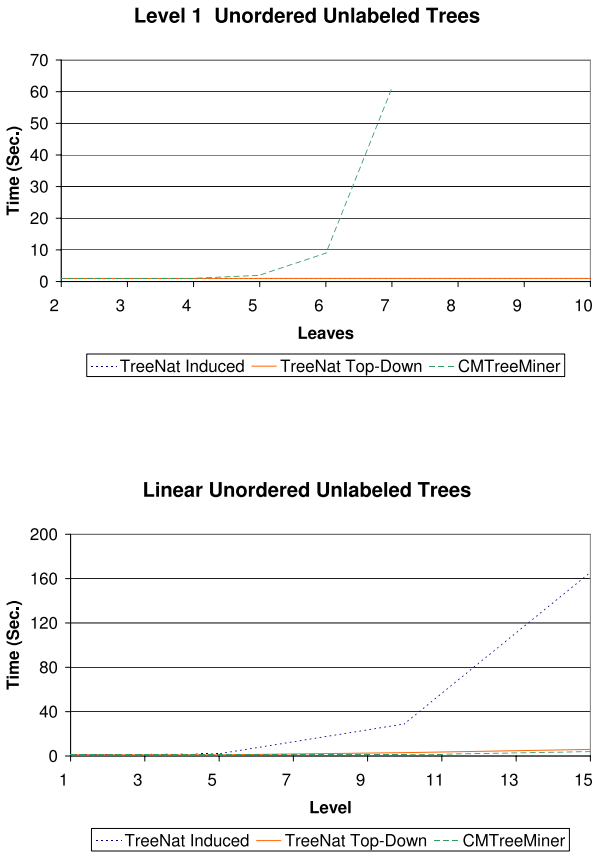


Fig. 18 Synthetic data experimental results on Unordered Trees varying the number of nodes: Support versus Running Time on level 1 trees and on linear trees

- CSLOGS Dataset (Zaki 2002). It is available from Zaki's web page. 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 CSLOGS dataset contains 59,691 trees. The average tree size is 12.
- NASA multicast data (Chalmers and Almeroth 2001). The data was measured during the NASA shuttle launch between 14th and 21st of February, 1999. It has 333 vertices where each vertex takes an IP address as its label. Chi et al. (2001b) sampled the data from this NASA data set in 10 minute sampling intervals and got a data set with 1,000 transactions. Therefore, the transactions are the multicast trees for the same NASA event at different times.
- GAZELLE Dataset. It is obtained from KDD Cup 2000 data (Kohavi et al. 2000). This dataset is a web log file of a real internet shopping mall (gazelle.com). This dataset of size 1.2 GB contains 216 attributes. We use the attribute 'Session ID' to associate to each user session a unique tree. The trees record the sequence of web pages that have been visited in a user session. Each node tree represents a content, assortment and product path. Trees are not built using the structure of the web site, instead they are built following the user streaming. Each time a user visit a page, if he has not visited it before, we take this page as a new deeper node, otherwise, we backtrack to the node this page corresponds to, if it is the last node visited on a concrete level. The resulting dataset consists of 225,558 trees.

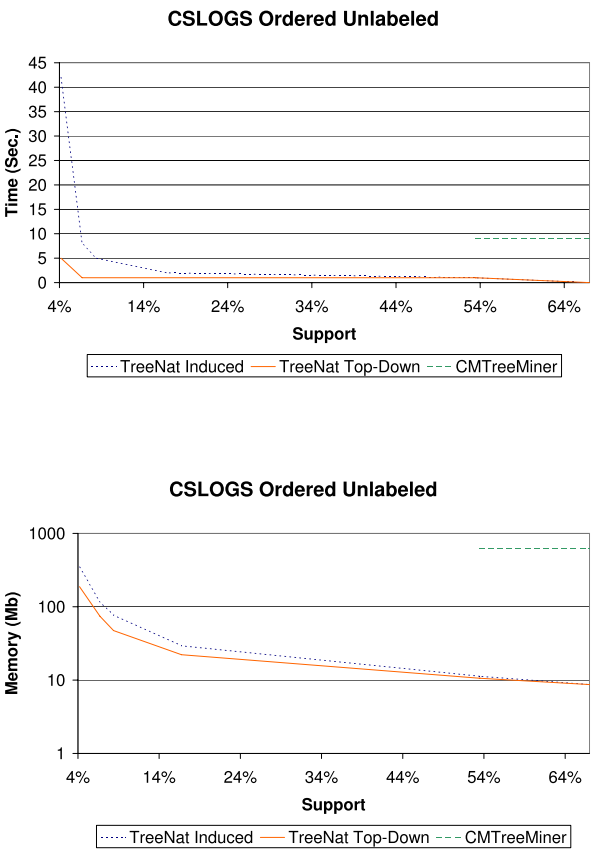


Fig. 19 CSLOGS real data experimental results on Ordered Trees: Support versus Running Time and Memory

8.2 Unlabeled trees

We compare two methods of TREENAT, our algorithm for obtaining closed frequent trees, with CMTreeMiner. The first one is TREENAT TOP-DOWN that obtains top-down subtrees and the second one is TREENAT INDUCED that works with induced subtrees.

On synthetic data, we use the ZAKI Synthetic Datasets for rooted ordered trees restricting the number of distinct node labels to one. We call this dataset T1MN1.

In the T1MN1 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 level of the tree is $D = 10$, the maximum fanout is $F = 10$ and the number of trees in the dataset is $T = 1{,}000{,}0000$.

The results of our experiments on synthetic data are shown in Figs. 16 and 17. We see there that our algorithm TREENAT compares well to CMTreeMiner for top-down subtrees, using less memory in both ordered and unordered cases. Our induced subtree algorithm has similar performance to CMTreeMiner in the ordered case, but it's a bit worse for the unordered case, due to the fact that we take care of avoiding repetitions of structures that are isomorphic under the criterion of unordered trees (which CMTreeMiner would not prune). In these experiments the memory that our method uses depends mainly on the support, not as CMTreeMiner.

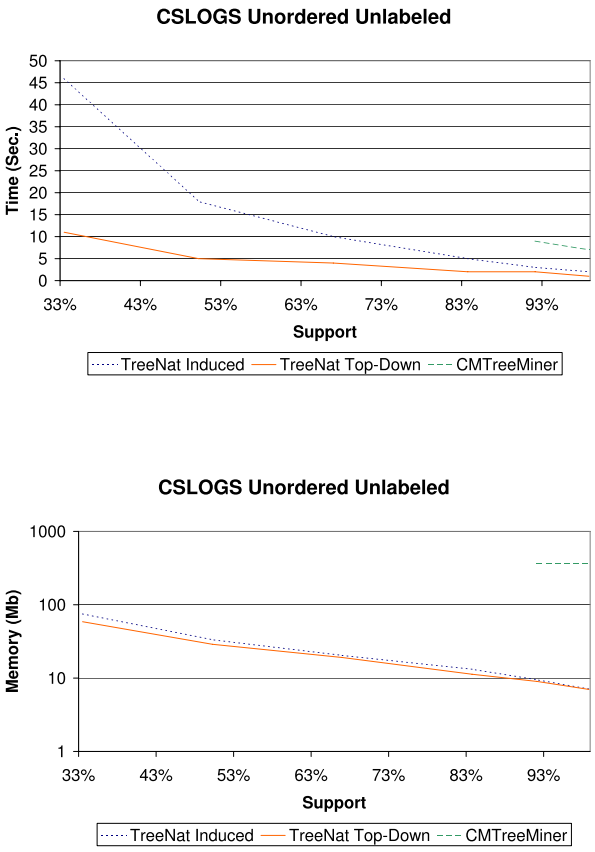


Fig. 20 CSLOGS real data experimental results on Unordered Trees: Support versus Running Time and Memory

In order to understand the behavior of TREENAT and CMTreeMiner respect to the tree structure of input data, we compare the mining performances of TREENAT and CMTreeMiner for two sets of 10,000 identical unlabeled trees, one where all the trees are linear with 10 nodes and another one where all the trees are of level 1 with 10 nodes (1 root and 9 leaves). We notice that

- CMTreeMiner cannot mine the dataset with unordered trees of level 1 and 10 nodes. The maximum number of nodes of unordered trees that CMTreeMiner is capable of mining is 7.
- TREENAT INDUCED has worst performance than CMTreeMiner for linear trees. However, TREENAT TOP-DOWN has similar results to CMTreeMiner.

Figure 18 shows the results of these experiments varying the number of nodes. CMTreeMiner outperforms TREENAT with linear trees, and TREENAT outperforms CMTreeMiner with trees of level 1. CMTreeMiner needs to store all subtree occurrences, but it can use it pruning methods. When the number of leaf nodes is large, the number of occurrences is large and CMTreeMiner has to keep a huge quantity of occurrences. When the trees are linear, CMTreeMiner uses its pruning techniques to outperform TREENAT INDUCED.

We tested our algorithms on two real datasets. The first one is the CSLOGS Dataset. As it is a labeled dataset, we changed it to remove the labels for our experiments with

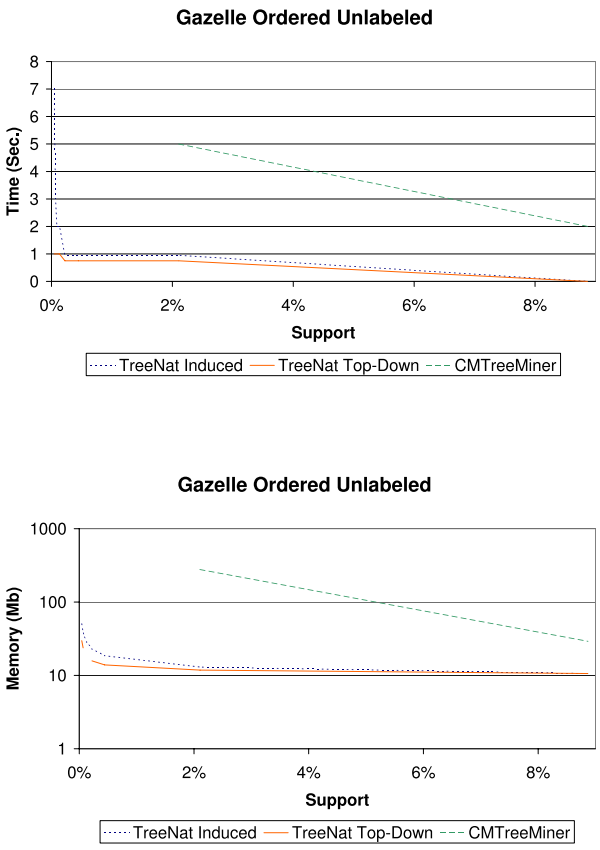


Fig. 21 Gazelle real data experimental results on Ordered Trees: Support versus Running Time and Memory

unlabeled trees. Figures 19 and 20 show the results. We see that CMTreeminer needs more than 1 GB of memory to execute for supports lower than 31,890 in the ordered case and 50,642 for the unordered case. The combinatorial complexity of this dataset seems too hard for CMTreeMiner, since it stores all occurrences of all possible subtrees of one label.

The second real dataset is GAZELLE. Figures 21 and 22 show the results of our experiments on this real-life data: we see that our method is better than CMTreeMiner at all values of support, both for ordered and unordered approaches. Again CMTreeMiner needs more memory than available to run for small supports.

Finally, we tested our algorithms using the NASA multicast data. Neither CMTreeMiner or our method could mine the data considering it unlabeled. The combinatorics are too hard to try to solve it using less than 2 GB of memory. An incremental method could be useful.

8.3 Labeled trees

On synthetic data, we use the same dataset as for the unlabeled case. In brief, a mother tree is generated first with the following parameters: the number of distinct node labels from $N = 1$ to $N = 100$, the total number of nodes in the tree $M = 10{,}000$, the maximal level 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 $T = 8{,}000{,}000$.

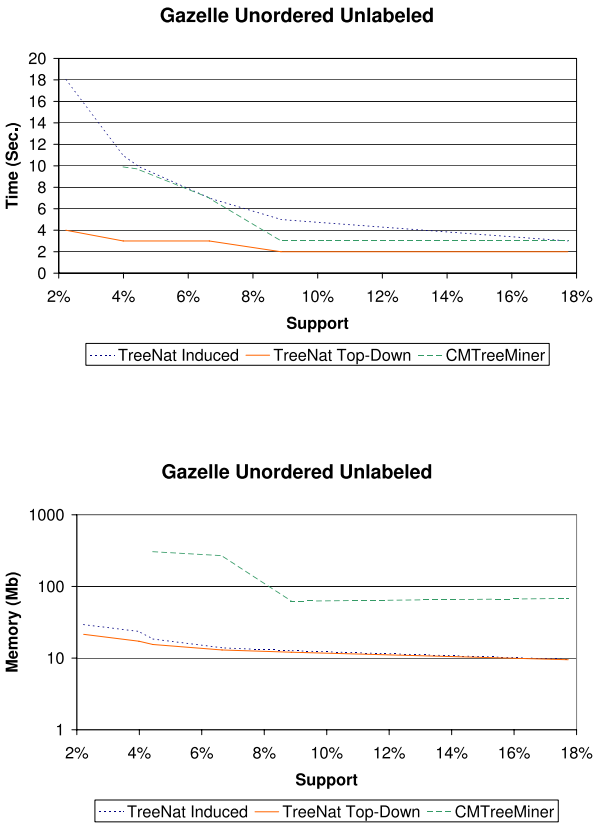


Fig. 22 Gazelle real data experimental results on Unordered Trees: Support versus Running Time and Memory

Figures 23 and 24 show the results of our experiments on these artificial data: we see that our method outperforms CMTreeMiner if the number of labels is small, but CMTreeMiner wins for large number of labels, both for ordered and unordered approaches. On the size of datasets, we observe that the time and memory needed for our method and CMTreeMiner are linear respect the size of the dataset. Therefore, in order to work with bigger datasets, an incremental method is needed.

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 our method does not. CMTreeMiner uses occurrences and pruning techniques based on them. TREENAT doesn't store occurrences. For labeled trees with a small number of labels, CMTreeMiner will need to store a huge number of occurrences, and it will take much more time and memory than TREENAT, that doesn't need to store all that information. Also, with unlabeled trees, if the tree size is big, CMTreeMiner needs more time and memory to store all possible occurrences. But if the number of labels is big, CMTreeMiner will be fast because the number of occurrences is small and it can use the power of its pruning methods.

On real dataset CSLOGS, CMTreeMiner outperforms our method as the number of labels is not low as shown in Fig. 25.

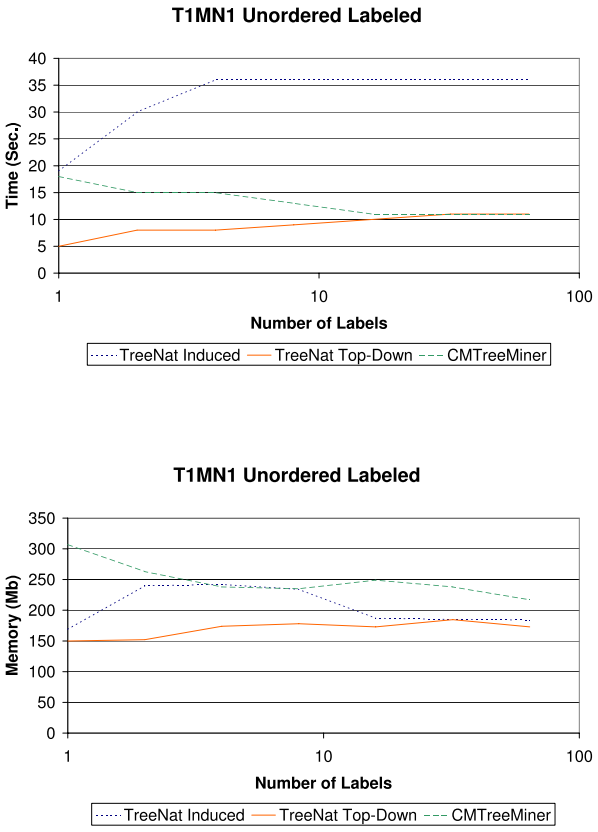


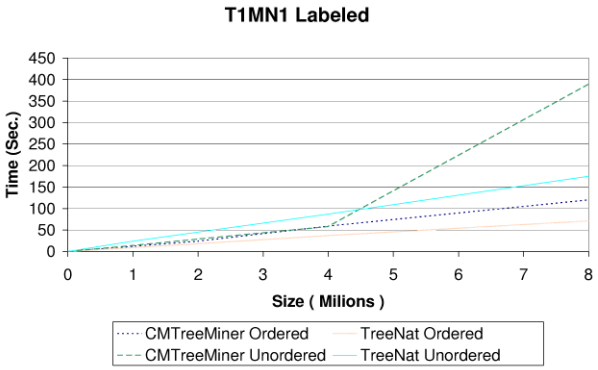
Fig. 23 Synthetic data experimental results on Labeled Trees: Number of Labels versus Running Time and Memory

9 Conclusions

We have described a rather formal study of trees from the point of view of closure-based mining. Progressing beyond the plain standard support-based definition of a closed tree, we have developed a rationale (in the form of the study of the operation of intersection on trees, both in combinatorial and algorithmic terms) for defining a closure operator, not on trees but on sets of trees, and we have indicated the most natural definition for such an operator; we have provided a mathematical study that characterizes closed trees, defined through the plain support-based notion, in terms of our closure operator, plus the guarantee that this structuring of closed trees gives us the ability to find the support of any frequent tree. Our study has provided us, therefore, with a better understanding of the closure operator that stands behind the standard support-based notion of closure, as well as basic algorithmics on the data type. Then we have presented efficient algorithms for subtree testing and for mining ordered and unordered frequent closed trees.

A number of variants have suggested themselves for further study: we have evaluated the behavior of our algorithms if we take into account labels, a case where our algorithm does not fare as well as in the unlabeled case; and we have considered also induced subtrees.

We believe that the sequential form of the representation used, where the number-encoded level furnishes the two-dimensional information, is key in the fast processing of the data, and will be useful in further studies, algorithms, and applications of similar techniques. In particular, our recent work (Balcázar et al. 2008) includes an analysis of the



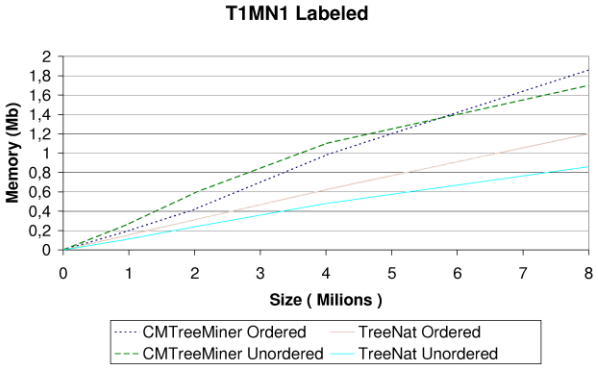


Fig. 24 Synthetic data experimental results on Labeled Trees: Dataset Size versus Running Time and Memory

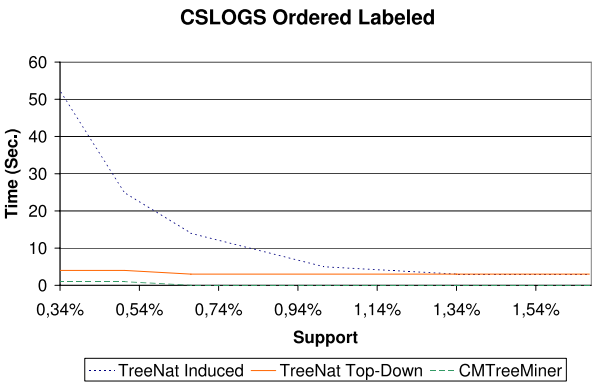


Fig. 25 CSLOGS real data experimental results on labeled ordered trees: Support versus Running Time

extraction of association rules of full confidence out of the closed sets of trees, along the same lines as the corresponding process on itemsets, and we have found there an interesting phenomenon that does not appear if other combinatorial structures are analyzed: rules whose propositional counterpart is nontrivial are, however, always implicitly true in trees due to the peculiar combinatorics of the structures. That study is not yet finished since we have powerful heuristics to treat those implicit rules but wish to obtain a full mathemat-

ical characterization. Additionally, we have recently tackled the problem of constructing closed sets of trees in the case where the dataset is so large that it is not possible to store it: we have proposed a development of algorithms based on those reported here for a Data Stream model (Bifet and Gavaldà 2008). We hope to obtain further progress along these lines: confidence-bounded association rules are not yet really understood, and the problem of how to find them in the Data Stream model is also an interesting issue, worthy of further study. We also keep in mind options for further work of a more applied nature, such as our original motivation for this study, namely, the potential advantages of running closure-based tree mining on navigation patterns in order to improve a novel, decentralized, adaptive web crawler that resorts to a P2P-style of cooperation to offer a new notion of web search engine.

Acknowledgements We are grateful to the reviewers of each of the many versions of the papers containing the results described here, both conference and workshop papers and versions of the present journal paper, for their thoughtful efforts and useful comments. Partially supported by the European Union PASCAL2 Network of Excellence (FP7-ICT-216886), and European Union PASCAL Network of Excellence (IST-2002-506778), and by projects European Union DELIS (IST-001907), Spanish Government iDEAS (TIN2004-04343), Spanish Government SESAAME-BAR (TIN2008-06582-C03-01), Spanish Government MOISES-BAR (TIN2005-08832-C03-03), Spanish Government TRANGRAM (TIN2004-07925-C03-02), and Spanish Government FORMALISM (TIN2007-66523).

References

Arimura, H., & Uno, T. (2005). An output-polynomial time algorithm for mining frequent closed attribute trees. In *ILP* (pp. 1–19).

Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., & Arikawa, S. (2002). Efficient substructure discovery from large semi-structured data. In *SDM*.

Asai, T., Arimura, H., Uno, T., & Nakano, S.-I. (2003). Discovering frequent substructures in large unordered trees. In *Discovery science* (pp. 47–61).

Baixeries, J., & Balcázar, J. L. Discrete deterministic data mining as knowledge compilation. In *Workshop on discrete mathematics and data mining at SIAM DM conference*.

Balcázar, J. L., Bifet, A., & Lozano, A. (2007). Mining frequent closed unordered trees through natural representations. In *ICCS 2007, 15th international conference on conceptual structures* (pp. 347–359).

Balcázar, J. L., & Garriga, G. C. (2005). On Horn axiomatizations for sequential data. In *ICDT* (pp. 215–229) (extended version in *Theoretical Computer Science*, *371*, 247–264, 2007).

Balcázar, J. L., Bifet, A., & Lozano, A. (2006). Intersection algorithms and a closure operator on unordered trees. In *MLG 2006, 4th international workshop on mining and learning with graphs*.

Balcázar, J. L., Bifet, A., & Lozano, A. (2007). Subtree testing and closed tree mining through natural representations. In *DEXA workshops* (pp. 499–503).

Balcázar, J. L., Bifet, A., & Lozano, A. (2008). Mining implications from lattices of closed trees. In *Extraction et gestion des connaissances (EGC'2008)* (pp. 373–384).

Beyer, T., & Hedetniemi, S. M. (1980). Constant time generation of rooted trees. *SIAM Journal on Computing*, *9*(4), 706–712.

Bifet, A., & Gavaldà, R. (2008). Mining adaptively frequent closed unlabeled rooted trees in data streams. In *14th ACM SIGKDD international conference on knowledge discovery and data mining* (2008).

Chakrabarti, S. (2002). *Mining the web: analysis of hypertext and semi structured data*. San Mateo: Morgan Kaufmann.

Chalmers, R., & Almeroth, K. (2001). Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proceedings of the IEEE INFOCOM'2001*, April 2001.

Chi, Y., Muntz, R., Nijssen, S., & Kok, J. (2001a). Frequent subtree mining—an overview. *Fundamenta Informaticae*, *XXI*, 1001–1038.

Chi, Y., Xia, Y., Yang, Y., & Muntz, R. (2001b). Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *Fundamenta Informaticae*, *XXI*, 1001–1038.

Chi, Y., Yang, Y., & Muntz, R. R. (2004). Hybridtreeminer: an efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In *SSDBM '04: proceedings of the 16th international conference on scientific and statistical database management (SSDBM'04)* (p. 11). Washington: IEEE Computer Society.

Chi, Y., Yang, Y., & Muntz, R. R. (2005). Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*, *8*(2), 203–234.

Ganter, B., & Wille, R. (1999). *Formal concept analysis*. Berlin: Springer.

Garriga, G. C. (2006). *Formal methods for mining structured objects*. PhD thesis.

Garriga, G. C., & Balcázar, J. L. (2004). Coproduct transformations on lattices of closed partial orders. In *ICGT* (pp. 336–352).

Hashimoto, K., Aoki-Kinoshita, K. F., Ueda, N., Kanehisa, M., & Mamitsuka, H. (2008). A new efficient probabilistic model for mining labeled ordered trees applied to glycobiology. *ACM Transactions on Knowledge Discovery from Data*, *2*(1), 1–30.

Hein, J., Jiang, T., Wang, L., & Zhang, K. (1995). On the complexity of comparing evolutionary trees. In Z. Galil & E. Ukkonen (Eds.) *Proceedings of the 6th annual symposium on combinatorial pattern matching* (pp. 177–190), Espoo, Finland. Berlin: Springer.

Nakano, S. I., & Uno, T. (2003). *Efficient generation of rooted trees* (Tech. Rep. NII-2003-005e). National Institute for Informatics, Japan.

Knuth, D. E. (1997). *The art of computer programming*: *Vol. 1. Fundamental algorithms* (3rd ed.). Redwood City: Addison-Wesley/Longman.

Knuth, D. E. (2005). *The art of computer programming*: *Vol. 4. Fascicle 4: generating all trees—history of combinatorial generation*. New York: Addison-Wesley.

Kohavi, R., Brodley, C., Frasca, B., Mason, L., & Zheng, Z. (2000). KDD-Cup 2000 organizers' report: peeling the onion. *SIGKDD Explorations*, *2*(2), 86–98.

Liu, T.-L., & Geiger, D. (1999). Approximate tree matching and shape similarity. In *ICCV* (pp. 456–462).

Luccio, F., Enriquez, A. M., Rieumont, P. O., & Pagli, L. (2001). *Exact rooted subtree matching in sublinear time* (Technical Report TR-01-14). Università Di Pisa.

Luccio, F., Enriquez, A. M., Rieumont, P. O., & Pagli, L. (2004). *Bottom-up subtree isomorphism for unordered labeled trees*.

Nijssen, S., & Kok, J. N. (2003). Efficient discovery of frequent unordered trees. In *First international workshop on mining graphs, trees and sequences* (pp. 55–64).

Plotkin, J. M., & Rosenthal, J. W. (1994). How to obtain an asymptotic expansion of a sequence from an analytic identity satisfied by its generating function. *Journal of the Australian Mathematical Society (Series A)*, *56*, 131–143.

Shasha, D., Wang, J. T. L., & Zhang, S. (2004). Unordered tree mining with applications to phylogeny. In *ICDE '04: proceedings of the 20th international conference on data engineering* (p. 708). Washington: IEEE Computer Society.

Termier, A., Rousset, M.-C., & Sebag, M. (2004). DRYADE: a new approach for discovering closed frequent trees in heterogeneous tree databases. In *ICDM* (pp. 543–546).

Termier, A., Rousset, M.-C., Sebag, M., Ohara, K., Washio, T., & Motoda, H. (2008). DryadeParent, an efficient and robust closed attribute tree mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, *20*(3), 300–320.

Valiente, G. (2002). *Algorithms on trees and graphs*. Berlin: Springer.

Weiss, S., Indurkhya, N., Zhang, T., & Damerau, F. (2004). *Text mining: predictive methods for analyzing unstructured information*. Berlin: Springer.

Xiao, Y., Yao, J.-F., Li, Z., & Dunham, M. H. (2003). Efficient data mining for maximal frequent subtrees. In *ICDM '03: proceedings of the third IEEE international conference on data mining* (p. 379). Washington: IEEE Computer Society.

Yan, X., & Han, J. (2002). gspan: graph-based substructure pattern mining. In *ICDM '02: proceedings of the 2002 IEEE international conference on data mining (ICDM'02)* (p. 721). Washington: IEEE Computer Society.

Yan, X., & Han, J. (2003). Closegraph: mining closed frequent graph patterns. In *KDD '03: proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 286–295). New York, ACM Press.

Yan, X., Han, J., & Afshar, R. (2003). Clospan: mining closed sequential patterns in large databases. In *SDM*.

Yan, X., Zhou, X. J., & Han, J. (2005). Mining closed relational graphs with connectivity constraints. In *KDD '05: proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining* (pp. 324–333). New York: ACM Press.

Zaki, M. J. (2002). Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD international conference on knowledge discovery and data mining*.

Zaki, M. J. (2005). Efficiently mining frequent embedded unordered trees. *Fundamenta Mathematicae*, *66*(12), 33–52.