Information Sciences 185 (2012) 114–127

Contents lists available at SciVerse ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data

Jan Outrata *, Vilem Vychodil

Dept. Computer Science, Palacky University, 771 46 Olomouc, Czech Republic

ARTICLE INFO

Article history:
Received 8 November 2010
Received in revised form 14 May 2011
Accepted 19 September 2011
Available online 24 September 2011

Keywords:
Galois connection
Object-attribute data
Formal concept analysis
Frequent itemset mining

ABSTRACT

Fixpoints of Galois connections induced by object-attribute data tables represent important patterns that can be found in relational data. Such patterns are used in several data mining disciplines including formal concept analysis, frequent itemset and association rule mining, and Boolean factor analysis. In this paper we propose efficient algorithm for listing all fixpoints of Galois connections induced by object-attribute data. The algorithm, called FCbO, results as a modification of Kuznetsov's CbO in which we use more efficient canonicity test. We describe the algorithm, prove its correctness, discuss efficiency issues, and present an experimental evaluation of its performance and comparison with other algorithms.

1. Introduction and Preliminaries

This paper describes a new algorithm for computing fixpoints of Galois connections. In particular, we focus on Galois connections [5,12,26,33] that appear in formal concept analysis (FCA) – a method of qualitative analysis of object-attribute relational data [10,33]. In a broader sense, the algorithm belongs to an important family of algorithms for listing combinatorial structures [11] and algorithms for biclustering [3,29]. The algorithm we propose is a refinement of Kuznetsov's [19,21] Close-by-One algorithm (CbO) in which we improve the canonicity test. The improvement significantly reduces the number of fixpoints which are computed multiple times, resulting in an algorithm that is considerably faster than the original CbO.

Recall that an antitone Galois connection between nonempty sets X and Y is a pair $\langle f,g\rangle$ of maps $f: 2^X \to 2^Y$ and $g: 2^Y \to 2^X$ satisfying, for any $A, A_1, A_2 \subseteq X$ and $B, B_1, B_2 \subseteq Y$,

$$A \subseteq g(f(A)), \tag{1}$$
$$B \subseteq f(g(B)), \tag{2}$$
$$\text{if } A_1 \subseteq A_2 \text{ then } f(A_2) \subseteq f(A_1), \tag{3}$$
$$\text{if } B_1 \subseteq B_2 \text{ then } g(B_2) \subseteq g(B_1). \tag{4}$$

The composed maps $f \circ g : 2^X \to 2^X$ and $g \circ f : 2^Y \to 2^Y$ are closure operators in 2^X and 2^Y, respectively [10,12]. A pair $\langle A,B\rangle \in 2^X \times 2^Y$ is called a fixpoint of $\langle f,g\rangle$ if $f(A) = B$ and $g(B) = A$. Since we are interested in listing all fixed points of $\langle f,g\rangle$, we restrict ourselves to finite X and Y.

Galois connections appear as induced structures in data analysis. Namely, suppose that X and Y are sets of objects and attributes/features, respectively, and let $I \subseteq X \times Y$ be an incidence relation, $\langle x,y\rangle \in I$ saying that object $x \in X$ has attribute

* Corresponding author.
E-mail addresses: jan.outrata@upol.cz (J. Outrata), vychodil@acm.org (V. Vychodil).

0020-0255/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2011.09.023

$y \in Y$. In FCA, the triplet $\langle X, Y, I\rangle$ is called a formal context and represents the input object-attribute data. Given $I \subseteq X \times Y$, we introduce two concept-forming operators [10] $^{\uparrow_I} : 2^X \to 2^Y$ and $^{\downarrow_I} : 2^Y \to 2^X$ defined, for each $A \subseteq X$ and $B \subseteq Y$, by

$$A^{\uparrow_I} = \{y \in Y | \text{ for each } x \in A : \langle x, y\rangle \in I\}, \tag{5}$$

$$B^{\downarrow_I} = \{x \in X | \text{ for each } y \in B : \langle x, y\rangle \in I\}. \tag{6}$$

By definition (5), $A^{\uparrow_I}$ is the set of all attributes shared by all objects from A and, by (6), $B^{\downarrow_I}$ is the set of all objects sharing all attributes from B. It is easily seen that $\langle {}^{\uparrow_I}, {}^{\downarrow_I}\rangle$ is a Galois connection between X and Y and it shall be called a Galois connection induced by I. The fixpoints of $\langle {}^{\uparrow_I}, {}^{\downarrow_I}\rangle$ are called formal concepts in I [10,12]. Formal concepts represent basic patterns that can be found in I and that have two common interpretations: (i) a geometric one: formal concepts are maximal rectangular subsets of I; (ii) a conceptual one: each formal concept $\langle A, B\rangle$ represents a concept in data with an extent A (objects that fall under the concept) and an intent B (attributes covered by the concept) such that A is a set of objects sharing all attributes from B and B is the set of all attributes shared by all objects from A. The latter interpretation of concepts is inspired by a traditional understanding of concepts as notions having their extent and intent which goes back to traditional Port-Royal logic [8,23].

In this paper, we propose an algorithm that lists all formal concepts in I, each of them exactly once. In the past, there have been proposed various algorithms for solving this task, see [22] for a survey and comparison. One of the main issues solved by all the algorithms is how to prevent listing the same formal concept multiple times. There are several approaches to cope with the problem. For instance, Lindig's algorithm [24] stores found concepts in a data structure (a particular search tree) and uses the data structure to check whether a formal concept has already been found. On the other hand, Ganter's NextClosure [9], CbO [19,21], and the algorithm proposed by Norris [30] use canonicity tests: formal concepts are supposed to be listed in certain order. The fact whether two consecutive concepts are listed in the order is ensured by a canonicity test. If a newly computed formal concept does not pass the canonicity test, it is not further considered. Hence, the canonicity test ensures that even if a formal concept is computed several times, it is listed exactly once. Conceptually, our algorithm can be seen as an improved version of CbO [19,21] in which we modify the canonicity test. The improvement significantly reduces the number of formal concepts which are computed multiple times. The reduction has a great impact on the performance of the algorithm because computing formal concepts using the closures $A^{\uparrow_I\downarrow_I}$ or $B^{\downarrow_I\uparrow_I}$ of a set of objects A or a set of attributes B, respectively, is the most critical operation. Note that other promising approaches related to CbO have been introduced in [27] and recently in [2].

Let us stress the importance of listing formal concepts. First, formal concepts are the basic output of formal concept analysis. If we denote by $\mathcal{B}(X, Y, I)$ the set of all formal concepts in $I \subseteq X \times Y$, we can define a partial order $\leqslant$ on $\mathcal{B}(X, Y, I)$ as follows:

$$\langle A_1, B_1\rangle \leqslant \langle A_2, B_2\rangle \text{ iff } A_1 \subseteq A_2 \text{ (or, equivalently, iff } B_2 \subseteq B_1). \tag{7}$$

If $\langle A_1, B_1\rangle \leqslant \langle A_2, B_2\rangle$ then $\langle A_1, B_1\rangle$ is called a subconcept of $\langle A_2, B_2\rangle$. The set $\mathcal{B}(X, Y, I)$ together with $\leqslant$is called a concept lattice [33]. A concept lattice is a complete lattice whose structure is described by the Basic Theorem of Concept Lattices [10]. The concept lattice is a formalization of a hierarchy of concepts that are found in the input data represented by I. FCA has been applied in many disciplines to analyze object-attribute data including program analysis and software engineering [31,32] and evaluation of questionnaires [6]. Another source of applications of formal concepts comes from data mining. The task of listing all formal concepts is closely related to mining of association rules [1]. Namely, the frequent closed itemsets which appear in mining nonredundant association rules [1,25,34] can be identified with intents of formal concepts whose extents are sufficiently large. Recently, it has been shown in [7] that formal concepts can be used to find optimal factorization of Boolean matrices. In fact, formal concepts correspond with optimal solutions to the discrete basis problem discussed by Miettinen et al. [28]. Finding formal concepts is therefore an important task. The algorithm we propose in this paper behaves well on both sparse and dense incidence data (of reasonable size).

This paper is organized as follows. In Section 2 we recall CbO and introduce the canonicity test. Section 3 describes the new algorithm, shows its correctness, and comments on the relationship to other algorithms. In Section 4 we discuss complexity and efficiency issues, and present an experimental evaluation of the performance of the algorithm.

2. Canonicity test and CbO

In this section we recall CbO [19,21] and the canonicity test. The next section will describe the new algorithm. In the sequel, we assume that $X = \{0, 1, \ldots, m\}$ and $Y = \{0, 1, \ldots, n\}$ are finite nonempty sets of objects and attributes, respectively, and $I \subseteq X \times Y$. Since I is fixed, the concept-forming operators $^{\uparrow_I}$ and $^{\downarrow_I}$ defined by (5) and (6) will be denoted just by $^{\uparrow}$ and $^{\downarrow}$, respectively. The set of all formal concept in I will be denoted by $\mathcal{B}(X, Y, I)$.

CbO has been introduced in [19] (a paper in Russian) and later used and described in [21]. The algorithm is also related to the algorithm proposed by Norris [30] which can be seen as an incremental variant of CbO. CbO lists all formal concepts by a systematic search in the space of all formal concepts, avoiding to list the same concept multiple times by performing a canonicity test. Conceptually, CbO is similar to NextClosure [9] because it uses the same canonicity test but NextClosure lists concepts in a different order. In [21], CbO is described in terms of backtracking. In this section we are going to use a simplified version of CbO introduced in [15] which is formalized by a recursive procedure performing a depth-first search in the space of all formal concepts. This type of description will shed more light on the new algorithm.

The core of CbO is a recursive procedure GenerateFrom, see Algorithm 1. The procedure accepts a formal concept $\langle A,B\rangle$ (an initial formal concept) and an attribute $y \in Y$ (first attribute to be processed) as its arguments. The procedure recursively descends through the space of formal concepts, beginning with $\langle A,B\rangle$.

Algorithm 1: Procedure GenerateFrom($\langle A,B\rangle, y$)

```
1  list ⟨A,B⟩ (e.g., print ⟨A,B⟩ on the screen);
2  if B = Y or y > n then
3    return
4  end
5  for j from y upto n do
6    if j ∉ B then
7      set C to A ∩ {j}↓;
8      set D to C↑;
9      if B ∩ Y_j = D ∩ Y_j then
10       GenerateFrom(⟨C,D⟩, j + 1);
11     end
12   end
13 end
14 return
```

When invoked with $\langle A,B\rangle$ and $y \in Y$, GenerateFrom first lists $\langle A,B\rangle$ (line 1) and then it checks its halting condition (lines 2–4). The computation stops either when $\langle A,B\rangle$ equals $\langle Y^{\downarrow},Y\rangle$ (the least formal concept has been reached) or $y > n$ (there are no more remaining attributes to be processed). Otherwise, the procedure goes through all attributes $j \in Y$ such that $j \geqslant y$ which are not present in the intent B (lines 5 and 6). For each such $j \in Y$, a new formal concept $\langle C,D\rangle = \langle A \cap \{j\}^{\downarrow}, (A \cap \{j\}^{\downarrow})^{\uparrow}\rangle$ is computed (lines 7 and 8). After obtaining $\langle C,D\rangle$, the algorithm uses the canonicity test to check whether it should continue with $\langle C,D\rangle$ by recursively calling GenerateFrom or whether $\langle C,D\rangle$ should be "skipped". The canonicity test (line 9) is based on comparing $B \cap Y_j = D \cap Y_j$ where $Y_j \subseteq Y$ is defined by

$$Y_j = \{y \in Y | y < j\}. \tag{8}$$

If the test passes, GenerateFrom is called with $\langle C,D\rangle$ and $j+1$, otherwise, the loop between lines 5–13 continues with the next value of j. The algorithm is correct: if GenerateFrom is invoked with $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle$ and 0, it lists each formal concept exactly once, i.e., the canonicity test prevents a concept from being listed multiple times. The proof for the original CbO is elaborated in [18].

Since we have formulated the algorithm as a recursive procedure rather than using backtracking, we provided an independent proof of its correctness using so-called derivations which we introduced in [15] for the purpose of analysis of parallel implementations of CbO. Recall from [15] that derivations correspond to recursive invocations of GenerateFrom. In a more detail, for $\langle A_1,B_1\rangle$, $\langle A_2,B_2\rangle \in \mathcal{B}(X,Y,I)$ and integers $y_1,y_2 \in Y \cup \{n+1\}$ let $\langle\langle A_1,B_1\rangle,y_1\rangle \vdash \langle\langle A_2,B_2\rangle,y_2\rangle$ denote that for $m = y_2 - 1$ the following conditions

(i) $m \notin B_1$,
(ii) $y_1 < y_2$,
(iii) $B_2 = (B_1 \cup \{m\})^{\downarrow\uparrow}$, and
(iv) $B_1 \cap Y_m = B_2 \cap Y_m$ where Y_m is defined by (8)

are all satisfied. A derivation of $\langle A,B\rangle \in \mathcal{B}(X,Y,I)$ of length $k+1$ is any sequence

$$\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle = \langle\langle A_0,B_0\rangle,y_0\rangle, \langle\langle A_1,B_1\rangle,y_1\rangle, \ldots, \langle\langle A_k,B_k\rangle,y_k\rangle = \langle\langle A,B\rangle,y_k\rangle \tag{9}$$

such that $\langle\langle A_i,B_i\rangle,y_i\rangle \vdash \langle\langle A_{i+1},B_{i+1}\rangle,y_{i+1}\rangle$ for each $i = 0,\ldots,k-1$. If $\langle A,B\rangle$ has a derivation of length k we say that $\langle A,B\rangle$ is derivable in k steps.

We can prove the following

Theorem 1 (Existence and Uniqueness of Derivations [15]). *Each $\langle A,B\rangle \in \mathcal{B}(X,Y,I)$ has exactly one derivation. Namely, the derivation of the form (9) in which $y_i = m_i + 1$ and $m_i = \min\{y \in B | y \notin B_{i-1}\}$ hold for all $0 < i \leqslant k$.* □

There is a correspondence between derivations and consecutive invocations of the procedure GenerateFrom. Namely, $\langle\langle A,B\rangle,y\rangle \vdash \langle\langle C,D\rangle,k\rangle$ iff the invocation of GenerateFrom($\langle A,B\rangle, y$) causes GenerateFrom($\langle\langle C,D\rangle, k\rangle$ to be called in line 10 of Algorithm 1. Indeed, (i) ensures that the condition in line 6 of Algorithm 1 is satisfied, (ii) corresponds to the fact that the loop between lines 5–13 goes from y upwards, (iii) says that D is the intent computed in line 8 because

$$D = (B \cup \{m\})^{\downarrow\uparrow} = (B \cup \{k-1\})^{\downarrow\uparrow} = (A \cap \{k-1\}^{\downarrow})^{\uparrow} = C^{\uparrow}$$

and (iv) is true iff the condition in line 9 is true.

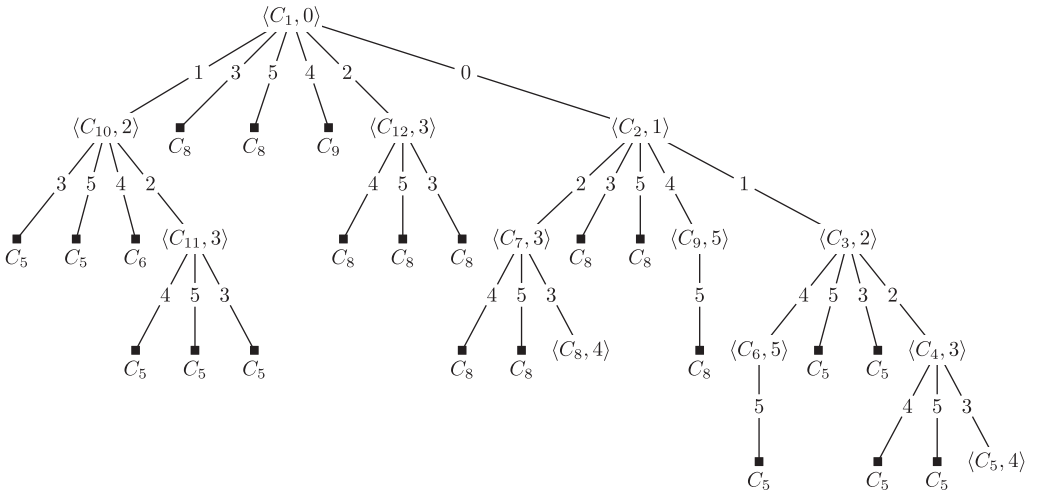

Fig. 1. Call tree of GenerateFrom for $I \subseteq X \times Y$ from Example 1.

The computation of Algorithm 1 and the corresponding derivations can be depicted by a tree as that in Fig. 1. The tree contains two types of nodes: (i) nodes represented by couples $\langle C_i, y_i \rangle$ corresponding to invocations of GenerateFrom with the arguments C_i (a formal concept) and y_i (an attribute), and (ii) leaf nodes denoted by black squares representing computed concepts for which the canonicity test fails. Edges in the tree are labeled by the values of j which are used to compute (new) formal concepts, see lines 7 and 8 of Algorithm 1. That is, nodes $\langle C_i, y_i \rangle$ and $\langle C_j, y_j \rangle$ are connected by an edge with label k iff $\langle C_i, y_i \rangle \vdash \langle C_j, y_j \rangle$ and $y_j = k + 1$. We call such a tree a call tree of GenerateFrom for a given $I \subseteq X \times Y$. It is easily seen that each path from the root of the tree to any node labeled by $\langle C_i, y_i \rangle$ corresponds to a derivation of $\langle C_i, y_i \rangle$. Due to Theorem 1, the nodes labeled by $\langle C_i, y_i \rangle$ are always in a one-to-one correspondence with formal concepts in $\mathcal{B}(X, Y, I)$, showing that the Algorithm 1 is correct. Let us note that there is a correspondence between a call tree like that in Fig. 1 and a CbO-tree described in [21]: our derivations correspond to canonical paths in the CbO-tree. Moreover, paths which are not canonical according to [21] can be seen as paths from the root node of the call tree of GenerateFrom to nodes labeled by black squares.

Example 1. Algorithm 1 and derivations are further demonstrated by the following example. Consider a set $X = \{0, \ldots, 3\}$ of objects and a set $Y = \{0, \ldots, 5\}$ of attributes. An incidence relation $I \subseteq X \times Y$ is given by the following table:

I	0	1	2	3	4	5
0	×	×	×			
1	×		×	×	×	×
2	×	×			×	
3		×	×			

where rows correspond to objects from X, columns correspond to attributes from Y, and table entries "×" or "blank" indicate whether for an object x and an attribute y we have $\langle x, y \rangle \in I$ or $\langle x, y \rangle \notin I$, respectively. The concept-forming operators $^{\uparrow} : 2^X \to 2^Y$ and $^{\downarrow} : 2^Y \to 2^X$ induced by such I have 12 fixpoints:

$$
\begin{aligned}
&C_1 = \langle \{0,1,2,3\}, \emptyset \rangle, && C_5 = \langle \emptyset, \{0,1,2,3,4,5\} \rangle, && C_9 = \langle \{1,2\}, \{0,4\} \rangle,\\
&C_2 = \langle \{0,1,2\}, \{0\} \rangle, && C_6 = \langle \{2\}, \{0,1,4\} \rangle, && C_{10} = \langle \{0,2,3\}, \{1\} \rangle,\\
&C_3 = \langle \{0,2\}, \{0,1\} \rangle, && C_7 = \langle \{0,1\}, \{0,2\} \rangle, && C_{11} = \langle \{0,3\}, \{1,2\} \rangle,\\
&C_4 = \langle \{0\}, \{0,1,2\} \rangle, && C_8 = \langle \{1\}, \{0,2,3,4,5\} \rangle, && C_{12} = \langle \{0,1,3\}, \{2\} \rangle.
\end{aligned}
$$

The concepts are numbered as they are listed by procedure GenerateFrom. Notice that $C_1 = \langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$ represents the initial formal concept which is processed by GenerateFrom. The corresponding call tree can be found in Fig. 1. One can read from the tree that, for example, $\langle C_1, 0 \rangle \vdash \langle C_2, 1 \rangle$, $\langle C_2, 1 \rangle \vdash \langle C_3, 2 \rangle$, and $\langle C_3, 2 \rangle \vdash \langle C_6, 5 \rangle$. Therefore, $\langle C_1, 0 \rangle$, $\langle C_2, 1 \rangle$, $\langle C_3, 2 \rangle$, $\langle C_6, 5 \rangle$ is a derivation and C_6 is derivable in 4 steps. The dataset used in this example will be used to illustrate our improvement of the canonicity test. ■

3. Improved canonicity test and FCbO

In this section, we propose an improvement of the canonicity test used by CbO that reduces the number of formal concepts computed multiple times. In a call tree like that in Fig. 1, such formal concepts are depicted by the black-square nodes.

Our new test and the improved algorithm will reduce the number of such nodes in the call tree without altering the rest of the tree. The major problem with the original canonicity test used by CbO is that it is always used *after* a new formal concept is computed, i.e., after performing the operation of computing a new fixpoint of $^{\downarrow\uparrow}$. We propose to employ an additional test that can be performed *before* a new formal concept is computed, eliminating thus the expensive computation of fixpoints.

3.1. Fast canonicity test

Let us first inspect the canonicity test

$$B \cap Y_j = D \cap Y_j \tag{10}$$

that appears in line 9 of Algorithm 1. Since $^{\downarrow\uparrow}$ is a closure operator and $D = (B \cup \{j\})^{\downarrow\uparrow}$, the monotony of $^{\downarrow\uparrow}$ yields $B \subseteq D$. Thus, it is sufficient to check just the inclusion $B \cap Y_j \supseteq D \cap Y_j$ instead of (10). In other words, the test succeeds iff D and B agree on all attributes which are smaller than j. Hence, the test (10) fails (i.e., the equality is not true) iff the fixpoint $D = (B \cup \{j\})^{\downarrow\uparrow}$ contains an attribute which is "before j" and the attribute is not present in B. Let us denote all such attributes by $B \otimes j$, i.e.

$$B \otimes j = (D \setminus B) \cap Y_j = \left((B \cup \{j\})^{\downarrow\uparrow} \setminus B\right) \cap Y_j. \tag{11}$$

The following lemma shows that knowing that (10) fails for given B and $j \notin B$, we can conclude that the test will also fail for each $B' \supseteq B$ with $j \notin B'$ as long as $B \otimes j$ contains an attribute which is not in B':

Lemma 2 (*On Test Failure Propagation*). *Let $B \subseteq Y$, $j \notin B$, and $B \otimes j \neq \emptyset$. Then, for each $B' \supseteq B$ such that $j \notin B'$ and $B \otimes j \not\subseteq B'$, we have $B' \otimes j \neq \emptyset$.*

Proof. Notice that $B \otimes j = (D \setminus B) \cap Y_j \neq \emptyset$ for $D = (B \cup \{j\})^{\downarrow\uparrow}$ means that (10) fails for such B, D and $j \notin B$. Take any $B' \supseteq B$ such that $j \notin B'$ and $B \otimes j \not\subseteq B'$. Let $D' = (B' \cup \{j\})^{\downarrow\uparrow}$. Since $j \notin B'$, we get $B' \subset D'$. In order to show that $B' \otimes j \neq \emptyset$, we prove that $B' \cap Y_j \subset D' \cap Y_j$. Since $B \otimes j \not\subseteq B'$, there is an attribute $y \in B \otimes j$ such that $y \notin B'$. Thus, it suffices to prove that $y \in D' \cap Y_j$. The fact that $y \in Y_j$ follows directly from $y \in B \otimes j = (D \setminus B) \cap Y_j$. Moreover, $y \in B \otimes j$ yields $y \in D$. Using monotony of the closure operator $^{\downarrow\uparrow}$, we get $y \in D = (B \cup \{j\})^{\downarrow\uparrow} \subseteq (B' \cup \{j\})^{\downarrow\uparrow} = D'$, proving the claim. Altogether, $B' \cap Y_j \subset D' \cap Y_j$, i.e. $B' \otimes j \neq \emptyset$. □

Based on Lemma 2, we get the following characterization of derivations:

Theorem 3 (On Nonexistence of Derivations). *Let $\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle, \ldots, \langle\langle A, B\rangle, y\rangle$ be a derivation and let $j \geqslant y$ be such that $j \notin B$ and $B \otimes j \neq \emptyset$. Then there is no derivation*

$$\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle, \ldots, \langle\langle A, B\rangle, y\rangle, \ldots, \langle\langle A', B'\rangle, y'\rangle, \langle\langle C', D'\rangle, j+1\rangle,$$

where $B \otimes j \not\subseteq B'$.

Proof. The claim is a consequence of Lemma 2. Indeed, take arbitrary $B' \supseteq B$ such that $B \otimes j \not\subseteq B'$. Assume there is a sequence

$$\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle, \ldots, \langle\langle A, B\rangle, y\rangle, \ldots, \langle\langle A', B'\rangle, y'\rangle$$

which is a derivation of $\langle A', B'\rangle$. We can prove that the derivation cannot be extended by $\langle\langle C', D'\rangle, j+1\rangle$. By contradiction, assume that $\langle\langle A', B'\rangle, y'\rangle \vdash \langle\langle C', D'\rangle, j+1\rangle$. By definition of "$\vdash$", we get $D' = (B' \cup \{j\})^{\downarrow\uparrow}$ and $B' \cap Y_j = D' \cap Y_j$, i.e., $B' \otimes j = (D' \setminus B') \cap Y_j = \emptyset$. On the other hand, we have assumed $B \otimes j \not\subseteq B'$, i.e. Lemma 2 yields $B' \otimes j \neq \emptyset$, a contradiction to $B' \otimes j = \emptyset$. □

The result shown in Theorem 3 allows us to split the canonicity test into two parts: First part which is quick and does not require computing closures and a second part which is basically the original canonicity test. Indeed, according to Theorem 3, if we know that $B \otimes j \neq \emptyset$ for some $j \notin B$ then having a derivation

$$\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle, \ldots, \langle\langle A, B\rangle, y\rangle, \ldots, \langle\langle A', B'\rangle, y'\rangle$$

with $B \otimes j \not\subseteq B'$, we automatically know (without computing any closures) that it cannot be further extended by $\langle\langle C', D'\rangle, j+1\rangle$. In other words, $D' = (B' \cup \{j\})^{\downarrow\uparrow}$ is not computed at all. Therefore, the first part of the new test uses the observation of Theorem 3. If the first part of the test cannot be applied because $B \otimes j = \emptyset$, we still have to perform the second part of the test, i.e., the original canonicity test which involves computing the closure $(B' \cup \{j\})^{\downarrow\uparrow}$. Nevertheless, we will see in Section 4 that the number of cases in which we actually perform the original canonicity test is surprisingly low compared to the number of quick tests based on Theorem 3. The idea of the new combined canonicity test is further illustrated by the following example.

Example 2. Consider the input data from Example 1 and the corresponding call tree in Fig. 1. If we apply the new canonicity test based on Theorem 3, we in fact perform a particular tree pruning in which we omit some of the black-square leaf nodes of the tree. The result is shown in Fig. 2. The bold edges are those which remain in the call tree. The leaf nodes that are omitted are denoted in gray and the corresponding edges are dotted. Notice that not all black-square leaf nodes are omitted.

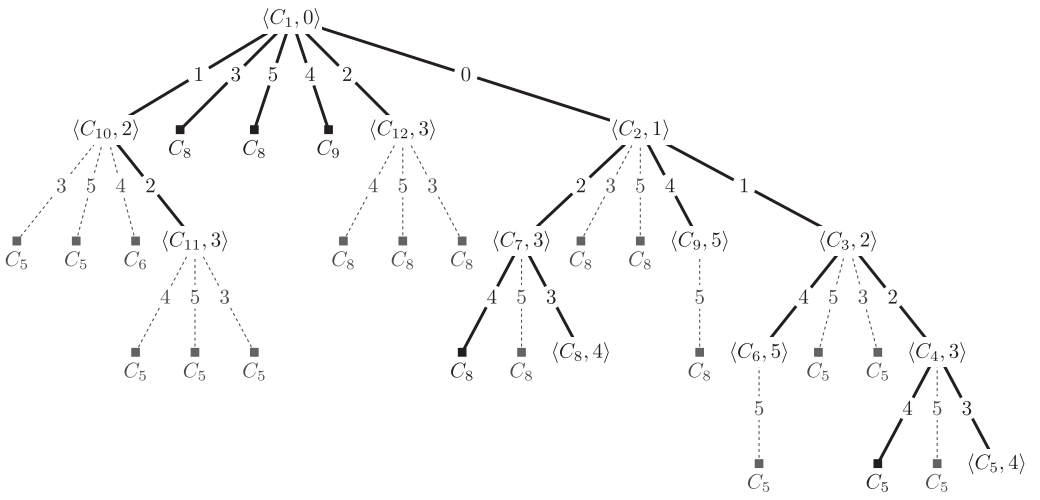

Fig. 2. Example of a call tree with a reduced number of leaf nodes.

C_8 appears three times as a leaf node, C_9 and C_5 appear once, meaning that the formal concept C_8 is computed four times during the computation and both C_9 and C_5 are computed twice. The total number of closures computed during the computation is 17 which is a significant reduction compared to the 34 nodes of the original call tree in Fig. 1.

Let us outline how the new test is used to prune the tree. Consider the first formal concept $C_1 = \langle A_1, B_1 \rangle = \langle \{0,1,2,3\}, \emptyset \rangle$, see Example 1 for the list of all concepts. At this point, we perform the usual canonicity test because we have no information from previous levels of the tree (we are on the top of the tree). For $j \in \{0,1,2\}$, the test succeeds. For instance, in case of $j = 2$, we get $C_{12} = \langle A_{12}, B_{12} \rangle = \langle \{0,1,3\}, \{2\} \rangle$, i.e. $B_1 \cap Y_2 = \emptyset = B_{12} \cap Y_2$. On the other hand, the test fails for $j \in \{3,4,5\}$. For instance, in case of $j = 3$, we get $C_8 = \langle A_8, B_8 \rangle = \langle \{1\}, \{0,2,3,4,5\} \rangle$ and hence $B_1 \cap Y_3 = \emptyset \neq \{0,2\} = B_8 \cap Y_3$. Therefore, $B_1 \otimes 3 = \{0,2\}$. Analogously, we get $B_1 \otimes 4 = \{0\}$ and $B_1 \otimes 5 = \{0,2,3,4\}$. The sets $B_1 \otimes 3$, $B_1 \otimes 4 = \{0\}$, and $B_1 \otimes 5$ can be further used to prune the tree according to Theorem 3. Indeed, consider the tree node $\langle C_{10}, 2 \rangle$. Since $C_{10} = \langle A_{10}, B_{10} \rangle = \langle \{0,2,3\}, \{1\} \rangle$, we get $B_1 \otimes 3 \not\subseteq B_{10}$, $B_1 \otimes 4 \not\subseteq B_{10}$, and $B_1 \otimes 5 \not\subseteq B_{10}$, i.e. neither $j \in \{3,4,5\}$ can be used to extend the derivation. In case of $j = 2$, we perform the usual canonicity test which is successful. In a similar way, the tree can be pruned beginning with the other nodes $\langle C_i, y_i \rangle$.

The fast test based on Theorem 3 is not always applicable. It is evident that we cannot apply the test on the top-most level of the call tree. There are, however, situations where it cannot be applied on deeper levels as well. Consider, e.g., the tree node $\langle C_7, 3 \rangle$ where $C_7 = \langle A_7, B_7 \rangle = \langle \{0,1\}, \{0,2\} \rangle$. Since $B_1 \otimes 4 = \{0\} \subseteq B_7$, Theorem 3 cannot be applied. On the other hand, if we perform the original canonicity test with B_7 and $(B_7 \cup \{4\})^{\downarrow\uparrow} = \{0,2,3,4,5\} = B_8$, we get $B_7 \cap Y_4 = \{0,2\} \neq \{0,2,3\} = B_8 \cap Y_4$, i.e., the derivation cannot be extended by $\langle C_8, 5 \rangle$ but in order to see this we had to compute the closure $(B_7 \cup \{4\})^{\downarrow\uparrow} = B_8$ which should be considered an expensive operation (especially in case of large data sets). A similar situation appears in case of the node $\langle C_4, 3 \rangle$ and $j = 4$, cf. Fig. 2. ■

3.2. Modified algorithm

In this section, we describe how the new canonicity test based on Theorem 3 can be implemented in an extended version of CbO. As Example 2 shows, during the computation we have to propagate the information about sets $B_i \otimes y_i$ which take part in the new test. In particular, the information must be propagated in the top-down direction, from the root node of the call tree to the leaves. As a consequence, we have to change the search strategy of the algorithm (the depth-first search in the space of concepts as it is used in CbO is no longer useful), resulting in a new algorithm called FCbO ("F" stands for "Fast").

Remark 1. A call tree is a diagram depicting recursive calls of GenerateFrom. Consecutive invocations of GenerateFrom correspond to the depth-first search in the call tree. For instance, in case of node $\langle C_1, 0 \rangle$ in Fig. 1, GenerateFrom continues with the subtree with root node $\langle C_2, 1 \rangle$. After the whole subtree is processed, it continues with the subtree with root node $\langle C_{10}, 2 \rangle$, etc. The problem with this behavior is that in order to apply the new canonicity test in the subtree with root $\langle C_2, 1 \rangle$, we shall already have the information about $B_1 \otimes 3 = \{0,2\}$. Analogously, we get $B_1 \otimes 4 = \{0\}$ and $B_1 \otimes 5 = \{0,2,3,4\}$, see Example 2, which is available *only after* we process all attributes in the invocation of $\langle C_1, 0 \rangle$. Therefore, we are going to modify GenerateFrom so that instead of the recursive calls, it stores information about computed concepts in a queue. Then, after all attributes are processed, it performs a recursive invocation for each concept in the queue. This effectively changes the order in which we compute new concepts because we use a combined depth-first and breadth-first search in the call tree but it *does not* change the order of listing of formal concepts because the listing appears *after* each recursive call, as in CbO. ■

Algorithm 2: Procedure FastGenerateFrom($\langle A, B\rangle$, y, $\{N_y | y \in Y\}$)

```
1  list ⟨A,B⟩ (e.g., print A and B on screen);
2  if B = Y or y > n then
3    return
4  end
5  for j from y upto n do
6    set M_j to N_j;
7    if j ∉ B and N_j ∩ Y_j ⊆ B ∩ Y_j then
8      set C to A ∩ {j}^↓;
9      set D to C^↑;
10     if B ∩ Y_j = D ∩ Y_j then
11       put ⟨⟨C,D⟩,j + 1⟩ to queue;
12     else
13       set M_j to D;
14     end
15   end
16 end
17 while get ⟨⟨C,D⟩,j⟩ from queue do
18   FastGenerateFrom(⟨C, D⟩, j, {M_y | y ∈ Y});
19 end
20 return
```

We are going to represent FCbO by a recursive procedure FastGenerateFrom, see Algorithm 2. The procedure accepts three arguments: a formal concept $\langle A,B\rangle$ (an initial formal concept), an attribute $y \in Y$ (first attribute to be processed), and a set $\{N_y \subseteq Y | y \in Y\}$ of subsets of attributes Y. The intended meaning of the first two arguments is the same as in case of GenerateFrom, see Algorithm 1. The purpose of the third argument is to carry information about attributes in sets $B_i \oslash y_i$. The precise meaning of N_y will be specified later. Each invocation of FastGenerateFrom uses the following *local variables*: a *queue* as a temporary storage for computed concepts and sets of attributes M_y ($y \in Y$) which are used in place of the third argument for further invocations of FastGenerateFrom.

When invoked with $\langle A,B\rangle$, $y \in Y$, and $\{N_y | y \in Y\}$, FastGenerateFrom first processes $\langle A,B\rangle$ and then it checks the same halting condition as GenerateFrom, see lines 1–4. If the computation does not halt, the procedure goes through all attributes $j \in Y$ such that $j \geqslant y$, see lines 5–16. For each such j, the procedure creates a local copy M_j of the set N_j (line 6). If $j \notin B$, a test based on Theorem 3 is performed by checking $N_j \cap Y_j \subseteq B \cap Y_j$. If the test succeeds, the procedure goes on with computing a new formal concept $\langle C,D\rangle$, see lines 8 and 9. Then it performs the original canonicity test (line 10). If the test is positive, the formal concept $\langle C,D\rangle$ together with the attribute $j + 1$ are stored in a *queue* (line 11). Otherwise, M_j is set to D (line 13). Notice that the loop between lines 5–15 does not perform any recursive calls of FastGenerateFrom. Instead, the information about computed concepts and attributes used to generate the concepts is stored in the *queue*. The recursive invocations of FastGenerateFrom are performed after all the attributes are processed. Indeed, the loop between lines 17–19 goes over all records in the *queue* and recursively calls FastGenerateFrom with arguments being the new concept, new starting attribute, and new set of subsets $\{M_y | y \in Y\}$ of attributes.

In order to list all formal concepts, we invoke Algorithm 2 with $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle$, $y = 0$ and $\{N_y = \emptyset | y \in Y\}$ as its initial arguments. The following assertion says that the algorithm is correct:

Theorem 4 (Correctness of FCbO). *When invoked with $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle$, $y = 0$, and $\{N_y = \emptyset | y \in Y\}$, Algorithm 2 lists all formal concepts in $\langle X, Y, I\rangle$, each of them exactly once.*

Proof. Since Algorithm 1 (CbO) is correct, is it sufficient to show that Algorithm 2 (FCbO) does not omit any formal concept during the computation. Thus, we have to check that the new canonicity test is applied correctly. The rest follows from the correctness of Algorithm 1, in particular the existence and uniqueness of derivations, see Theorem 1. Let us inspect the values of N_j's and M_j's during each invocation of FastGenerateFrom. During the first invocation, $\{N_y = \emptyset | y \in Y\}$, i.e. $N_j \cap Y_j = \emptyset \subseteq B \cap Y_j$ is trivially true, i.e. each attribute $j \notin B$ is processed between lines 8–14. As one can see, during each invocation of FastGenerateFrom, the value of M_j is either equal to N_j (we say that the value of M_j is *inherited* from previous invocation) or M_j equals $D = (B \cup \{j\})^{\downarrow\uparrow}$ (we say that the value of M_j is *updated* in the current invocation). If M_j is updated then M_j is the intent of a formal concept $\langle C,D\rangle$ which fails the canonicity test in line 10. Therefore, it is easy to see that during an invocation of FastGenerateFrom($\langle A, B\rangle, y, \{N_y | y \in Y\}$), for each $j \geqslant y$, either $N_j = \emptyset$ or there is a formal concept $\langle A^*, B^*\rangle$ such that the following hold

(i) $B^* \subseteq B$,
(ii) $B^* \oslash j \neq \emptyset$, and
(iii) $N_j = (B^* \cup \{j\})^{\downarrow\uparrow}$.

Notice that from (iii) it follows that $B^* \otimes j = ((B^* \cup \{j\})^{\downarrow\uparrow} \setminus B^*) \cap Y_j = (N_j \setminus B^*) \cap Y_j$. Hence, in order to prove correctness, it suffices to show that the condition $N_j \cap Y_j \subseteq B \cap Y_j$ present in line 7 of Algorithm 2 fails iff $B^* \otimes j \not\subseteq B$ which appears in Theorem 3 as a necessary condition for pruning. Therefore, we prove the following

Claim 1.

$B^* \otimes j \subseteq B$ iff $N_j \cap Y_j \subseteq B \cap Y_j$:

"⇒": Let $B^* \otimes j \subseteq B$. Using (iii), we get $(N_j \setminus B^*) \cap Y_j \subseteq B$. Furthermore, (i) yields $N_j \setminus B \subseteq N_j \setminus B^*$, i.e. we obtain $(N_j \setminus B) \cap Y_j \subseteq B$. The last inclusion implies that $N_j \cap Y_j \subseteq B \cap Y_j$. Indeed, by contradiction, from $y \in N_j \cap Y_j$ and $y \notin B$ it follows that $y \in N_j$, i.e., $y \in N_j \setminus B$ and thus $y \in (N_j \setminus B) \cap Y_j \subseteq B$ because $y \in Y_j$, contradicting the fact that $y \notin B$. Therefore, we have $N_j \cap Y_j \subseteq B \cap Y_j$. "⇐": Suppose that $B^* \otimes j \not\subseteq B$. Then, there is $y \in B^* \otimes j$ such that $y \notin B$. From $y \in B^* \otimes j$ and (iii), we get $y \in N_j$ and $y \in Y_j$. Therefore, $y \in N_j \cap Y_j$ and $y \notin B \cap N_j$ because $y \notin B$, showing $N_j \cap Y_j \not\subseteq B \cap Y_j$.

Therefore, as a consequence of Theorem 3, if $N_j \cap Y_j \subseteq B \cap Y_j$ fails then we can skip the attribute j because B and $D = (B \cup \{j\})^{\downarrow\uparrow}$ would fail the canonicity test in line 10. Altogether, FCbO lists all formal concepts, each of them exactly once. □

Remark 2. In Algorithm 2, the additional information about attributes that is needed to perform the test is stored in procedure arguments N_y which are, in fact, particular intents. On the other hand, the test formulated in Theorem 3 is based on sets of the form $B^* \otimes j$. We use N_y's instead of sets $B^* \otimes j$ because of efficiency reasons: Since N_y represents an intent of a concept that has already been computed, the third argument $\{N_y = \emptyset | y \in Y\}$ for FastGenerateFrom can be organized as a list (or an array) of references (pointers) to such intents. Storing referenced objects in a linear data structure is much cheaper an operation than computing $B^* \otimes j$ and storing the resulting value. More efficiency issues will be discussed in Section 4. ■

The following example illustrates recursive invocations of FastGenerateFrom during the computation.

Example 3. We demonstrate the computation of Algorithm 2 for the input data from Example 1 by listing important steps of the algorithm. We focus on steps performed in lines 1 (listing of found formal concepts), 7 (quick canonicity test), 11 (putting a new concept to a queue), 13 (updating information about attributes in sets $B_i \otimes y_i$), and 18 (recursive invocations of FastGenerateFrom). In addition to that, if an invocation of FastGenerateFrom terminates either in line 3 or 20, we denote this fact by "⊥" in a separate line. Nested invocations are separated by horizontal indent. In the example, each formal concept is denoted by $C_i = \langle A_i, B_i \rangle$, i.e., each B_i is the intent of the corresponding C_i. The rest of the notation is the same as in Algorithm 2. When FastGenerateFrom is invoked with C_1, 0, and $\{N_y = \emptyset | y \in Y\}$, the computation proceeds as follows:

line 1: **list** C_1 = $\langle \{0,1,2,3\}, \emptyset \rangle$
line 7: trivial success for $j = 0$ because $N_0 = \emptyset$
line 11: put $\langle C_2, 1 \rangle = \langle \langle \{0,1,2\}, \{0\} \rangle, 1 \rangle$ to *queue*
line 7: trivial success for $j = 1$ because $N_1 = \emptyset$
line 11: put $\langle C_{10}, 2 \rangle = \langle \langle \{0,2,3\}, \{1\} \rangle, 2 \rangle$ to *queue*
line 7: trivial success for $j = 2$ because $N_2 = \emptyset$
line 11: put $\langle C_{12}, 3 \rangle = \langle \langle \{0,1,3\}, \{2\} \rangle, 3 \rangle$ to *queue*
line 7: trivial success for $j = 3$ because $N_3 = \emptyset$
line 13: set M_3 to $D = (\emptyset \cup \{3\})^{\downarrow\uparrow} = \{0,2,3,4,5\} = B_8$
line 7: trivial success for $j = 4$ because $N_4 = \emptyset$
line 13: set M_4 to $D = (\emptyset \cup \{4\})^{\downarrow\uparrow} = \{0,4\} = B_9$
line 7: trivial success for $j = 5$ because $N_5 = \emptyset$
line 13: set M_5 to $D = (\emptyset \cup \{5\})^{\downarrow\uparrow} = \{0,2,3,4,5\} = B_8$
line 18: get $\langle C_2, 1 \rangle$ from *queue* and call FastGenerateFrom(C_2, 1, $\{M_y | y \in Y\}$)
  line 1: **list** C_2 = $\langle \{0,1,2\}, \{0\} \rangle$
  line 7: trivial success for $j = 1$ because $N_1 = \emptyset$
  line 11: put $\langle C_3, 2 \rangle = \langle \langle \{0,2\}, \{0,1\} \rangle, 2 \rangle$ to *queue*
  line 7: trivial success for $j = 2$ because $N_2 = \emptyset$
  line 11: put $\langle C_7, 3 \rangle = \langle \langle \{0,1\}, \{0,2\} \rangle, 3 \rangle$ to *queue*
  line 7: failure for $j = 3$, $B = \{0\}$, and $N_3 = \{0,2,3,4,5\} = B_8$ because $\{2\} \not\subseteq B$
  line 7: success for $j = 4$, $B = \{0\}$, and $N_4 = \{0,4\} = B_9$
  line 11: put $\langle C_9, 5 \rangle = \langle \langle \{1,2\}, \{0,4\} \rangle, 5 \rangle$ to *queue*
  line 7: failure for $j = 5$, $B = \{0\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{2,3,4\} \not\subseteq B$
  line 18: get $\langle C_3, 2 \rangle$ from *queue* and call FastGenerateFrom(C_3, 2, $\{M_y | y \in Y\}$)
    line 1: **list** C_3 = $\langle \{0,2\}, \{0,1\} \rangle$
    line 7: trivial success for $j = 2$ because $N_2 = \emptyset$
    line 11: put $\langle C_4, 3 \rangle = \langle \langle \{0\}, \{0,1,2\} \rangle, 3 \rangle$ to *queue*

(*continued on next page*)

line 7: failure for $j = 3$, $B = \{0,1\}$, and $N_3 = \{0,2,3,4,5\} = B_8$ because $\{2\} \not\subseteq B$
line 7: success for $j = 4$, $B = \{0,1\}$, and $N_4 = \{0,4\} = B_9$
line 11: put $\langle C_6, 5\rangle = \langle\langle\{2\},\{0,1,4\}\rangle, 5\rangle$ to *queue*
line 7: failure for $j = 5$, $B = \{0,1\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{2,3,4\} \not\subseteq B$
line 18: get $\langle C_4, 3\rangle$ from *queue* and call FastGenerateFrom$(C_4, 3, \{M_y | y \in Y\})$
line 1: **list** $C_4 =$ $\langle\{0\},\{0,1,2\}\rangle$
line 7: success for $j = 3$, $B = \{0,1,2\}$, and $N_3 = \{0,2,3,4,5\} = B_8$
line 11: put $\langle C_5, 4\rangle = \langle\ \langle\emptyset,\{0,1,2,3,4,5\}\rangle, 4\rangle$ to *queue*
line 7: success for $j = 4$, $B = \{0,1,2\}$, and $N_4 = \{0,4\} = B_9$
line 13: set M_4 to $D = (\{0,1,2\} \cup \{4\})^{\downarrow\uparrow} = \{0,1,2,3,4,5\} = B_5$
line 7: failure for $j = 5$, $B = \{0,1,2\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{3,4\} \not\subseteq B$
line 18: get $\langle C_5, 4\rangle$ from *queue* and call FastGenerateFrom$(C_5, 4, \{M_y | y \in Y\})$
line 1: **list** $C_5 =$ $\langle\emptyset,\{0,1,2,3,4,5\}\rangle$
$\perp$ return from invocation for C_5
$\perp$ return from invocation for C_4
line 18: get $\langle C_6, 5\rangle$ from *queue* and call FastGenerateFrom$(C_6, 5, \{M_y | y \in Y\})$
line 1: **list** $C_6 =$ $\langle\{2\},\{0,1,4\}\rangle$
line 7: failure for $j = 5$, $B = \{0,1,4\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{2,3\} \not\subseteq B$
$\perp$ return from invocation for C_6
$\perp$ return from invocation for C_3
line 18: get $\langle C_7, 3\rangle$ from *queue* and call FastGenerateFrom$(C_7, 3, \{M_y | y \in Y\})$
line 1: **list** $C_7 =$ $\langle\{0,1\},\{0,2\}\rangle$
line 7: success for $j = 3$, $B = \{0,2\}$, and $N_3 = \{0,2,3,4,5\} = B_8$
line 11: put $\langle C_8, 4\rangle = \langle\langle\{1\},\{0,2,3,4,5\}\rangle, 4\rangle$ to *queue*
line 7: success for $j = 4$, $B = \{0,2\}$, and $N_4 = \{0,4\} = B_9$
line 13: set M_4 to $D = (\{0,2\} \cup \{4\})^{\downarrow\uparrow} = \{0,2,3,4,5\} = B_8$
line 7: failure for $j = 5$, $B = \{0,2\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{3,4\} \not\subseteq B$
line 18: get $\langle C_8, 4\rangle$ from *queue* and call FastGenerateFrom$(C_8, 4, \{M_y | y \in Y\})$
line 1: **list** $C_8 =$ $\langle\{1\},\{0,2,\ 3,4,5\}\rangle$
$\perp$ return from invocation for C_8
$\perp$ return from invocation for C_7
line 18: get $\langle C_9, 5\rangle$ from *queue* and call FastGenerateFrom$(C_9, 5, \{M_y | y \in Y\})$
line 1: **list** $C_9 =$ $\langle\{1,2\},\{0,4\}\rangle$
line 7: failure for $j = 5$, $B = \{0,4\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{2,3\} \not\subseteq B$
$\perp$ return from invocation for C_9
$\perp$ return from invocation for C_2
line 18: get $\langle C_{10}, 2\rangle$ from *queue* and call FastGenerateFrom$(C_{10}, 2, \{M_y | y \in Y\})$
line 1: **list** $C_{10} =$ $\langle\{0,2,3\},\{1\}\rangle$
line 7: trivial success for $j = 2$ because $N_2 = \emptyset$
line 11: put $\langle C_{11}, 3\rangle = \langle\langle\{0,3\},\{1,2\}\rangle, 3\rangle$ to *queue*
line 7: failure for $j = 3$, $B = \{1\}$, and $N_3 = \{0,2,3,4,5\} = B_8$ because $\{0,2\} \not\subseteq B$
line 7: failure for $j = 4$, $B = \{1\}$, and $N_4 = \{0,4\} = B_9$ because $\{0\} \not\subseteq B$
line 7: failure for $j = 5$, $B = \{1\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{0,2,3,4\} \not\subseteq B$
line 18: get $\langle C_{11}, 3\rangle$ from *queue* and call FastGenerateFrom$(C_{11}, 3, \{M_y | y \in Y\})$
line 1: **list** $C_{11} =$ $\langle\{0,3\},\{1,2\}\rangle$
line 7: failure for $j = 3$, $B = \{1,2\}$, and $N_3 = \{0,2,3,4,5\} = B_8$ because $\{0\} \not\subseteq B$
line 7: failure for $j = 4$, $B = \{1,2\}$, and $N_4 = \{0,4\} = B_9$ because $\{0\} \not\subseteq B$
line 7: failure for $j = 5$, $B = \{1,2\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{0,3,4\} \not\subseteq B$
$\perp$ return from invocation for C_{11}
$\perp$ return from invocation for C_{10}
line 18: get $\langle C_{12}, 3\rangle$ from *queue* and call FastGenerateFrom$(C_{12}, 3, \{M_y | y \in Y\})$
line 1: **list** $C_{12} =$ $\langle\{0,1,3\},\{2\}\rangle$
line 7: failure for $j = 3$, $B = \{2\}$, and $N_3 = \{0,2,3,4,5\} = B_8$ because $\{0\} \not\subseteq B$
line 7: failure for $j = 4$, $B = \{2\}$, and $N_4 = \{0,4\} = B_9$ because $\{0\} \not\subseteq B$
line 7: failure for $j = 5$, $B = \{2\}$, and $N_5 = \{0,2,3,4,5\} = B_8$ because $\{0,3,4\} \not\subseteq B$
$\perp$ return from invocation for C_{12}
$\perp$ return from invocation for C_1

Notice that line 7 is either success or failure depending on the outcome of the new canonicity test. Each occurrence of line 7 is followed either by line 11 or line 13 depending on the outcome of the original canonicity test in line 10 (for brevity, line 10 is not displayed). ■

3.3. On the relationship to NextClosure

Notice that FCbO lists all formal concepts in the same order as CbO. Although FCbO first computes closures which are put in a queue and then makes the appropriate recursive calls, it lists the concepts in the same order as CbO because the listing is performed as a first action after the invocation of FastGenerateFrom. Hence, the listing does not necessarily follow the computation of a closure as it can be seen from Example 3.

The order in which concepts are listed by FCbO can be changed in various ways. For instance, if we move line 1 of Algorithm 2 between lines 11 and 12, the concept will be listed in an order which agrees with the combined breadth-first and depth-first search order of the call tree, see Remark 1.

More importantly, the algorithm can be easily modified to produce formal concepts in the same order as Ganter's NextClosure algorithm [9]. Recall that NextClosure lists all concepts in a lectic order: $B_1 \subseteq Y$ is *lecticly smaller* [10] than $B_2 \subseteq Y$, denoted $B_1 <_\ell B_2$, if the smallest element that distinguishes B_1 and B_2 belongs to B_2. That is,

$$B_1 <_\ell B_2 \quad \text{iff} \quad \text{there is } j \in B_2 \setminus B_1 \text{ such that } B_1 \cap Y_j = B_2 \cap Y_j, \tag{12}$$

where Y_j is defined as in (8). It can be shown that $<_\ell$ is a total strict order on 2^Y. NextClosure lists the formal concepts in the (unique) order $<_\ell$ by an iterative computation of lectic successors, starting with the lecticly smallest concept $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$. The following claim characterizes nodes of a call tree in terms of their lectic relationship.

Theorem 5. *Let $\{\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle, \ldots, \langle\langle C, y\rangle, \langle C_i, y_i + 1\rangle | i \in J\}$ be a J-indexed set of derivations of formal concepts C_i with intents B_i. Let B be the intent of C. Then the following are true:*

(i) for each $i \in J$: $B <_\ell B_i$,
(ii) for each $j, k \in J$ with $y_j < y_k$ and each $\langle C^*, y^* + 1\rangle$ such that $\langle C_k, y_k + 1\rangle \vdash^* \langle C^*, y^* + 1\rangle : B^* <_\ell B_j$ where B^* is intent of C^* and $\vdash^*$ is the reflexive and transitive closure of $\vdash$.

Proof. See Fig. 3 for a symbolic schema for the proof.

(i) is easy to see: Since $\langle C, y\rangle \vdash \langle C_i, y_i + 1\rangle$, we have $B \cap Y_{y_i} = B_i \cap Y_{y_i}$. In addition to that, $y_i \in B_i$ and $y_i \notin B$, i.e. (12) is satisfied for j being y_i, showing $B <_\ell B_i$.

(ii) We check that for y_j we have $B^* \cap Y_{y_j} = B_j \cap Y_{y_j}$, $y_j \notin B^*$, and $y_j \in B_j$. From this, we get $B^* <_\ell B_j$ directly from (12). Notice that $y_j < y_k$ implies $Y_{y_j} \subset Y_{y_k}$. Thus, from $B \cap Y_{y_k} = B_k \cap Y_{y_k}$ it follows that $B \cap Y_{y_j} = B_k \cap Y_{y_j}$. Using $B \cap Y_{y_j} = B_j \cap Y_{y_j}$ and the latter equality, we get $B_k \cap Y_{y_j} = B_j \cap Y_{y_j}$. Moreover, from $\langle C_k, y_k + 1\rangle \vdash^* \langle C^*, y^* + 1\rangle$ it follows that $B_k \cap Y_{y_k} = B^* \cap Y_{y_k}$, i.e., $B_k \cap Y_{y_j} = B^* \cap Y_{y_j}$ because $y_j < y_k$. Putting $B_k \cap Y_{y_j} = B^* \cap Y_{y_j}$ and $B_k \cap Y_{y_j} = B_j \cap Y_{y_j}$ together, we get $B^* \cap Y_{y_j} = B_j \cap Y_{y_j}$. Thus, it remains to show that $y_j \in B_j$ and $y_j \notin B^*$. The first claim is evident. In order to prove $y_j \notin B^*$, observe that $y_j \notin B$ and consequently $y_j \notin B \cap Y_{y_k} = B_k \cap Y_{y_k} = B^* \cap Y_{y_k}$, meaning that $y_j \notin B^*$ because $y_j < y_k$. □

Theorem 5 shows how the call tree should be traversed if anyone wants to list all concepts according to $<_\ell$. If we take the concepts C and $\{C_i | i \in J\}$ as in Theorem 5, we can see that C should be listed before all C_i's due to (i). Furthermore, if we take two different concepts C_j and C_k ($j, k \in J$), C_k should be listed before C_j iff $y_j < y_k$ because of (ii). Therefore, (i) and (ii) mean that given a subtree of a call tree, the root node must be listed first and the descending nodes C_i should be listed in a descending order according to y_i. Furthermore, (ii) says that each node C^* derivable from C_k should also be listed before C_j and, at the same time, after C_k because of (i). This means that in order to list all formal concepts in a lectic order, we have to perform a depth-first search through the call tree, assuming that we process all attributes in the descending order. Since Algorithm 2 already performs the depth-first search, it suffices to ensure the descending order of processed attributes. We can do that by modifying Algorithm 2 in one of the following ways:

(i) we can use a *stack* instead of a *queue* to store computed formal concepts, or
(ii) we can modify the loop in line 5 so that it goes "**from** n **downto** y".

This way for obtaining concepts in a lectic order is much faster than the iterative algorithm from [10] because we can compute fixpoints of $^{\downarrow\uparrow}$ more efficiently, see Section 4, and we employ the fast canonicity test. Performance comparisons can be found in Section 4.2.

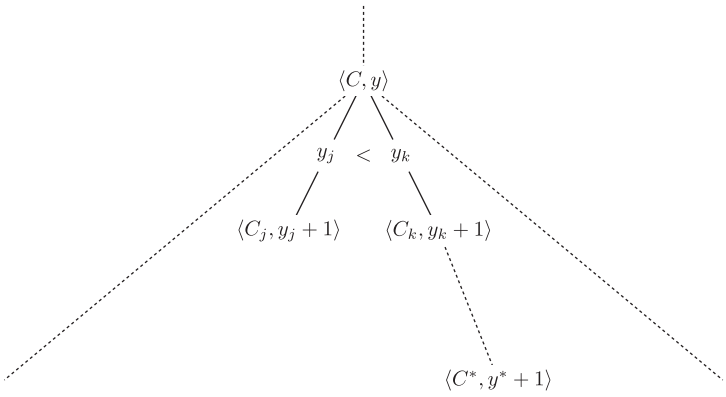

Fig. 3. Schema for the proof of Theorem 5.

3.4. On the relationship to AddIntent

In [27], the authors have introduced an incremental algorithm AddIntent for computing formal concepts together with the subconcept–superconcept hierarchy $\leqslant$ given by (7). Unlike FCbO, the algorithm proposed in [27] is incremental, i.e., it incrementally computes the concept lattice of given context by adding all attributes (or objects as in [27]) one by one. Interestingly, AddIntent includes a particular optimization that is analogous to the fast canonicity test of FCbO introduced in this paper. The key difference is that AddIntent uses a slightly different canonicity test that is based on the ordering $\leqslant$ of formal concepts (7), whereas FCbO uses the order of processed attributes. The approach used by AddIntent is more beneficial if one wants to compute the whole concept lattice instead of just computing the formal concepts. On the other hand, the approach taken by FCbO is simpler and is more efficient if only the set of formal concepts is considered.

Using the notation from our paper, [27] defines a notion of a canonical generator of a formal concept $\langle C,D\rangle$ which can be described as follows. First, denote by $\mathcal{B}_i(X,Y_i,I_i)$ the set of all formal concepts of a formal context $\langle X,Y_i,I_i\rangle$ where $I_i = I \cap (X \times Y_i)$ with Y_i defined as in (8) (i.e., $\langle X,Y_i,I_i\rangle$ represents the original context restricted to attributes $0,\ldots,i-1$). Then, $\langle C,D\rangle \in \mathcal{B}_{i+1}(X,Y_{i+1},I_{i+1})$ is called new in $\mathcal{B}_{i+1}(X,Y_{i+1},I_{i+1})$ if C is distinct from all concept extents from $\mathcal{B}_i(X,Y_i,I_i)$. Furthermore, if $\langle C,D\rangle$ is new in $\mathcal{B}_{i+1}(X,Y_{i+1},I_{i+1})$, then $\langle A,B\rangle \in \mathcal{B}_i(X,Y_i,I_i)$ is called a generator of $\langle C,D\rangle$ if $D = (B \cup \{i\})^{\downarrow_{i+1}\uparrow_{i+1}}$ and thus $C = A \cap \{i\}^{\downarrow_{i+1}}$. A canonical generator $\langle A,B\rangle$ of $\langle C,D\rangle$ is then the infimum

$$\langle A,B\rangle = \left\langle \bigcap_{j\in J} A_j, \left(\bigcup_{j\in J} B_j\right)^{\downarrow_i\uparrow_i} \right\rangle$$

of all generators $\langle A_j,B_j\rangle \in \mathcal{B}_i(X,Y_i,I_i)$ $(j \in J)$ of $\langle C,D\rangle$. Then, the authors of [27] utilize the fact that if $\langle A,B\rangle$ is a canonical generator of a new concept $\langle E,F\rangle$ and $\langle C,D\rangle$ is a non-canonical generator of $\langle E,F\rangle$ then any concept $\langle G,H\rangle$ such that $D \subset H$ and $B \not\subseteq H$ is not a canonical generator of any new concept, cf. [27, Proposition 1]. As a consequence, AddIntent does not have to process concepts like $\langle G,H\rangle$ during the search for canonical generators. On one hand, this is an analogous improvement like that proposed in this paper, see Lemma 2. On the other hand, improvements in both AddIntent and FCbO are based on different notions of canonicity (we do not use the lattice order $\leqslant$) and different approaches (incremental and non-incremental) of computing formal concepts.

4. Complexity and efficiency issues

It is a well-known fact that the limiting factor of computing all formal concepts is that the corresponding counting problem is #*P*-complete [18,20]. Fortunately, if $|I|$ is considerably small, one can get the set of all formal concepts in reasonable time even if X and Y are large. Therefore, there have been proposed various algorithms for FCA specialized on sparse incidence data. FCbO performs well in case of both sparse and dense data of reasonable size. From the point of view of the asymptotic worst-case complexity, FCbO has time delay $O(|Y|^3 \cdot |X|)$, see [14], and asymptotic time complexity $O(|\mathcal{B}(X,Y,I)| \cdot |Y|^2 \cdot |X|)$ because in the worst case, FCbO can degenerate into the original CbO [19,21] but in general, it cannot do worse than CbO. In addition, there are strong indications that on average FCbO delivers the results faster than CbO. Therefore, the average-case complexity analysis of FCbO and ramifications of the worst-case complexity of FCbO seem to be challenging and important *open problems.*

In this section we focus on two aspects of FCbO. First, we discuss suitable data representation for efficient computation of closures of $^{\downarrow\uparrow}$ which can be used by both CbO and FCbO including their derivatives like PCbO [15]. The second subsection presents an experimental evaluation of FCbO performance on real and synthesized data sets. The observations made therein illustrate the average-case behavior of the algorithm.

4.1. Improving efficiency of computing closures

The input formal context $I \subseteq X \times Y$ is usually represented in a computer by a two-dimensional array which corresponds with I in the obvious way. We suggest to represent I by a collection of sets representing all rows of such two-dimensional array. By a row (corresponding to $x \in X$) we mean a set of attributes $\{x\}^{\uparrow} = \{y \in I | \langle x,y \rangle \in I\}$. Clearly, if we take set $\mathcal{O} = \{\{x\}^{\uparrow} | x \in X\}$, we have $\langle x,y \rangle \in I$ iff $y \in \{x\}^{\uparrow}$ iff $x \in \{y\}^{\downarrow}$.

Representing I by $\mathcal{O}$, we can significantly improve the computation of a new formal concept which is done in lines 8 and 9 of Algorithm 2. Given formal concept $\langle A,B \rangle$ and $j \notin B$, we can compute

$$\langle C,D \rangle = \left\langle A \cap \{j\}^{\downarrow}, (A \cap \{j\}^{\downarrow})^{\uparrow} \right\rangle = \left\langle A \cap \{j\}^{\downarrow}, (B \cup \{j\})^{\downarrow\uparrow} \right\rangle$$

as it is shown in Algorithm 3. Thus, lines 8 and 9 of Algorithm 2 can be replaced by a single call of ComputeClosure($\langle A,B \rangle, j$). The algorithm is correct. Indeed, it is evident that $C = A \cap \{j\}^{\downarrow}$ and $D = \bigcap_{x \in A \cap \{j\}^{\downarrow}} \{x\}^{\uparrow} = \left(\bigcup_{x \in A \cap \{j\}^{\downarrow}} \{x\}\right)^{\uparrow} = (A \cap \{j\}^{\downarrow})^{\uparrow} = (B \cup \{j\})^{\downarrow\uparrow}$.

Algorithm 3: Procedure ComputeClosure($\langle A,B \rangle, j$)

```
1  set C to ∅;
2  set D to Y;
3  foreach x in A do
4    if j ∈ {x}↑ then
5      set C to C ∪ {x};
6      set D to D ∩ {x}↑;
7    end
8  end
9  return ⟨C,D⟩
```

Remark 3. A straightforward method for computing new formal concepts is based on definitions (5) and (6) of the concept-forming operators. The method can be implemented by a direct two-way algorithm which first computes the extent $(B \cup \{j\})^{\downarrow}$ which is further used to compute the intent $(B \cup \{j\})^{\downarrow\uparrow}$. Contrary to that, the procedure ComputeClosure computes the extent by filtering out the objects x from A for which it does not hold $j \in \{x\}^{\uparrow}$. In addition to that, during the computation of an extent, we also compute the corresponding intent by computing intersections of D and rows $\{x\}^{\uparrow}$. This can be done more efficiently especially if $\{x\}^{\uparrow}$ are organized as bit arrays. Thus, the algorithm relies on efficient implementation of sets and a single operation on sets: the intersection. Since computing intersections is generally more efficient than implementing the concept-forming operators, Algorithm 3 significantly outperforms the naive two-way algorithm. A detailed comparison of various data structures used for computing formal concepts can be found in [17]. ■

4.2. Experimental evaluation

We have run several experiments to compare the algorithm with CbO [19,21], Andrews's In-Close [2] and Ganter's NextClosure [9]. For the sake of comparison, we have implemented our algorithm, CbO and NextClosure in ANSI C while the implementation of In-Close was borrowed from the author. As suggested in the previous section, we represented input data tables by set $\mathcal{O}$ of table rows. Sets of attributes were represented by bit-arrays, where each bit represented the presence/absence of an attribute in a set. When storing a bit-array as an array of 32-bit or 64-bit integers, depending on the hardware architecture, all of the set operations with attributes, especially the set intersection, can be implemented by bitwise operations "and", "not", and "xor" on integers. These operations are implemented in arithmetic logic units (ALUs) of all computer processors. This representation is beneficial, e.g., in Algorithm 3 in line 6, where we can process up to 32 or 64 attributes at a time.

The experiments were run on otherwise idle 32-bit i386 hardware (Intel Core 2 Duo T9600, 2.8 GHz, 4 GB RAM). We performed two types of experiments. First, we were interested in the performance of all four algorithms measured by running time. Second, and more importantly, in order to evaluate the influence of the new canonicity test, we compared FCbO and CbO in terms of the total number of computed closures.

In the first set of experiments, we have run the algorithms on randomly generated data tables with various percentages of 1's in the table. We have used tables with 10,000 objects and the number of attributes ranging from 50 to 200 attributes. To illustrate the performance of algorithms, Fig. 4(left) shows a graph of dependency of time required to compute all formal concepts on the number of attributes in data tables with 10% of nonzero entries. We have not depicted the graph of average running time of NextClosure since there is a huge performance gap between the algorithm and the other algorithms (for instance FCbO is approximately 100 times faster than NextClosure on the evaluated data); the solid line is for FCbO, the dashed line is for CbO and the dotted line is for In-Close. In the FCbO/CbO comparison, the graph illustrating the average numbers of computed closures is depicted in Fig. 5(left); again, the solid line is for FCbO and the dashed line for CbO. Note that the graph

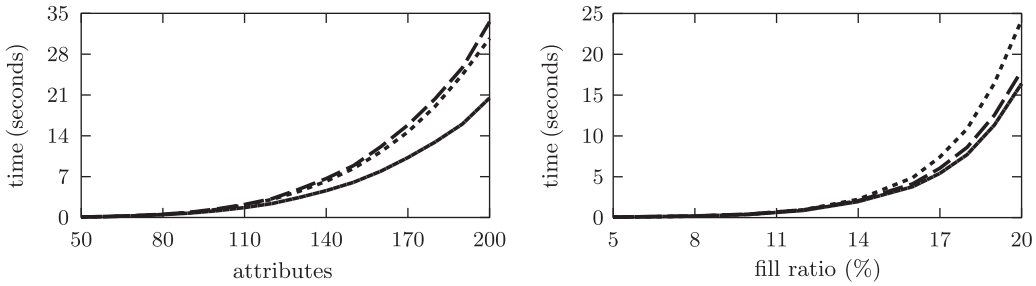

Fig. 4. Average running time dependent on number of attributes (on the left) and on fill ratio (density of 1's, on the right), solid line – FCbO, dashed line – CbO, dotted line – In-Close.

actually depicts the numbers divided by average concept lattice size (i.e., the number of closures which pass both the new, in case of FCbO, and the original canonicity test). Furthermore, to illustrate the influence of the fill ratio (density of 1's) of a data table on the speed of the algorithms and on the number of computed closures, we have included Figs. 4 and 5(right) which show graphs of dependencies on the fill ratios.

The second set of experiments were done with several data sets from the UCI Machine Learning Repository [4,13]. The results for performance times and numbers of computed closures are depicted in Fig. 6, along with the information on size and fill ratio of used data sets and concept lattice size.

From all the time and closure number dependency graphs and the table we can see that the FCbO algorithm significantly outperforms NextClosure and also considerably outperforms both the CbO and In-Close algorithms. In both cases, the performance gain is due to the new canonicity test which avoids a large number of concepts to be computed multiple times (cf. the numbers of closures computed by FCbO and CbO in case of the MUSHROOM data set, for instance, in Fig. 6). In case of NextClosure [9], the performance gain is then further multiplied by a more efficient computation of closures described in Section 4.1. The efficiency of the new "fast" test is illustrated by the graphs and the table depicting the numbers of closures computed by CbO (and NextClosure) and by FCbO.

5. Conclusions

We have introduced an algorithm called FCbO for computing formal concepts in object-attribute data tables. The algorithm results from CbO [19,21] by introducing a new canonicity test. We have proved correctness of the algorithm and pre-

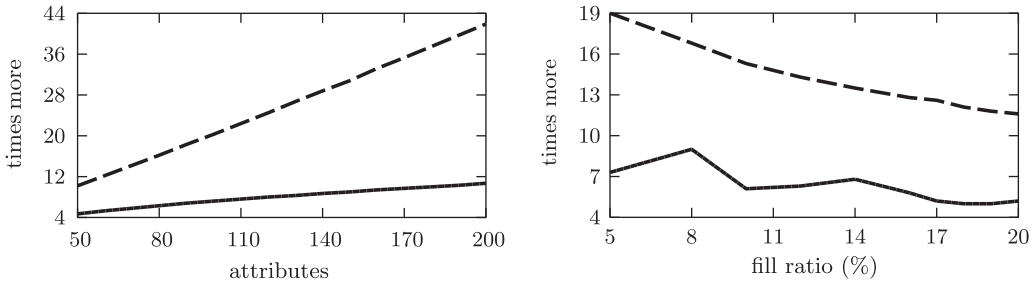

Fig. 5. Ratio of average number of closures computed by FCbO (solid line) and by CbO (dashed line) to average concept lattice size dependent on number of attributes (on the left) and on fill ratio (density of 1's, on the right).

dataset	mushroom	anonymous web	adult	internet ads
size	8124 × 119	32711 × 296	48842 × 104	3279 × 1557
fill ratio	19.33 %	1.02 %	8.65 %	0.88 %
#concepts	238710	129009	180115	9192
NextClosure time	53.891	243.325	134.954	114.493
CbO time	0.508	0.238	0.302	0.332
In-Close Time	0.544	0.524	0.302	0.158
FCbO time	0.340	0.240	0.318	0.160
#closures computed by CbO	1321411	785112	585253	1783871
#closures computed by FCbO	299202	398148	305644	309357

Fig. 6. Performance (in seconds) and numbers of closures computed by CbO and FCbO for selected datasets.

sented an experimental evaluation of its performance compared to the original CbO, Ganter's NextClosure and also to Andrews's In-Close, another contemporary derivative of CbO. The experiments have shown that FCbO significantly reduces the number of computed closures while maintaining a resonable overhead and hence delivers results faster than the other algorithms. The implementation of the algorithm can be downloaded from

http://fcalgs.sourceforge.net/fcbo-ins.html.

The future research will focus on further refinements and extensions of the algorithm and will focus in a more detail on the relationship between various recently-developed algorithms [2,27].

Acknowledgment

Supported by Grant Nos. P103/10/1056 and P202/10/P360 of the Czech Science Foundation and by Grant No. MSM 6198959214. The algorithm described in this paper has been presented during ICCS 2009 and CLA 2010 [16] conferences. However, in ICCS 2009, the algorithm took part in a performance contest only and was not published in the proceedings, and the CLA 2010 paper contains the pseudocode and a brief summary of the algorithm without further analysis or proofs. The present paper is aimed as a detailed explanation of the algorithm with emphasis on the canonicity test.

References

[1] R. Agrawal, T. Imielinski, A.N. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM International Conference of Management of Data, 1993, pp. 207–216.
[2] S. Andrews, In-Close, a Fast Algorithm for Computing Formal Concepts, in: S. Rudolph, F. Dau, S.O. Kuznetsov (Eds.): *Supplementary Proceedings of ICCS '09*, CEUR WS, vol. 483, 2009, p. 14. <http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-483/paper1.pdf>.
[3] F. Angiulli, E. Cesario, C. Pizzuti, Random walk biclustering for microarray data, Information Sciences 178 (6) (2008) 1479–1497.
[4] A. Asuncion, D. Newman, UCI Machine Learning Repository, University of California, Irvine, School of Information and Computer Sciences, 2007.
[5] R. Belohlavek, Lattices of fixed points of fuzzy Galois connections, Math. Logic Quarterly 47 (1) (2001) 111–116.
[6] R. Belohlavek, E. Sigmund, J. Zacpal, Evaluation of IPAQ questionnaires supported by formal concept analysis, Inform. Sci. 181 (10) (2011) 1774–1786.
[7] R. Belohlavek, V. Vychodil, Discovery of optimal factors in binary data via a novel method of matrix decomposition, J. Comput. Syst. Sci. 76 (1) (2010) 3–20.
[8] J.H. Correia, G. Stumme, R. Wille, U. Wille, Conceptual knowledge discovery – A human-centered approach, Appl. Artif. Intell. 17 (3) (2003) 281–302.
[9] B. Ganter, Two basic algorithms in concept analysis. (Technical Report FB4-Preprint No. 831). TH Darmstadt, 1984.
[10] B. Ganter, R. Wille, Formal Concept Analysis, Mathematical Foundations, Springer, Berlin, 1999.
[11] L.A. Goldberg, Efficient Algorithms for Listing Combinatorial Structures, Cambridge University Press, 1993.
[12] G.A. Gratzer, General Lattice Theory, 2nd ed., Birkhauser, 1998.
[13] S. Hettich, S.D. Bay, The UCI KDD Archive University of California, Irvine, School of Information and Computer Sciences, 1999.
[14] D.S. Johnson, M. Yannakakis, C.H. Papadimitriou, On generating all maximal independent sets, Inform. Process. Lett. 27 (3) (1988) 119–123.
[15] P. Krajca, J. Outrata, V. Vychodil, Parallel algorithm for computing fixpoints of galois connections, Ann. Math. Artif. Intell. 59 (2) (2010) 257–272.
[16] P. Krajca, J. Outrata, V. Vychodil, Advances in algorithms based on CbO, in: Proceedings of the CLA, 2010, pp. 325–337.
[17] P. Krajca, V. Vychodil, Comparison of data structures for computing formal concepts, in: Proc. MDAI, LNCS 5861, 2009, pp. 114–125.
[18] S.O. Kuznetsov, Interpretation on graphs and complexity characteristics of a search for specific patterns, Automat. Document. Math. Linguist. 24 (1) (1989) 37–45.
[19] S.O. Kuznetsov, A fast algorithm for computing all intersections of objects in a finite semi-lattice, Automat. Document. Math. Linguist. 27 (5) (1993) 11–21.
[20] S.O. Kuznetsov, On computing the size of a lattice and related decision problems, Order 18 (2001) 313–321.
[21] S.O. Kuznetsov, Learning of simple conceptual graphs from positive and negative examples, PKDD (1999) 384–391.
[22] S.O. Kuznetsov, S.A. Obiedkov, Comparing performance of algorithms for generating concept lattices, J. Exp. Theor. Artif. Int. 14 (2002) 189–216.
[23] W. Kneale, M. Kneale, The Development of Logic, Oxford University Press, USA, 1985.
[24] C. Lindig, Fast concept analysis, Working with Conceptual Structures – Contributions to ICCS 2000, Shaker Verlag, Aachen, 2000.
[25] H. Liu, X. Wang, J. He, J. Han, D. Xin, Z. Shao, Top–down mining of frequent closed patterns from very high dimensional data, Inform. Sci. 179 (7) (2009) 899–924.
[26] J. Medina, M. Ojeda-Aciego, Multi-adjoint t-concept lattices, Inform. Sci. 180 (5) (2010) 712–725.
[27] D. van der Merwe, S.A. Obiedkov, D.G. Kourie, AddIntent: A New Incremental Algorithm for Constructing Concept Lattices, in: Proceedings of the ICFCA 2004, LNAI 2961, 2004, pp. 205–206.
[28] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, H. Mannila, The Discrete Basis Problem, PKDD, Springer, 2006.
[29] B. Mirkin, Mathematical Classification and Clustering, Kluwer Academic Publishers, 1996.
[30] E.M. Norris, An algorithm for computing the maximal rectangles in a binary relation, Revue Roumaine de Mathématiques Pures et Appliquées 23 (2) (1978) 243–250.
[31] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, ACM Trans. Program. Lang. Syst. 22 (3) (2000) 540–582.
[32] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, IEEE Trans. Softw. Eng. 29 (6) (2003) 495–509.
[33] R. Wille, Restructuring lattice theory: an approach based on hierarchies of concepts, Ordered Sets, Dordrecht-Boston, 1982, pp. 445–470.
[34] M.J. Zaki, Mining non-redundant association rules, Data Min. Knowledge Discov. 9 (2004) 223–248.