LCM is well implemented CbO: study of LCM from FCA point of view

Radek Janostik, Jan Konecny, and Petr Krajča

Dept. Computer Science, Palacký University Olomouc
17. listopadu 12, CZ–77146 Olomouc, Czech Republic
{radek.janostik, jan.konecny, petr.krajca}@upol.cz

Abstract. LCM is an algorithm for enumeration of frequent closed itemsets in transaction databases. It is well known that when we ignore the required frequency, the closed itemsets are exactly intents of formal concepts in Formal Concept Analysis (FCA). We describe LCM in terms of FCA and show that LCM is basically the Close-by-One algorithm with multiple speed-up features for processing sparse data. We analyze the speed-up features and compare them with those of similar FCA algorithms, like FCbO and algorithms from the In-Close family.

Keywords: algorithm; formal concept analysis; frequent closed itemset; close-by-one

1 Introduction

Frequent closed itemsets in transaction databases are exactly intents in Formal Concept Analysis (FCA) with sufficient support—cardinality of the corresponding extents. If the minimum required support is zero (i.e. any attribute set is considered frequent), one can easily unify these two notions. LCM (**L**inear time **C**losed itemset **M**iner) is an algorithm for the enumeration of frequent closed itemsets developed by Takeaki Uno [15,16,17,18] in 2003–2005. It is considered to be one of the most efficient algorithms for this task.[1]

We have thoroughly studied Uno's papers and source codes and, in the present paper, we deliver a complete description of LCM from the point of view of FCA. Despite the source codes being among the main sources for this study, we stay at a very comprehensible level in our description and avoid delving into implementation details. We explain that the basis of LCM is Kuznetsov's Close-by-One (CbO) [11].[2] We describe its additional speed-up features and compare them with those of state-of-art CbO-based algorithms, like FCbO [14] and In-Close2+ [4,5,6,7].[3]

[1] Its implementations with source codes are available at URL http://research.nii.ac.jp/~uno/codes.htm.

[2] Although LCM was most likely developed independently.

[3] In the rest of this paper, whenever we write 'CbO-based algorithms' we mean CbO, FCbO and In-Close family of algorithms. By version number 2+, we mean the version 2 and higher.

 Published in Francisco J. Valverde-Albacete, Martin Trnecka (Eds.): Proceedings of the 15th International Conference on Concept Lattices and Their Applications, CLA 2020, pp. 47–58, 2020.

Remark 1 (Some subjective notes on our motivations). Besides the obvious importance of LCM for FCA[4], there are two motivational points for this work. We separated them into this remark as they are based on our subjective impressions and views. We ask the reader to excuse the rather subjective tone in this remark.

(a) A significant part of the FCA community is aware of LCM and uses Uno's implementation to enumerate formal concepts for further processing or for comparison with their methods. However, it is our impression that, more often than not, the implementation is used merely as a black box. We believe that this part of the FCA community would appreciate 'unwrapping the black box'.
(b) Uno's papers provide quite confusing descriptions of LCM and the source codes of the implementations are not written to allow easy analysis. Moreover, Uno's implementation and description of LCM have some differences; there is even an important feature present in the implementation which is not described in the papers. We believe that a clearer and more complete description would be fruitful.

The paper is structured as follows. First, we recall the basic notions and notations of FCA (Section 2) we use in the rest of this paper. Then we describe CbO (Section 3) as it is a basis for description of LCM. Afterwards, we provide a description of LCM's features (Section 4), namely, initial preprocessing of data (Section 4.1), handling data using arraylists computing all attribute extents at once (Section 4.2), conditional datasets (Section 4.3), and pruning (Section 4.4). In all of the above, we ignore the requirement of frequency and we describe it separately (Section 5). Finally, we summarize our conclusions (Section 6).

2 Formal Concept Analysis

An input to FCA is a triplet $\langle X, Y, I \rangle$, called a *formal context*, where X and Y are non-empty sets of objects and attributes, respectively, and I is a binary relation between X and Y; $\langle x, y \rangle \in I$ means that the object x has the attribute y. Finite formal contexts are usually depicted as tables, in which rows represent objects, columns represent attributes, and each entry contains a cross if the corresponding object has the corresponding attribute, and is left blank otherwise. In this paper, we consider only finite formal contexts.

The formal context induces concept-forming operators:

$^{\uparrow} : \mathbf{2}^X \to \mathbf{2}^Y$ assigns to a set A of objects the set $A^{\uparrow}$ of all attributes shared by all the objects in A.
$^{\downarrow} : \mathbf{2}^Y \to \mathbf{2}^X$ assigns to a set B of attributes the set $B^{\downarrow}$ of all objects which share all the attributes in B.

[4] However, the closed sets are also relevant in other disciplines, like association rule mining [1,8], condensed representation of inductive queries [13], or logical analysis of data [2,3].

Formally,

$$A^{\uparrow} = \{y \in Y \mid \forall x \in A : \langle x, y\rangle \in I\}, \quad \text{and} \quad B^{\downarrow} = \{x \in X \mid \forall y \in B : \langle x, y\rangle \in I\}.$$

For singletons, we use shortened notation and write $x^{\uparrow}$, $y^{\downarrow}$ instead of $\{x\}^{\uparrow}$, $\{y\}^{\downarrow}$, respectively.

In this paper, we assume a set of attributes $Y = \{1, \ldots, n\}$. Whenever we write about lower and higher attributes, we refer to the natural ordering $\leqslant$ of the numbers in Y.

A *formal concept* is a pair $\langle A, B\rangle$ of sets $A \subseteq X, B \subseteq Y$, such that $A^{\uparrow} = B$ and $B^{\downarrow} = A$. The first component of a formal concept is called the extent, whilst the second one is called the intent.

The compositions $^{\uparrow\downarrow}$ and $^{\downarrow\uparrow}$ of concept-forming operators are closure operators on 2^X and 2^Y, respectively. That is, the composition $^{\downarrow\uparrow}$ satisfies

$$\text{(extensivity)} \qquad B \subseteq B^{\downarrow\uparrow} \tag{1}$$

$$\text{(monotony)} \qquad B \subseteq D \text{ implies } B^{\downarrow\uparrow} \subseteq D^{\downarrow\uparrow} \tag{2}$$

$$\text{(idempotency)} \qquad B^{\downarrow\uparrow} = B^{\downarrow\uparrow\downarrow\uparrow} \tag{3}$$

for all $B, D \subseteq Y$ (analogously for the composition $^{\uparrow\downarrow}$).

Sets of attributes satisfying $B = B^{\downarrow\uparrow}$ are called closed sets and they are exactly the intents of formal concepts.

3 Close-by-One

In the context of FCA, the foundation of LCM is CbO.[5] Therefore, we firstly turn our attention to CbO.

We start the description of CbO with a naïve algorithm for generating all closed sets. The naïve algorithm traverses the space of all subsets of Y, each subset is checked for closedness and is outputted. This approach is quite inefficient as the number of closed subsets is typically significantly smaller than the number of all subsets.

The algorithm is given by a recursive procedure `GenerateFrom`, which accepts two attributes:

- B – the set of attributes, from which new sets will be generated.
- y – the auxiliary argument to remember the highest attribute in B.

The procedure first checks the input set B for closedness and prints it if it is closed (lines 1,2). Then, for each attribute i higher than y:

[5] It is called a backtracking algorithm with prefix-preserving closure extensions in Uno's papers.

Algorithm 1: Naïve algorithm to enumerate closed subsets

```
   def GenerateFrom(B, y):
   |   input  : B – set of attributes
   |             y – last added attribute
1  |   if B = B^{↓↑} then
2  |   |   print(B)
3  |   for i ∈ {y + 1, . . . , n} do
4  |   |   D ← B ∪ {i}
5  |   |   GenerateFrom(D, i)
6  |   return
   GenerateFrom(∅, 0)
```

- a new set is generated by adding the attribute i into the set B (line 4);
- the procedure recursively calls itself to process the new set (line 5).

The procedure is initially called with an empty set and zero as its arguments.

The naïve algorithm represents a depth-first sweep through the tree of all subsets of Y (see Fig. 1) and printing the closed ones.

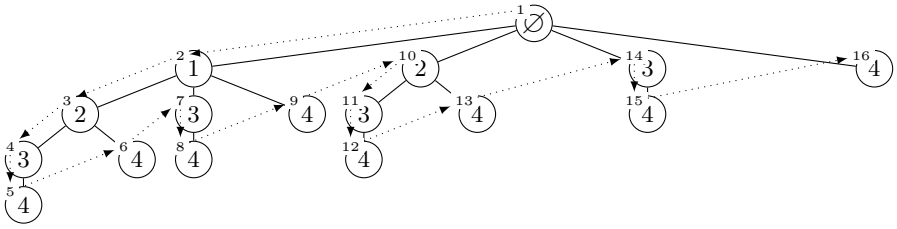

Fig. 1. Tree of all subsets of $\{1, 2, 3, 4\}$. Each node represents a unique set containing all elements in the path from the node to the root. The dotted arrows and small numbers represent the order of traversal performed by the algorithm for generating all subsets.

In the tree of all subsets (Fig. 1), each node is a superset of its predecessors. We can use the closure operator $^{\downarrow\uparrow}$ to skip non-closed sets. In other words, to make jumps in the tree to closed sets only. Instead of simply adding an element to generate a new subset $D \leftarrow B \cup \{i\}$, CbO adds the element and then closes the set

$$D \leftarrow (B \cup \{i\})^{\downarrow\uparrow}. \tag{4}$$

We need to distinguish the two outcomes of the closure (4). Either

- the closure contains some attributes lower than i which are not included in B, i.e. $D_i \neq B_i$ where $D_i = D \cap \{1, \ldots, i-1\}$, $B_i = B \cap \{1, \ldots, i-1\}$;
- or it does not, and we have $D_i = B_i$.

The jumps with $D_i \neq B_i$ are not desirable because they land on a closed set which was already processed or will be processed later (depending on the direction of the sweep). CbO does not perform such jumps. The check of the condition $D_i = B_i$ is called a *canonicity test*.

Algorithm 2: Close-by-One

```
   def GenerateFrom(A, B, y):
   |   input : A – extent
   |           B – set of attributes
   |           y – last added attribute
1  |   D ← A↑
2  |   if D_y ≠ B_y then
3  |   |   return
4  |   print(⟨A, D⟩)
5  |   for i ∈ {y + 1, ..., n} \ D do
6  |   |   C ← A ∩ i↓
7  |   |   GenerateFrom(C, D ∪ {i}, i)
8  |   return
   GenerateFrom(X, ∅, 0)
```

One can see the pseudocode of CbO in Algorithm 2. Firstly, we additionally pass an extent A to the procedure `GenerateFrom` and the closed set $B^{\downarrow\uparrow}$ is computed as $A^{\uparrow}$ (line 1), which is more efficient. Then the canonicity test is performed. If it fails, we end the procedure (lines 2, 3). Otherwise, we print the the concept and continue with the generation of its subtree.

Remark 2. What we show in Algorithm 2 is not the usual pseudocode of CbO presented in literature. We made some superficial changes to emphasize the link between CbO and LCM, which will be apparent later. Specifically:

(a) The closure and the canonicity test (lines 1–3) are usually performed before the recursive invocation (line 7), not as a first steps of the procedures.
(b) The main loop (line 5) of CbO usually processes the attributes in ascending order, which corresponds to a left-to-right depth-first sweep through the tree of all subsets (Fig. 1). In actual fact, for CbO there is no reason for a particular order of processing attributes.

4 Features of LCM

There are three versions of the LCM algorithm:

LCM1 is CbO with arraylist representation of data and computing of all extents at once (described in Section 4.2), data preprocessing (described in Section 4.1), and using of diffsets [19] to represent extents for dense data (this is not present in later versions).

LCM2 is LCM1 (without diffsets) with conditional databases (described in Section 4.3)

LCM3 is LCM2 which uses a hybrid data structure to represent a context. The data structure uses a combination of FP-trees and bitarrays, called a complete prefix tree, to handle the most dense attributes. Arraylists are used for the rest, the same way as in the previous versions.

In this paper, we describe all features present in LCM2.

4.1 Initialization

To speed the computation up, LCM initializes the input data as follows:

- removes empty rows and columns,
- merges identical rows,
- sorts attributes by cardinality ($|y^{\downarrow}|$) in descending order,
- sorts objects by cardinality ($|x^{\uparrow}|$) in descending order.

In the pseudocode in Algorithm 3 (later in the paper), the initialization is not shown and it is supposed that it is run before the first invocation of the procedure `GenerateFrom`.

FCA aspect: The attribute sorting is well known to most likely cause a smaller number of computations of closures in CbO-based algorithms [9,4,5]. This feature is included in public implementations of In-Close4 and FCbO.

The object sorting is a different story. Andrews [4] tested the performance of In-Close2 and concluded that lexicographic order tends to significantly reduce L1 data cache misses. However, the test were made for bitarray representation of contexts.

The reason for object sorting in LCM is probably that a lesser amount of inverses occurs in a computation of a union of rows (shown later (5)), which is consequently easier to sort. Our testing with Uno's implementation of LCM did not show any difference in runtime for unsorted and sorted objects when attributes are sorted. In the implementation of LCM3, the object sorting is not present.

Remark 3. In examples in the present paper, we do not use sorted data, in order to keep the examples small.

4.2 Ordered arraylists and occurrence deliver

LCM uses arraylists[6] as data representation of the rows of the context. It is directly bound to one of LCM's main features – *occurrence deliver*:

LCM computes extents $A \cap i^{\downarrow}$ (line 6 in Algorithm 2) all at once using a single traversal through the data. Specifically, it sequentially traverses through

[6] Whenever we write arraylist, we mean ordered arraylists.

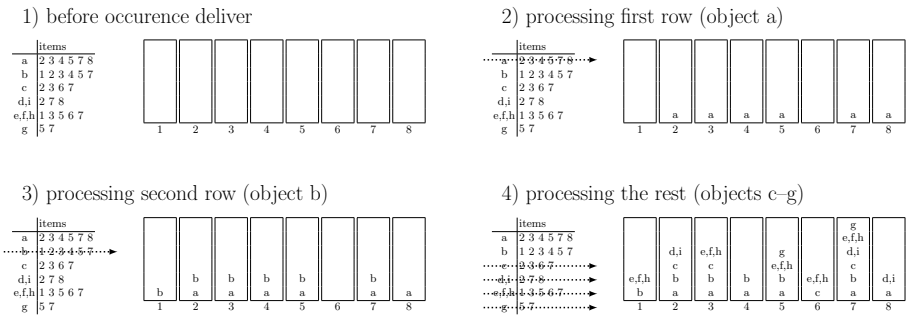

Fig. 2. Occurrence deliver in LCM.

all rows $x^{\uparrow}$ of the context and whenever it encounters an attribute i, it adds x to an initially empty arraylist – *bucket* – for i (see Fig. 2). As LCM works with conditional datasets (see Section 4.3), attribute extents correspond to extents $A \cap i^{\downarrow}$ (see Algorithm 2, line 6).

LCM generates childs of each node from right to left. That way, it can reuse the memory for extents (buckets). For example, when computing extents in the node $\{2\}$, that is $\{2,3\}^{\downarrow}$ and $\{2,4\}^{\downarrow}$, the algorithm can reuse the memory used by extents $\{3\}^{\downarrow}$ and $\{4\}^{\downarrow}$, because $\{3\}$ and $\{4\}$ (and their subtrees) are already finalized (see Fig. 3).

FCA aspect: In FCA, the CbO-based algorithms do not specify data representation used for handling contexts, sets of objects, and sets of attributes. This is mostly considered a matter of specific implementations (see Remark 4). Generally, the data representation issues are almost neglected in literature on FCA. The well-known comparison study [12] of FCA algorithms mentioned the need to study the influence of data structures on practical performances of FCA algorithms but it does not pay attention to that particular issue. The comparison

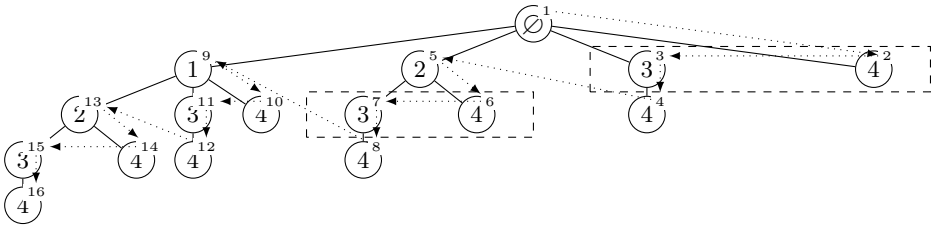

Fig. 3. Demonstration of bucket reuse in LCM with right-first sweep.

study [10] provided the first steps to an answer for this need.[7] The latter paper concludes that binary search trees or linked lists are good choices for large or sparse datasets, while bitarray is an appropriate structure for small or dense datasets. Arraylists did not perform particularly well in any setting. However, this comparison did not assume other features helpful for this data representation, like conditional databases (see Section 4.3) and computation of all required attribute extents in one sweep by occurrence deliver. More importantly, the minimal tested density is 5 %, which is still very dense in the context of transactional data.

Remark 4. Available implementations of FCbO[8] and In-Close[9] utilize bitarrays for rows of contexts, and sets of attributes, and arraylists for sets of objects.

4.3 Conditional Database and Interior Intersections

LCM reduces the database for the recursive invocations of `GenerateFrom` to so-called conditional databases.

Let $\mathcal{K} = \langle X, Y, I \rangle$ be a formal context, B be an intent, and y be the attribute used to build B. The conditional database (context) $\mathcal{K}_{B,y}$ w.r.t. $\langle B, y \rangle$ is created from $\mathcal{K}$ as follows:

(a) First, remove objects from $\mathcal{K}$ which are not in the corresponding extent $A = B^{\downarrow}$.
(b) Remove attributes which are full or empty.
(c) Remove attributes lesser than y. [10]
(d) Merge identical objects together.
(e) Put back attributes removed in step (c); set each new object created in step (d) by merging objects $x_1, \ldots, x_k \in X$ to have attributes common to the objects $x_1, \ldots, x_k$. That is, the merged objects are intersections of the original objects. The part of the context added in this step is called an interior intersection.

Alternatively, we can describe conditional databases with interior intersections as:

- Restricting the context $\mathcal{K}$ to objects in A and attributes in N where

$$N = \left(\bigcup_{x \in A} x^{\uparrow} \right) \setminus A^{\uparrow}. \tag{5}$$

  This covers the steps (a)–(c).

[7] Paper [10] compares bitarrays, sorted linked lists, arraylists, binary search trees, and hash tables.
[8] Available at `http://fcalgs.sourceforge.net/`.
[9] Available at `https://sourceforge.net/projects/inclose/`.
[10] In the implementation, when the database is already too small (less than 6 objects, and less than 2 attributes), steps (c)–(d) are skipped.

- Subsequent merging/intersecting of those objects which have the same incidences with attributes in $\{1, 2, \ldots, y-1\}$. This covers the steps (d)–(e).

Note, that from the conditional database (with interior intersections) $\mathcal{K}_{B,i}$ all intents in the subtree of B can be extracted. More specifically, if D is an intent of the conditional database and it passes the canonicity test (tested on interior intersections) then $D \cup B$ is an intent of the original context which passes the canonicity test; i.e. is in the subtree of B.

In pseudocode in Algorithm 3 (later in the paper), the creation of the conditional databases with interior intersections is represented by the procedure named `CreateConditionalDB`$(\mathcal{K}, A, N, y)$.

FCA aspect: CbO-based algorithms do not utilize conditional databases. However, we can see partial similarities with features of CbO-based algorithms.

First, all of the algorithms skip attributes in B and work only with part of the formal context given by $B^{\downarrow}$ and $Y \setminus B$. This corresponds to step (a) and the first part of step (b) (full attributes).

Second, the removal of empty attributes in step (b) utilizes basically the same idea as in In-Close4 [6]: if the present extent A and an attribute extent $i^{\downarrow}$ have no common object, we can skip the attribute i in the present subtree. In FCbO and In-Close3, such attribute would be skipped due to pruning (see Section 4.4).

Steps (c)–(e) have no analogy in CbO algorithms.

Pseudocode of LCM without pruning

At this moment, we present pseudocode of LCM (Algorithm 3) with above-described features. As in the case for CbO, the algorithm is given by recursive procedure `GenerateFrom`. The procedure takes four arguments: an extent A, a set of attributes B, the last attribute y added to B, and a (conditional) database $\mathcal{K}$). The procedure performs the following steps:

(line 1) The set N (5) of non-trivial attributes is computed.
(line 2) The frequencies of all attributes in N are computed, this is made by a single traversal through $\mathcal{K}$ similar to the occurence deliver (described in Section 4.2).
(lines 3–5) The loop checks whether any attribute in N lesser than y has frequency equal to $|A|$. If so, the attribute causes the canonicity test to fail, therefore we end the procedure.
(lines 6–9) The loop closes B (and updates N) based on the computed frequencies.
(line 10) As the cannonicity is checked and B is closed, we can output $\langle A, B \rangle$.
(line 11) The conditional database $\mathcal{K}_{B,y}$ (described in Sec. 4.3) is created.
(line 12) Attribute extents from $\mathcal{K}_{B,y}$ are computed using occurence deliver (described in Section 4.2).

Algorithm 3: LCM (without pruning)

```
def GenerateFrom(A, B, y, 𝒦):
    input : A – extent
            B – set of attributes
            y – last added attribute
            𝒦 – input context/conditional database
1   N ← (⋃_{x∈A} x^↑) \ B
2   {n_i | i ∈ N} ←Frequencies(𝒦, N)
3   for i ∈ N, i < y do
4       if n_i = |A| then
5           return
6   for i ∈ N, i > y do
7       if n_i = |A| then
8           B ← B ∪ {i}
9           N ← N \ {i}
10  print(⟨A, B⟩)
11  𝒦' ← CreateConditionalDB(𝒦, A, N, y)
12  {C_i | i ∈ N} ← OccurenceDeliver(𝒦')
13  for i ∈ N, i > y, (in descending order) do
14      GenerateFrom(C_i, B ∪ {i}, i, 𝒦')
15  return

GenerateFrom(X, X^↑, 0, ⟨X, Y, I⟩)
```

(lines 13–14) The procedure `GenerateFrom` is recursively called for attributes in N with the conditional database $\mathcal{K}_{B,y}$ and the corresponding attribute extent.

4.4 Bonus Feature: Pruning

The jumps using closures in CbO significantly reduce the number of visited nodes in comparison with the naïve algorithm. The closure, however, becomes the most time consuming operation in the algorithm. The pruning technique in LCM[11] avoids computations of some closures based on the monotony property: for any set of attributes $B, D \subseteq Y$ satisfying $B \subseteq D$, we have

$$j \in (B \cup \{i\})^{\downarrow\uparrow} \quad \text{implies} \quad j \in (D \cup \{i\})^{\downarrow\uparrow}. \tag{6}$$

When $i, j \notin D$ and $j < i$, the implication (6) says that if j causes $(B \cup \{i\})^{\downarrow\uparrow}$ to fail the canonicity test then it also causes $(D \cup \{i\})^{\downarrow\uparrow}$ to fail the canonicity test. That is, if we store that $(B \cup \{i\})^{\downarrow\uparrow}$ failed, we can use it to skip computation of the closure $(D \cup \{i\})^{\downarrow\uparrow}$ for any $D \supset B$ with $j \notin D$.

[11] Pruning is not described in papers on LCM, however, it is present in the implementation of LCM2.

Due to the page limit, we skip details here.

FCA aspect: Similar pruning is also present in FCbO and In-Close3+. LCM's pruning is weaker than the pruning in FCbO and In-Close3, stronger than the pruning in In-Close4, and incomparable with pruning in In-Close5.

5 Frequency Counting

In previous sections, we did not take into account the frequency of itemsets, as in FCA, the frequency is not usually assumed. However, the implementations of FCbO and In-Close4 allow us to pass minimum support as a parameter, and then enumerate only frequent intents.

The CbO-based algorithms utilize a simple *apriori* principle: if a set is infrequent then all its supersets are infrequent. That directly translates into the tree-like computation of the algorithms – if a node represents an infrequent set, then its subtree does not contain any frequent set.

In LCM, we make the following modification of the features described in Section 4.

Data representation – each arraylist is accompanied with weight, i.e. number of objects it corresponds to.

Initialization – additionally, infrequent attributes are removed and the weights of rows are set to reflect the number of merged rows (1 for unique rows).

Conditional databases – in step (b), infrequent attributes are removed as well; and in step (d), the weights of rows are updated.

6 Conclusions

We analyzed LCM from the point of view of FCA and concluded that it is a CbO-based algorithm with additional features directed towards processing sparse data. We also compared the additional features with those of FCbO and InClose2+.

Future research: We see two main directions for our upcoming research:

- The investigation of other algorithms for closed frequent itemset mining and putting them into context with FCA algorithms.
- Experimental evaluation of the incorporation of LCM's features in CbO-based algorithms; this could lead to fast implementations of the algorithms.

Acknowledgment: The authors acknowledge support by the grants:

- IGA 2018 of Palacký University Olomouc, No. IGA_PrF_2019_034,
- JG 2019 of Palacký University Olomouc, No. JG_2019_008.

References

1. Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.
2. Gabriela Alexe, Sorin Alexe, Tibérius O Bonates, and Alexander Kogan. Logical analysis of data–the vision of Peter L. Hammer. *Annals of Mathematics and Artificial Intelligence*, 49(1-4):265–312, 2007.

3. Gabriela Alexe and Peter L. Hammer. Spanned patterns for the logical analysis of data. *Discrete Applied Mathematics*, 154(7):1039–1049, 2006.
4. Simon Andrews. In-Close2, a high performance formal concept miner. In *Proceedings of the 19th International Conference on Conceptual Structures for Discovering Knowledge*, ICCS'11, pages 50–62, Berlin, Heidelberg, 2011. Springer-Verlag.
5. Simon Andrews. A 'Best-of-Breed' approach for designing a fast algorithm for computing fixpoints of Galois connections. *Information Sciences*, 295:633–649, 2015.
6. Simon Andrews. Making use of empty intersections to improve the performance of CbO-type algorithms. In *International Conference on Formal Concept Analysis*, pages 56–71. Springer, 2017.
7. Simon Andrews. A new method for inheriting canonicity test failures in Close-by-One type algorithms. In *CLA*, volume 2123, pages 255–266, 2018.
8. Roberto J. Bayardo Jr. Efficiently mining long patterns from databases. In *ACM Sigmod Record*, volume 27, pages 85–93. ACM, 1998.
9. Petr Krajca, Jan Outrata, and Vilem Vychodil. Advances in algorithms based on CbO. In *CLA*, volume 672, pages 325–337, 2010.
10. Petr Krajca and Vilem Vychodil. Comparison of data structures for computing formal concepts. In *International Conference on Modeling Decisions for Artificial Intelligence*, pages 114–125. Springer, 2009.
11. Sergei O. Kuznetsov. A fast algorithm for computing all intersections of objects from an arbitrary semilattice. *Nauchno-Tekhnicheskaya Informatsiya Seriya 2-Informatsionnye Protsessy i Sistemy*, (1):17–20, 1993.
12. Sergei O. Kuznetsov and Sergei Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14:189–216, 2002.
13. Heikki Mannila and Hannu Toivonen. Multiple uses of frequent sets and condensed representations. In *KDD*, volume 96, pages 189–194, 1996.
14. Jan Outrata and Vilem Vychodil. Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data. *Information Sciences*, 185(1):114–127, 2012.
15. Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *FIMI*, volume 90. Citeseer, 2003.
16. Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *International Conference on Discovery Science*, pages 16–31. Springer, 2004.
17. Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, volume 126, 2004.
18. Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 3: collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 77–86. ACM, 2005.
19. Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM, 2003.