Revisiting Pattern Structures for Structured Attribute Sets

Mehwish Alam[1], Aleksey Buzmakov[1], Amedeo Napoli[1], and Alibek Sailanbayev[2⋆]

[1]LORIA (CNRS – Inria NGE – U. de Lorraine), Vandœuvre-lès-Nancy, France
[2]Nazarbayev University, Astana, Kazakhstan
{ mehwish.alam, aleksey.buzmakov, amedeo.napoli, } @loria.fr,
alibek.sailanbayev@nu.edu.kz

Abstract. In this paper, we revisit an original proposition on pattern structures for structured sets of attributes. There are several reasons for carrying out this kind of research work. The original proposition does not give many details on the whole framework, and especially on the possible ways of implementing the similarity operation. There exists an alternative definition without any reference to pattern structures, and we would like to make a parallel between two points of view. Moreover we discuss an efficient implementation of the intersection operation in the corresponding pattern structure. Finally, we discovered that pattern structures for structured attribute sets are very well adapted to the classification and the analysis of RDF data. We terminate the paper by an experimental section where it is shown that the provided implementation of pattern structures for structured attribute sets is quite efficient.

Keywords: Formal Concept Analysis, Pattern Structures, Structured Attribute Sets, Least Common Ancestor, Range Minimum Query.

1 Introduction

In this paper, we want to make precise and develop a section of [1] related to pattern structures and structured sets of attributes. There are several reasons for carrying out this kind of research work. Firstly, the the pattern structures, the similarity operator $\sqcap$ and the associated subsumption operator $\sqsubseteq$ for structured sets of attributes are based on antichains and rather briefly sketched in the original paper. Secondly, there is an alternative and a more "qualitative" point of view on the same subject in [2, 3] without any reference to pattern structures, and we would like to make a parallel between these two points of view. Finally, for classifying RDF triples in the analysis of the content of Linked Open Data (LOD), we discovered that actually pattern structures for structured sets of attributes are very well adapted to solve this problem [4]. Moreover, the

⋆ This work was done during the stay of Alibek Sailanbayev at LORIA, France.

ISBN 978–2–9544948–0–7, ISSN 2311–701X, Blaise Pascal University, LIMOS laboratory, Clermont-Ferrand, 2015.

classification of RDF triples provides a very good and practical example for illustrating the use of such a pattern structure and helps to reconcile the two above points of view.

Accordingly, in this paper, we will go back to the two original definitions and show how they are related. For completing the history, it is worth mentioning that antichains, whose intersection is the basis of the similarity operation in the pattern structure for structured attribute sets, our paper, are studied in the book [5]. Moreover, this book cites as an application of antichain intersection an older paper from 1994 [6], written in French, about the decomposition of total orderings and its application to knowledge discovery.

Then, we proceed to present a way of efficiently working with antichains and intersection of antichains, which can be very useful, especially in case of large sets of data. The last section details a series of experiments where it is shown that pattern structures can be implemented with an efficient intersection operation and that they have a generally better behavior than scaled contexts.

2 Pattern Structures for Structured Attributes

2.1 Pattern Structures

Formal Concept Analysis [7] can process only binary contexts. Pattern structures are an extension of FCA which allow a direct processing of such kind of data. The formalism of pattern structures was introduced in [1].

A *pattern structure* is a triple $(G, (D, \sqcap), \delta)$, where G is the set of objects, $(D, \sqcap)$ is a meet-semilattice of descriptions, and $\delta : G \to D$ maps an object to its description. In other words, a pattern structure composed of a set of objects, a set of descriptions equipped with a similarity operation denoted by $\sqcap$. This similarity operation is idempotent, commutative and associative. If $(G, (D, \sqcap), \delta)$ is a pattern structures then the derivation operators (Galois connection) are defined as:

$$A^\diamond := \prod_{g \in A} \delta(g) \qquad \text{for } A \subseteq G$$

$$d^\diamond := \{g \in G | d \sqsubseteq \delta(g)\} \qquad \text{for } d \in D$$

Each element in D is referred to as a *pattern*. The natural order on $(D, \sqcap)$, given by $c \sqsubseteq d \Leftrightarrow c \sqcap d = c$ is called the subsumption order. Now a pattern concept can be defined as follows:

Definition 1 (Pattern Concept). *A pattern concept of a pattern structure $(G, (D, \sqcap), \delta)$ is a pair (A, d) where $A \subseteq G$ and $d \in D$ such that $A^\diamond = d$ and $A = d^\diamond$, where A is called the concept extent and d is called the concept intent.*

A pattern extent corresponds to the maximal set of objects A whose descriptions subsume the description d, where d is the maximal common description

for objects in A. The set of all pattern concepts is partially ordered w.r.t. inclusion on extents, i.e., $(A_1, d_1) \leq (A_2, d_2)$ iff $A_1 \subseteq A_2$ (or, equivalently, $d_2 \sqsubseteq d_1$), making a lattice, called pattern lattice.

2.2 Two original propositions on structured attribute sets

We briefly recall two original propositions supporting the present study. The first work is firstly published by Carpineto & Romano in [2] and then developed in [3]. The second work is related to the definition of pattern structures by Ganter & Kuznetsov in [1].

In [2, 3], the authors consider a formal context (G, M, I) and an extended set of attributes $M^* \supset M$ where attributes are organized within a subsumption hierarchy according to a partial ordering denoted by $\leq_{M^*}$. The following condition should be satisfied:

$\forall g \in G, m_1 \in M, m_2 \in M^* : [(g, m_1) \in I, m_1 \leq_{M^*} m_2] \Longrightarrow (g, m_2) \in I$

The subsumption hierarchy can be either a tree or an acyclic graph with a unique maximal element, as this is the case of attributes lying in a thesaurus for example. Then the building of a concept lattice from such a context can be done in two main ways. A first is to use a scaling and to complete the description of an object with all attributes implied by the original attributes. We discuss this scaling operation in detail later. The problem would be the space necessary to store the scaled context, especially in case of big data. A second way is to use an "extended intersection operation" between sets of attributes which is defined as follows. The intersection of two sets of attributes Y_1 and Y_2 is obtained by finding for each pair $(m_1, m_2), m_1 \in Y_1, m_2 \in Y_2$, the most specific attributes in M^* that are more general than m_1 and m_2, and then retaining only the most specific elements of the set of attributes generated in this way. Then if (X_1, Y_1) and (X_2, Y_2) are two concepts, we have:

$(X_1, Y_1) \leq (X_2, Y_2) \Longleftrightarrow \forall m_2 \in Y_2, \exists m_1 \in Y_1, m_1 \leq_{M^*} m_2$

In other words, this intersection operation corresponds to the intersection of two antichains as this is explained in [1], where the authors define the formalism of pattern structures and take as an instantiation structured attribute sets. More formally, it is assumed that the attribute set $(M, \leq_M)$ is finite and partially ordered, and that all attribute combinations that can occur must be order ideals (downsets) of this order. Then, any order ideal O can be described by the set of its maximal elements; $O = \{x | \exists y \in M,\ x \leq y\}$. *It should be noticed that the order considered on the attribute sets in [1] is reversed with respect to the order considered in [2, 3].* However, we keep the original definitions used in [1] in the present paragraph. These maximal elements form an antichain, and conversely, each antichain is the set of maximal elements of some order ideal. Thus, the semilattice $(D, \sqcap)$ of patterns in the pattern structure consists of all antichains of the ordered attribute set. In addition, it is isomorphic to the lattice of all order ideals of the ordered set, and thus isomorphic to the concept lattice of the context $(P, P, \not\geq)$. For two antichains AC_1 and AC_2, the infimum $AC_1 \sqcap AC_2$ consists of all maximal elements of the order ideal:

$\{m \in P \mid \exists ac_1 \in AC_1,\ \exists ac_2 \in AC_2,\ m \leq ac_1 \text{ and } m \leq ac_2\}$.

There is a "canonical representation context" (or an associated scaling operator) for the pattern structure $(G, (D, \sqcap), \delta)$ related to structured attribute sets, which is defined by the set of "principal ideals $\downarrow p$" as follows: (G, P, I) with $(g, p) \in I \Longleftrightarrow p \leq \delta(g)$.

In the next section, we make precise and discuss the pattern structure for structured attribute sets by taking the point of view of filters and not of ideals in agreement with the order from [2, 3], with the most general attributes above.

2.3 From Structured Attributes to Tree-shaped Attributes

An important case of structured attributes is "tree-shaped attributes", i.e., when the attributes are organized within a partial order corresponding to a rooted tree. If it is the case, then the root of the tree, denoted by $\top$, can be matched against the description of any object, while the leaves of this tree are the most detailed descriptions. For example, the root can correspond to the attribute 'Animal' and a leaf can correspond to the attribute 'Cat'; somewhere in between there could be attribute 'Mammal'.

An example of such kind of data naturally appears in the domain of semantic web data. For example, Figure 1 gives a small part of ACCS[1]. This attribute tree will be used as a running example and should be read as follows. If an object belongs to class C_1 (and probably to some other classes), then it necessarily belongs to classes C_{10}, C_{12}, and $\top$, e.g., if an object is a cat, then it is a mammal and an animal. Accordingly, the description of an object can include several classes, e.g., classes C_1, C_5 and C_8. Thus, some of the tree-shaped attributes can be omitted from the description of an object. However, they should be always taken into account when computing the intersection between descriptions. Thus, in order to avoid redundancy in the descriptions, we can allow only antichains of the tree as possible elements in the set D of descriptions, and then, accordingly compute the intersection of antichains.

An efficient way of computing intersection of antichains is explained in the next section. Here it is important to notice that although it is a hard task to efficiently compute intersection of antichains in an arbitrary partial order of attributes, the intersection of antichains in a tree can help in computing this more general intersection. Indeed, in a partial order of attributes, we can add an artificial attribute $\top$ that can be matched against any description. Then, instead of considering an intersection of antichains in an arbitrary poset we can take a spanning tree of it with $\top$ taken as the root. Although we have lost some relations between attributes, and, thus, the size of the antichains is probably larger, we can apply the efficient intersection of antichains of tree discussed below.

2.4 On Computing Intersection of Antichains in a Tree

In this subsection we show how to efficiently solve the problem of intersection of antichains in a tree. The problem is formalized as follows. A partial order is

[1] https://www.acm.org/about/class/2012

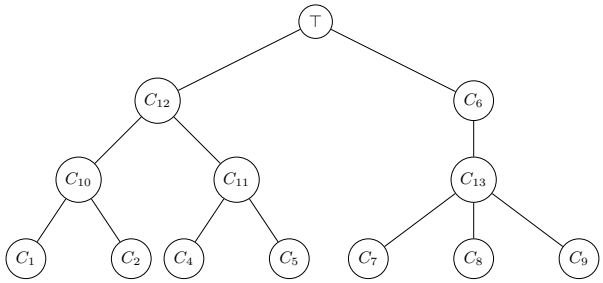


Fig. 1: A small part from ACM Computing Classification System (ACCS).

described by the Hasse diagram corresponding to the tree. The root is denoted by $\top$ and it is larger w.r.t. the partial order than any other element in the tree. Given a rooted tree $\mathcal{T}$ and two antichains X and Y, we should find an antichain Z such that (1) for all $x \in X \cup Y$ there is $z \in Z$ such that $x \leq z$ and (2) no $z \in Z$ can be removed or changed to $\tilde{z} < z$ without violating requirement (1).

If the cardinality of antichains X and Y is 1 then this task is reduced to the well-known problem of a Least Common Ancestor (LCA). In 1984 it was already shown that the LCA problem can be reduced to Range Minimum Query (RMQ) problem [8]. Later several simpler approaches were introduced for solving the LCA problem. Here we briefly introduce the reduction of LCA to RMQ in accordance with [9].

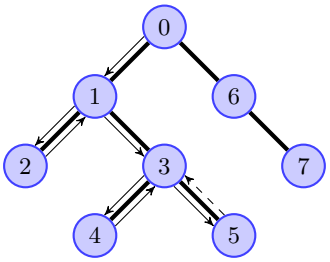
Reduction of LCA to RMQ. Given an array of numbers, the RMQ problem consists in efficient answering queries on the position of the minimal value in a given range (interval) of positions for this array. For example, given an array

Array [2 1 0 3 2]
Positions 1 2 3 4 5

where the first value is in position 1 and the last value is in position 5, the answer to the query on the position of the minimal number in the range 2–4, i.e., the corresponding part of array is [1;0;3], is 3 (the value of the 3rd element in the array is 0 and it is the minimal value in this range). Accordingly, the position of the minimal number in the range 1–2 (the part of the array is [2;1]) is 2. The good point about this problem is that it can be solved in $O(n)$ preprocessing computational time and in $O(1)$ computational time per one query [9], where n is the number of elements in the array.

In order to introduce the reduction of LCA to RMQ we need to know what is the depth of a tree vertex. The *depth* of a vertex in a rooted tree is the number of edges in the shortest path from that vertex to the root of the tree.

We create the array of depths of the vertices in the tree that is used as an input array for RMQ. We build this array in the following way. We traverse the tree in depth first order (see Figure 2). Every time the algorithm considers a



Depth array $\mathcal{D}$	[	0	1	2	1	**2**	3	**2**	3	**2**	1	0	**1**	2	**1**	0]
Corresponding vertex		v0	v1	v2	v1	**v3**	v4	**v3**	v5	**v3**	v1	v0	**v6**	v7	**v6**	v0
Positions		1	2	3	4	**5**	6	**7**	8	**9**	10	11	**12**	13	**14**	15

Fig. 2: Reducing RMQ task to LCA. Arrows show the depth first order traversal. The depth array $\mathcal{D}$ is accompanied by the corresponding vertices and positions.

vertex, i.e., the first visit or a return to the vertex, we should put the depth of that vertex at the end of the depth array $\mathcal{D}$. We also keep track of a vertex corresponding to each depth in $\mathcal{D}$. The depth array $\mathcal{D}$ has $2|\mathcal{T}|-1$ values, where $|\mathcal{T}|$ is the number of vertices in the tree.

Now for any value in $\mathcal{D}$ we know the corresponding vertex of the tree and any vertex of the tree is associated with several positions in $\mathcal{D}$. For example, in Figure 2 the value in the first position of $\mathcal{D}$, i.e., $\mathcal{D}[1]$, is 0, corresponding to the root of the tree. If we take vertex 3, then the associated values of $\mathcal{D}$ are on positions 5, 7, and 9.

Given two vertices $A, B \in \mathcal{T}$, let a be one of the positions in $\mathcal{D}$ corresponding to vertex A, let b be one of the positions in $\mathcal{D}$ corresponding to B. Then it can be shown that the vertex corresponding to the minimal value in $\mathcal{D}$ in the range a–b is the least common ancestor of A and B. For example, to find LCA between vertices 3 and 6 in Figure 2, one should first take two positions in $\mathcal{D}$ corresponding to vertices 3 and 6. Positions 5,7, and 9 in array $\mathcal{D}$ correspond to vertex 3, positions 12 and 14 correspond to vertex 6. Thus, we can query RMQ for ranges 5–14, 7–14, 7–12, etc. The minimal value in $\mathcal{D}$ for all these ranges is 0 located at position 11 in $\mathcal{D}$, i.e., $\texttt{RMQ}(5, 14) = 11$. Thus, the vertex corresponding to position 11, i.e., vertex 0, is the least common ancestor for vertices 3 and 6.

Let us notice that if $A \in \mathcal{T}$ is an ancestor of $B \in \mathcal{T}$ and a and b are two positions corresponding to the vertices A and B, then the position $\texttt{RMQ}(a, b)$ in $\mathcal{D}$ always corresponds to the vertex A, in most of the cases $\texttt{RMQ}(a, b) = a$. Thus we are also able to check if a vertex of $\mathcal{T}$ is an ancestor of another vertex of $\mathcal{T}$.

Now we know how to solve the LCA problem in $O(|\mathcal{T}|)$ preprocessing computational time and $O(1)$ computational time per query. Let us return to the problem of intersecting antichains of a tree.

Antichain intersection problem. Let us first discuss the naive approach to this problem. Given two antichains $A, B \subset \mathcal{T}$, one can compute the set

$\mathcal{D}$	[0	1	2	3	2	3	2	1	2	3	2	3	2	1	0	1	2	3	2	3	2	3	2	1	0]
	$\top$	C_{12}	C_{10}	C_1	C_{10}	C_2	C_{10}	C_{12}	C_{11}	C_4	C_{11}	C_5	C_{11}	C_{12}	$\top$	C_6	C_{13}	C_7	C_{13}	C_8	C_{13}	C_9	C_{13}	C_6	$\top$
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Fig. 3: Depth array, the corresponding vertices, and indices for the tree in Figure 1.

$\{\texttt{LCA}(a,b) \mid \forall a \in A \text{ and } \forall b \in B\}$. Then this set should be filtered for removing the comparable elements in order to get an antichain. It is easy to see that the result is the intersection of A and B but it requires at least $|A|\cdot|B|$ operations.

Let us reformulate this naive approach in terms of RMQ. Given a depth array $\mathcal{D}$ and two sets of indices $A, B \subseteq \mathbb{N}_{|\mathcal{D}|}$ forming an antichain, we should compute the set $Z = \{\texttt{RMQ}(a,b) \mid \forall a \in A \text{ and } \forall b \in B\}$ and then remove all elements $z \in Z$ such that there is $x \in Z \setminus \{z\}$ with the position $\texttt{RMQ}(z,x)$ corresponding to the same vertex as z, i.e., elements z corresponding to an ancestor of another element from Z.

Let us consider for example the tree $\mathcal{T}$ given in Figure 1. Figure 3 shows the depth array, the corresponding vertices, and indices of this array. Let us show how to compute the intersection of $A = \{C_1, C_5, C_8\}$ and $B = \{C_1, C_7, C_9\}$. The expected result is $\{C_1, C_{13}\}$. First we translate the sets A and B to the indices in array $\mathcal{D}$ for RMQ, i.e., $A = \{4, 12, 20\}$ and $B = \{4, 18, 22\}$. Then we compute RMQ for all pairs from A and B:

$$\{\texttt{RMQ}(4,4) = 4, \texttt{RMQ}(4,18) = 15, \texttt{RMQ}(4,22) = 15, \cdots, \texttt{RMQ}(20,18) = 19, \cdots\}.$$

Now we should remove positions corresponding to ancestors in the tree, e.g., $\texttt{RMQ}(4,15) = 15$ and, hence, 15 should be removed. The result is $\{4, 13\}$ representing exactly $\{C_1, C_{13}\}$.

Let us discuss two points that help us to reduce the complexity of the naive approach. Consider the positions $i \leq l \leq m \leq j$ and $k = \texttt{RMQ}(i,j)$, $n = \texttt{RMQ}(l,m)$. Then the depth in the position k is not larger than the depth in the position n, $\mathcal{D}[k] \leq \mathcal{D}[n]$. Hence the position $\texttt{RMQ}(k,n)$ corresponds to the same vertex as position k. For example, in Figure 3 $\texttt{RMQ}(4,6) = 5$ and $\texttt{RMQ}(2,7) = 2$. The value in position 5 in the array $\mathcal{D}$ is $\mathcal{D}[5] = 2$. It is larger than the value in position 2, $\mathcal{D}[2] = 1$. Thus, the value in position returned by RMQ for the larger range is smaller than the value in position returned by RMQ for the smaller range.

Thus, given two sets of indices $A, B \subseteq \mathbb{N}_{|\mathcal{D}|}$ corresponding to antichains, we can modify the naive algorithm by ordering the set $A \cup B$ and computing RMQ only for consecutive elements from different sets, rather then for all pairs from different sets. For example, for intersecting $A = \{4, 12, 20\}$ and $B = \{4, 18, 22\}$, we join them to the set $Z = \{4_A, 4_B, 12_A, 18_B, 20_A, 22_B\}$. Then, we compute RMQ only for consecutive elements from different sets, i.e., $\texttt{RMQ}(4,4) = 4$, $\texttt{RMQ}(4,12) = 8$, $\texttt{RMQ}(12,18) = 15$, $\texttt{RMQ}(18,20) = 19$, and $\texttt{RMQ}(20,22) = 21$. The cardinality of $A \cup B$ is less then $|A| + |B|$, hence, the number of the consecutive elements is $O(|A|+|B|)$, and, thus, the number of RMQs of consecutive elements is $O(|A| + |B|)$.

However, the set Z of RMQs of consecutive elements does not not necessarily correspond to an antichain in $\mathcal{T}$. Thus we should filter this set, in order to remove all ancestors of another elements form Z. Accordingly, it is clear that to filter the set Z it is enough to check only consecutive elements of Z. For example, the intersection of $A = \{4, 12, 20\}$ and $B = \{4, 18, 22\}$ gives us the following set $Z = \{4, 8, 15, 19, 21\}$. Let us now check the RMQs of consecutive elements. $\texttt{RMQ}(4, 8) = 8$, thus, 8 is an ancestor of 4 and 8 can be removed. Since 8 is removed, we compare $\texttt{RMQ}(4, 15) = 15$, thus, 15 should be also removed. Then we compute $\texttt{RMQ}(4, 19) = 15$, i.e., the indices 4 and 19 are not ancestors and both are kept. Now we compute $\texttt{RMQ}(19, 21) = 19$ and, thus, 19 should be removed (actually positions 19 and 21 correspond to the same vertex C_{13} and one of them should be removed). Thus, the result of intersecting A and B is $\{4, 21\}$ corresponding to the antichain $\{C_1, C_{13}\}$.

Since the number of elements in the set Z is $O(|A| + |B|)$, then overall complexity of computing intersection for two antichains $A, B \subset \mathcal{T}$ of a tree $\mathcal{T}$ is $O(|A| + |B|)$ or, taking into account that the cardinality of an antichain in a tree is less then the number of leaves (vertices having no descendants) in this tree, the complexity of computing intersection of two antichains is $O(|\texttt{Leaves}(\mathcal{T})|)$.

Antichain intersection by scaling. An equivalent approach for computing intersection of antichains is to scale the antichains to the corresponding filters. A *filter corresponding to an antichain* in a poset is the set of all elements of the poset that are larger then at least one element from the antichain. For example, let us consider a tree-shaped poset in Figure 1. A filter corresponding to the antichain $\{C_1, C_5, C_8\}$ is the set of all ancestors of all elements from the antichain, i.e., it is equal to $\{C_1, C_{10}, C_{12}, \top, C_5, C_{11}, C_8, C_{13}, C_6\}$.

The set-intersection of filters corresponding to the given antichains is a filter corresponding to the antichain resulting from intersection of the antichains. However this approach has a higher complexity. Indeed, the size of a filter is $O(|\mathcal{T}|)$ and, thus, the computational complexity of intersecting two antichains by means of a scaling is $O(|\mathcal{T}|)$ which is harder then $O(|\texttt{Leaves}(\mathcal{T})|)$ for intersecting antichains directly. Indeed, the number of leaves in a tree can be dramatically smaller than the number of vertices in this tree. For example, the number of vertices in Figure 1 is 13, while the number of leaves is only 7. Thus, the direct intersection of antichains is more efficient than the intersection by means of a scaling procedure.

Relation to intersection of antichains in partially ordered sets of attributes. As it was mentioned in the previous section, the intersection of antichains in arbitrary posets can be reduced to the intersection of antichains in a tree. However, the size of the antichain representing a description of an object can increase. Indeed, since we have reduced a poset to a tree, some relations have been lost, and thus the attributes that are subsumed in the poset for a given antichain A are no more subsumed in the tree for A, and hence should be added to A. However, the reduction is still more computationally efficient than

Table 1: Results of the experiments with different kind of data.

#objects is the number of objects in the corresponding dataset. #attributes is the number of numerical attributes before scaling. $|G|$ is the number of objects used for building the lattice. $|\mathcal{T}|$ is the size of the attribute tree and the number of attributes in the scaled context $|M|$. $\texttt{Leaves}(\mathcal{T})$ is the number of leaves in the attribute tree. $|\mathcal{L}|$ is the size of the concept lattice for the corresponding data. $t_{\mathcal{T}}$ is the computational time for data represented as a set of antichains in the attribute tree. $t_{\mathbb{K}}$ is the computational time represented by a scaled context, i.e., by a set of filters in the attribute tree; '*' shows that the we are not able to build the whole lattice. t_{num} is the computational time for numerical data represented by an interval pattern structure.

(a) Real data experiments.

Dataset	$\lvert G\rvert$	$\lvert\mathcal{T}\rvert$	$\texttt{Leaves}(\mathcal{T})$	$\lvert\mathcal{L}\rvert$	$t_{\mathcal{T}}$	$t_{\mathbb{K}}$
DBLP	5293	33207	33198	10134	45 sec	21 sec
Biomedical Data	63	1490	933	1725582	145 sec	162 sec

(b) Numerical data experiments.

Dataset	#objects	#attributes	$\lvert G\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert\texttt{Leaves}(\mathcal{T})\rvert$	$\lvert\mathcal{L}\rvert$	$t_{\mathcal{T}}$	$t_{\mathbb{K}}$	t_{num}
BK	96	5	35	626	10	840897	37 sec	42 sec*	19 sec
LO	16	7	16	224	26	1875	0.043 sec	0.088 sec	0.024 sec
NT	131	3	131	140	6	128624	3.6 sec	6.8 sec	3.1 sec
PO	60	16	22	1236	58	416837	49 sec	57 sec*	10.7 sec
PT	5000	49	22	4084	60	452316	50 sec	38 sec*	15 sec
PW	200	11	94	436	21	1148656	60 sec	49 sec*	48 sec
PY	74	28	36	340	53	771569	46 sec	40 sec*	21 sec
QU	2178	4	44	8212	8	783013	28 sec	30 sec*	15.4 sec
TZ	186	61	31	626	88	650041	58 sec	43 sec*	22 sec
VY	52	4	52	202	15	202666	5.9 sec	11.6 sec	3 sec

computing the intersection of antichains in a poset by means of a scaling as it is discussed in the previous paragraph. However, for the reduction it could be interesting to find the spanning tree with the minimal number of leaves. Unfortunately, this is an NP-complete task and it thus cannot be applied for increasing the computational efficiency [10]. We should notice here that there is some work that solves the LCA problem for more general cases, e.g., lattices [11] or partially ordered sets [9]. However, it is an open question whether these works can help to efficiently compute intersection of antichains in the corresponding structures.

3 Experiments and Discussion

Several experiments are conducted using publicly available data on a MacBook with a 1.3GHz Intel Core i5, 4GB of RAM running OS X Yosemite 10.3. We have used `FCAPS`[2] software developed in C++ for dealing with different kinds of pattern structures. It can build a concept lattice starting from a standard formal context or from object descriptions given as antichains of a given tree. The last one is based on the similarity operation that is discussed above.

We performed our experiments on two datasets from different domains namely DBLP and biomedical data. In these datasets, object descriptions are given as subsets of attributes. A taxonomy of the attributes is already known based on domain knowledge. We compute a concept lattice in two different ways. In the first one, we directly compute the concept lattice from the antichains in a taxonomy. In the second one we scale every description to the corresponding filter of the taxonomy. After this we do not rely on the taxonomy and process the scaled context with standard FCA.

The first data set is DBLP, from which we extracted a subset of papers with their keywords published in conferences in Machine Learning domain. The taxonomy used for classifying such kind of triples is ACM Computing Classification System (ACCS)[3].

The second data set belongs to the domain of life sciences. It contains information about drugs, their side effects (SIDER[4]), and their categories (DrugBank[5]). The taxonomies related to this dataset are MedDRA [6] for side effects and MeSH[7] for drug categories.

The parameters of the datasets and the computational results are shown in Table 1a. It can be noticed that for DBLP the context consists of 5293 objects and 33207 attributes, in the taxonomy of the attributes we have 33198 leaves meaning that most of attributes are mutually incomparable. It took 45 seconds to produce a lattice having 10134 concepts directly from the descriptions given by antichains of the taxonomy. To produce the same lattice starting from a scaled context the program only takes 21 seconds. However, if we consider the biomedical data, the approach based on antichains is better. Indeed, it takes 145 seconds, while the computation starting from the scaled contexts takes 162 seconds. In this case, the dataset contains 1490 attributes with 933 leaves. Thus, the direct approach works faster if the number of leaves is significantly smaller than the number of vertices. It is worth noticing that the size of antichains is significantly smaller than the size of the filters, and thus our approach is more efficient. However, when the number of leaves is comparable to the number of vertices, our approach is slower. Although in this case our approach has the same

[2] https://github.com/AlekseyBuzmakov/FCAPS
[3] https://www.acm.org/about/class/2012
[4] `http://sideeffects.embl.de/`
[5] `http://www.drugbank.ca/`
[6] http://meddra.org/
[7] http://www.ncbi.nlm.nih.gov/mesh/

computational complexity as the scaling approach, the antichain intersection problem requires more efforts than the set intersection.

Since the efficiency of the antichain approach is high for the trees with a low number of leaves, we can use this method to increase efficiency of standard FCA for special kind of contexts. In a context (G, M, I) an attribute m_1 can be considered as an ancestor of another attribute m_2 if any object containing the attribute m_2 also contains the attribute m_1. Accordingly we can construct an attribute tree $\mathcal{T}$ and rely on it for computing intersection operation. In this case the set of attributes M and the set of vertices of $\mathcal{T}$ are the same and $|M| = |\mathcal{T}|$. The second part of the experiment was based on this observation.

We used numerical data from Bilkent University in the second part of the experiments[8]. It was converted to formal contexts by the standard interodinal scaling. The scaled attributes are closely connected, i.e., there are a lot of pairs of attributes (m_1, m_2) such that the set of objects described by m_1 is a subset of objects described by m_2, i.e., $(m_1)' \subseteq (m_2)'$. Thus, we can say that $m_1 \leq m_2$. Using this property we built attribute trees from the scaled contexts. These trees have many more vertices than leaves, thus, the approach introduced in this paper should be efficient. We compare our approach with the scaling approach. Moreover, recently, it was shown that interval pattern structures (IPS) can be efficiently used to process such kind of data [12]. Accordingly we also compared our approach with IPS.

The results are shown in Table 1b. Compared to Table 1a it has several additional columns. First of all, since for numerical data we typically got large lattices, in most of the cases we considered only part of the objects. The actual number of used objects is given in the column $|G|$, while the total size of the dataset is given in the column '#objects', e.g., `BK` dataset contained 96 objects, while we have used only 35. In addition for every dataset we also provide the number of the numerical attributes, e.g., `BK` has 5 numerical attributes. We should notice that when we built the lattice from some datasets by standard FCA, the lattice was so large that the memory was swapping and we stopped the computation. It was not the case for our approach since antichains requires less memory to store than the corresponding filters. The fact of swapping is shown by '*' next to computational time in column $t_{\mathbb{K}}$. In addition we also show the time for IPS to process the same dataset. For example, the processing of `BK` dataset took 37 seconds by our approach, took more than 42 seconds by standard FCA and memory had started swapping, and took 19 seconds by IPS.

This experiment shows that our approach takes not only less time to compute concept lattice, but also requires less memory, since there is no memory swapping. We can also see that the computation time for IPS is smaller than for our approach. However, IPS is only applicable for numerical data, while our approach can be applied for all cases when attributes of a context are structured. For example, we can deal with graph data scaled to the set of frequent subgraphs where many such attributes are subgraphs of other attributes.

[8] `http://funapp.cs.bilkent.edu.tr/DataSets/`

4 Conclusion

In this paper we recalled two approaches for dealing with structured attributes and explained how we can compute intersection of antichains in tree-shaped posets of attributes, an essential operation for working with structured attributes. Our experiments showed the computational efficiency of the proposed approach. Accordingly, we are interested in applying our approach to other kinds of data such as graph data. Moreover, the generalization of our approach to other kinds of posets is also of high interest.

References

1. Ganter, B., Kuznetsov, S.O.: Pattern structures and their projections. In: ICCS. LNCS 2120, Springer (2001) 129–142
2. Carpineto, C., Romano, G.: A lattice conceptual clustering system and its application to browsing retrieval. Machine Learning **24**(2) (1996) 95–122
3. Carpineto, C., Romano, G.: Concept Data Analysis: Theory and Applications. John Wiley & Sons, Chichester, UK (2004)
4. Alam, M., Napoli, A.: Interactive exploration over RDF data using Formal Concept Analysis. In: International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19 - October 21, 2015, IEEE (2015)
5. Caspard, N., Leclerc, B., Monjardet, B.: Finite Ordered Sets. Cambridge University Press, Cambridge, UK (2012) First published in French as "Ensembles ordonnés finis : concepts, résultats et usages", Springer 2009.
6. Pichon, E., Lenca, P., Guillet, F., Wang, J.W.: Un algorithme de partition d'un produit direct d'ordres totaux en un nombre fini de chaînes. Mathématiques, Informatique et Sciences Humaines **125** (1994) 5–15
7. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Berlin/Heidelberg (1999)
8. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and Related Techniques for Geometry Problems. In: Proc. Sixt. Annu. ACM Symp. Theory Comput. STOC '84, New York, NY, USA, ACM (1984) 135–143
9. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and DAGs. J. Algorithms **57**(2) (2005) 75–94
10. Salamon, G., Wiener, G.: On finding spanning trees with few leaves. Inf. Process. Lett. **105**(5) (2008) 164–169
11. Aït-Kaci, H., Boyer, R., Lincoln, P., Nasr, R.: Efficient Implementation of Lattice Operations. ACM Trans. Program. Lang. Syst. **11**(1) (January 1989) 115–146
12. Kaytoue, M., Kuznetsov, S.O., Napoli, A., Duplessis, S.: Mining gene expression data with pattern structures in formal concept analysis. Inf. Sci. (Ny). **181**(10) (2011) 1989 – 2001