Intersection Algorithms and a Closure Operator on Unordered Trees

José L. Balcázar, Albert Bifet and Antoni Lozano

Universitat Politècnica de Catalunya,
{balqui,abifet,antoni}@lsi.upc.edu

Abstract. Link-based data may be studied formally by means of unordered trees. On a dataset formed by such link-based data, a natural notion of support-based closure can be immediately defined. Abstracting information from subsets of such data requires, first, a formal notion of intersection; second, deeper understanding of the notion of closure; and, third, efficient algorithms for computing intersections on unordered trees. We provide answers to these three questions.

1 Introduction

Closure-based mining is well-established by now as one of the various approaches to summarize subsets of a large dataset. Sharing some of the attractive features of frequency-based summarization of subsets, it offers an alternative view with both downsides and advantages; among the latter, 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.

Closure-based mining on itemsets is, by now, well understood, and there are interesting algorithmic developments; thus, there have been subsequent efforts in moving towards closure-based mining on structured data, particularly sequences, trees and graphs; see the survey [4] and the references there. 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 ([5], see below); 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 [10] or nonrepeated labels at all in sequences [8]).

Here 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 unordered, unlabeled, rooted trees; thus, our only

relevant information is the root and the link structure (so that the appropriate notion of subtree, so-called top-down subtree, preserves root and links), and solving the intersection problem along the same lines as in [3]. Thus, we only focus on the basic operations needed to phrase closure-based mining on such structures, but with a mathematically very demanding approach. We first formalize our structures and the notion of a tree being contained in another. We also evaluate the quantity of such combinatorial structures. We then move on to start our study of closure-based mining. Following the same path as in [7], we first need a notion of intersection: we will see that a natural notion of intersection of trees gives rise to intersection sets of trees, rather than individual trees, as with the sequences in [7]. We study the cardinality of such intersection sets, which we prove can be exponential in the worst case, although preliminary experiments suggest that intersection sets of cardinality beyond 1 hardly ever arise unless looked for.

We then 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. We complete this paper with preliminary algorithmic studies. We propose a natural recursive algorithm to compute intersections, and a more sophisticated method following a dynamic programming scheme; preliminary comparisons suggest that the dynamic programming algorithm is several orders of magnitude faster.

2 Preliminaries

2.1 Definitions

We will deal with rooted undirected trees with nodes of unbounded arity. This kind of trees will be referred throughout the paper simply as *trees*. The set of all trees will be denoted with $\mathcal{T}$. Additionally, we call *binary tree* a tree whose nodes have a maximum of two children. The letter $\mathcal{D} \subset \mathcal{T}$ will represent a finite list of data trees, sometimes treated as a set.

A tree t' is a *subtree* of a tree t (written $t' \preceq t$) if t' is a connected subgraph of t which contains the root of t (this is also known as *top-down subtree*). 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. The components form a set, not a sequence; therefore, permuting them does not give a different tree. In our drawings, we follow the convention that larger trees are drawn at the left of smaller trees. In the algorithms we will identify trees by strings in the following way, which will allow us to order trees lexicographically (the drawing convention corresponds to using the lexicographically least identification).

Definition 1. *We define the injective total function $\langle\cdot\rangle : \mathcal{T} \to \{0,1\}^*$ recursively as follows. If t is a single node, then $\langle t\rangle = 01$. Otherwise, suppose that $t_1, \ldots, t_k$ are the components of t enumerated so that $\langle t_1\rangle \leq \langle t_2\rangle \leq \ldots \leq \langle t_k\rangle$ (in lexicographical order). Then $\langle t\rangle = 0\langle t_1\rangle \ldots \langle t_k\rangle 1$. We also define $[\cdot] : \{0,1\}^* \to \mathcal{T}$ such that for any tree t, $[\langle t\rangle] = t$, and is undefined for strings not in the image of $\langle\cdot\rangle$.*

The one-node tree [01] will be represented with the symbol •, and the two-node tree [0011] by •–•.

Definition 2. *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; it is a maximum common subtree if there is no common subtree of larger size.*

Two trees have always some maximal common subtree but, as is shown in Figure 1, this common subtree does not need to be unique.

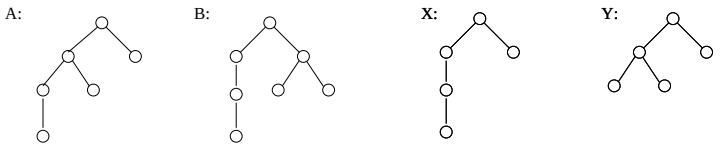


Fig. 1. Trees X and Y are maximal common subtrees of A and B.

In fact, trees X and Y have the maximum number of nodes among the common subtrees of A and B. As is shown in Figure 2, just a slight modification of A and B gives two maximal common subtrees of different sizes, showing that the concepts of maximal and maximum common subtree do not coincide in general.

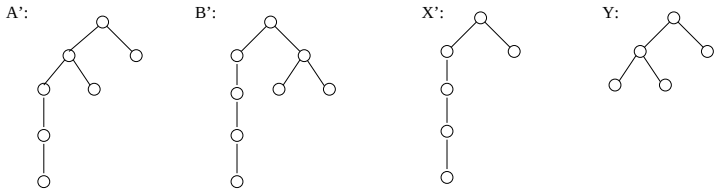


Fig. 2. Both X' and Y are maximal common subtrees of A' and B', but only X' is maximum.

2.2 Number of trees

The number of trees with n nodes is known to be $\Theta(\rho^n n^{-3/2})$, where $\rho = 0.3383218569$ ([9]). We provide a more modest lower bound based on an easy way to count the number of 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 binary trees with n nodes, and set $B_0 = 1$ for convenience. Clearly, a root without children is the only binary tree with one

node, so $B_1 = 1$. Now, B_n is the sum of all products $B_i B_j$ for every way to express $n-1$ as $i+j$ (meaning that the $n-1$ nodes other than the root are distributed into two subtrees having i and j nodes). So, we have

$$B_n = \sum_{\substack{i+j=n-1 \\ i \leq j}} B_i B_j = \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} B_i B_{n-i-1}.$$

The second summation can be rewritten as

$$B_n = B_0 B_{n-1} + \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor - 1} B_{i+1} B_{n-i-2} = B_{n-1} + \sum_{i=0}^{\lfloor \frac{n-3}{2} \rfloor} B_{i+1} B_{(n-2)-(i+1)-1}$$

which implies that $B_n \geq B_{n-1} + B_{n-2}$, 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

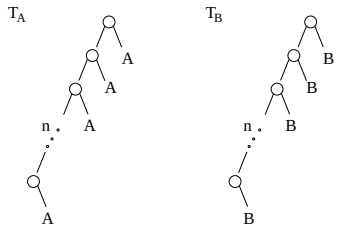
$$\phi^{n-2} \leq F_n \leq B_n.$$

which is also a lower bound for the total number of trees with n nodes.

2.3 Number of subtrees

We can easily observe, using the trees A, B, X, and Y of Section 2.1, 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.

3 Closure operator and mining closed trees

Once a proper notion of intersection is available, we move on to build a notion of closed sets of trees, with a view towards a data mining framework operating on tree-structured data.

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 an unordered 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.

The support of a tree t in $\mathcal{D}$ is the number of transactions where t is a subtree of the tree in the transaction. General usage would lead to the following notion of closed tree:

Definition 3. *A tree t is closed for $\mathcal{D}$ if no tree $t' \neq t$ exists with the same support such that $t \preceq t'$.*

Note 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.

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 [6], [3], [7], or [2] 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.

3.1 Galois Connection

A Galois connection is provided by two functions, relating two lattices in a certain way. Here our lattices 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 4. *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* $\mathcal{D}$, $\tau_{\mathcal{D}}(B) = \{t' \in \mathcal{D} \mid \forall\, t \in B\,(t \preceq t')\}$

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:*

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

Proof. By definition, each of the two sides is equivalent to

$$\forall\, t \in B\, \forall\, t' \in A\,(t \preceq t') \qquad \square$$

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

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

Thus, we have now a concept of closed sets of trees; 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, $\Gamma_{\mathcal{D}}(\{t\})$. 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 is maximal in $\Gamma_{\mathcal{D}}(\{t\})$ (that is, $\forall t' \in \Gamma_{\mathcal{D}}(\{t\})[t \preceq t' \Rightarrow t = t']$) 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 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, $\mathrm{supp}(t) \leq \mathrm{supp}(t')$. Hence, equality holds, and from the fact that t is closed, with $t \preceq t'$ and $\mathrm{supp}(t) = \mathrm{supp}(t')$, we infer $t = t'$. Thus, t is maximal in $\Gamma_{\mathcal{D}}(\{t\})$. $\square$

Now we can continue the argument as follows. Suppose t is maximal in some closed set of trees B. 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:

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

Yet another simpler observation 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 [3]). 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 [3] than our current development is. But there would be no indisputable advantages.

4 Intersection algorithms

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.

4.1 Finding the intersection recursively

We start with a straightforward algorithm for finding all maximal common subtrees of two trees in a recursive way. 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 represent the trees. The only condition needed is that every component t' of a tree t can be accessed with

```
RECURSIVE INTERSECTION(r, 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. 3. Algorithm RECURSIVE INTERSECTION

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 COMPONENT(t, i). The other procedures are as follows:

- #COMPONENTS(t) computes the number of components of t, this is, the arity of the root of t.
- MATCHINGS(n_1, n_2) computes the set of perfect matchings of the graph K_{n_1,n_2}, this 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,
  MATCHINGS$(2,3)$ $=$ $\{\{(1,1),(2,2)\},\ \{(1,1),(2,3)\},\ \{(1,2),(2,1)\},\ \{(1,2),(2,3)\},\ \{(1,3),(2,1)\},\ \{(1,3),(2,2)\}$.
- 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.
- MAX SUBTREES(S_1, S_2) returns the list of trees containing every tree in S_1 and every tree in S_2 that is not a subtree of another tree in S_1, thus leaving only the maximal subtrees. There is a further analysis of this procedure 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.

4.2 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 Figure 4 constructs a dictionary D accessed by pairs of trees (t_1, t_2) (or, more precisely, by their codes $(\langle t_1 \rangle, \langle t_2 \rangle)$), when the input trees are nontrival (different from $\bullet$ and $\bullet\!\!-\!\!\bullet$). 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.

```
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

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

```
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
```

Fig. 5. Algorithm MAX SUBTREES

information needed to compute an entry has already been computed in previous steps.

The procedure MAX SUBTREES, which appears in the penultimate step of the two intersection algorithms presented, is shown in Figure 5. The key point in the procedure MAX SUBTREES is the identification of subtrees made in steps 3 and 5. By standard algorithms, it can be decided whether $t_1 \preceq t_2$ in time $O(n_1 n_2^{1.5})$ ([11]), where n_1 and n_2 are the number of nodes of t_1 and t_2, respectively.

Finally, the table in Figure 6 shows an example of the intersections stored in the dictionary by the algorithm DYNAMIC PROGRAMMING INTERSECTION with trees A and B of Figure 1 as input.

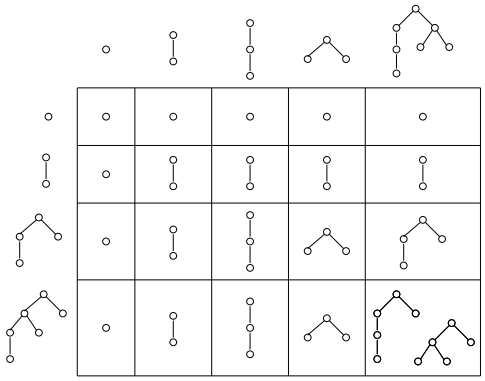


Fig. 6. Table with all partial results computed

5 Conclusion

Closure-based structures have been proposed in several references in a context of data mining. They may allow for summarizing the (huge) lattice of all the

subsets of a dataset by reducing it to only closure sets: these may add up to a much lesser quantity, and each closed subset of the dataset may offer some sort of actionable interpretation. We have studied such an approach to tree-like link structures.

Whereas we do not attempt at the design of specific algorithms here for computing closures yet, we have pointed out that the notion of closure given in the previous section does provide the appropriate framework for a closure-based data mining task on tree-structured data. Moreover, the properties established here suggest that it is possible to construct the lattice of closed sets of trees by mining first the closed trees and, then, organizing them into the desired lattice.

We describe 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 were not made up for the example, but were instead obtained through six different (rather arbitrary) random seeds of the synthetic tree mining generator of Zaki [12].

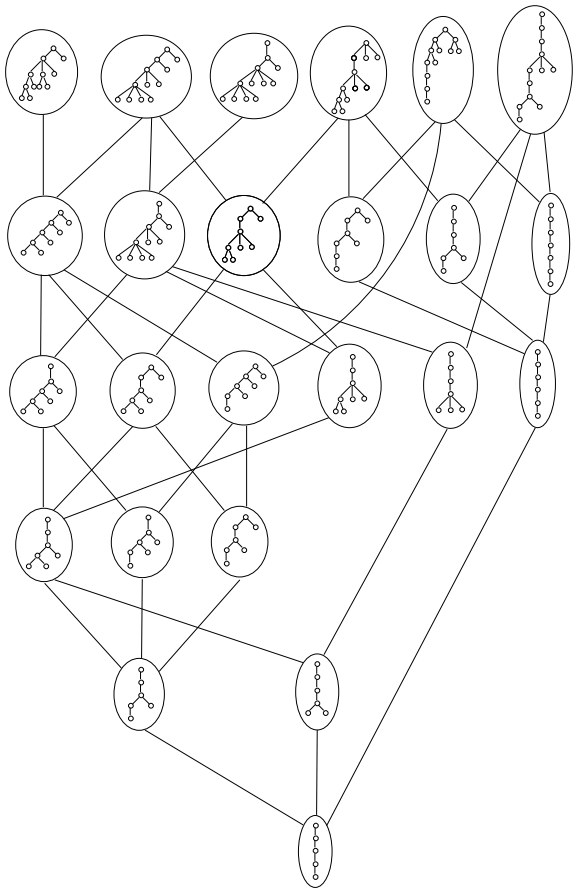


Fig. 7. Lattice of closed trees for the six input trees in the top row

The figure 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 intersections sets, which is possible as explained in a previous section, 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.

The study of algorithmics for the construction of this lattice, or of a fragment thereof (e.g. through frequency thresholds) since it will be usually quite large, will be subject of further work, as well as the corresponding notions of implications or association rules for the framework of unordered trees.

References

1. Hiroki Arimura and Takeaki Uno. An output-polyunomial time algorithm for mining frequent closed attribute trees. In *ILP*, pages 1–19, 2005.
2. Jaume Baixeries and José L. Balcázar. Discrete deterministic data mining as knowledge compilation. In *Workshop on Discrete Math. and Data Mining at SIAM DM Conference*, 2003.
3. José L. Balcázar and Gemma C. Garriga. On Horn axiomatizations for sequential data. In *ICDT*, pages 215–229 (extended version to appear in Theoretical Computer Science), 2005.
4. Yun Chi, Richard Muntz, Siegfried Nijssen, and Joost Kok. Frequent subtree mining – an overview. *Fundamenta Informaticae*, XXI:1001–1038, 2001.
5. Yun Chi, Yi Xia, Yirong Yang, and Richard Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17(2):190–202, 2005.
6. B. Ganter and R. Wille. *Formal Concept Analysis*. Springer-Verlag, 1999.
7. Gemma C. Garriga. Formal methods for mining structured objects. PhD Thesis, 2006.
8. Gemma C. Garriga and José L. Balcázar. Coproduct transformations on lattices of closed partial orders. In *ICGT*, pages 336–352, 2004.
9. J. M. Plotkin and John W. Rosenthal. How to obtain an asymptotic expansion of a sequence from an analytic identity satisfied by its generating function. *J. Austral. Math. Soc. (Series A)*, 56:131–143, 1994.
10. Alexandre Termier, Marie-Christine Rousset, and Michele Sebag. DRYADE: a new approach for discovering closed frequent trees in heterogeneous tree databases. In *ICDM*, pages 543–546, 2004.
11. Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.
12. Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.