A ‘Best-of-Breed’ approach for designing a fast algorithm for computing fixpoints of Galois Connections

ANDREWS, Simon <http://orcid.org/0000-0003-2094-7456>

Information Sciences 295 (2015) 633–649

Contents lists available at ScienceDirect

Information Sciences

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

A ‘Best-of-Breed’ approach for designing a fast algorithm for computing fixpoints of Galois Connections

Simon Andrews

Conceptual Structures Research Group, Communication and Computing Research Centre, Faculty of Arts, Computing, Engineering and Sciences, Sheffield Hallam University, Sheffield, UK

ARTICLE INFO

Article history:
Received 16 May 2012
Received in revised form 25 July 2014
Accepted 6 October 2014
Available online 12 October 2014

Keywords:
Fast CbO algorithm
Formal Concept Analysis
Fixpoints of Galois Connections
FCA
In-Close
Partial closure canonicity test

ABSTRACT

The fixpoints of Galois Connections form patterns in binary relational data, such as object-attribute relations, that are important in a number of data analysis fields, including Formal Concept Analysis (FCA), Boolean factor analysis and frequent itemset mining. However, the large number of such fixpoints present in a typical dataset requires efficient computation to make analysis tractable, particularly since any particular fixpoint may be computed many times. Because they can be computed in a canonical order, testing the canonicity of fixpoints to avoid duplicates has proven to be a key factor in the design of efficient algorithms. The most efficient of these algorithms have been variants of the Close-By-One (CbO) algorithm. In this article, the algorithms CbO, FCbO, In-Close, In-Close2 and a new variant, In-Close3, are presented together for the first time, with in-Close2 and In-Close3 being the results of breeding In-Close with FCbO. To allow them to be easily compared, the algorithms are presented in the same style and notation. The important advances in CbO are described and compared graphically using a simple example. For the first time, the algorithms are implemented using the same structures and techniques to provide a level playing field for evaluation. Their performance is tested and compared using a range of data sets and the most important features identified for a CbO ‘Best-of-Breed’. This article also presents, for the first time, the ‘partial-closure’ canonicity test.

1. Introduction

The emergence of Formal Concept Analysis (FCA) as a data analysis technique [5,31,16] has increased the need for algorithms that compute the fixpoints of Galois Connections quickly. These fixpoints are called formal concepts in FCA, and are the basis on which the Galois Connections form the line diagrams that are the fundamental visualisations for FCA. A problem in computing these fixpoints is the large number that can exist in a typical dataset. It is known that the number of concepts can be exponential in the size of the input context and the problem of determining this number is #P-complete [25]. Furthermore, computation of these fixpoints typically involves repeated computation of the same fixpoint [9]. When Ganter [12] determined how repeated computations of formal concepts could be detected using the natural lexical order of combinations, it was no longer necessary to search through previously generated concepts to determine the uniqueness of a newly generated one. If concepts were computed in lexical order it was possible to test the lexicographical rank of a concept to determine if it had already been computed. Kuznetsov specified in Close-by-One (CbO) an efficient means of achieving this called the ‘canonicty test’ [24,23] and this algorithm has formed the basis for several variants and suggested improvements,

E-mail address: s.andrews@shu.ac.uk

http://dx.doi.org/10.1016/j.ins.2014.10.011
0020-0255/

such as Krajca's realisation of CbO [18], the author's own In-Close algorithms [1,2] and Outrata and Vychodil's FCbO [20,29]. It has been shown that these CbO variants are significantly faster than previous algorithms and it is becoming accepted that they are the leading contenders for efficiency [23,18,1,20,30,2,17,29,7], outperforming algorithms including Chein [10], Norris [27], Next-Closure [12], Bordat [8], Godin [14], Nourine [28] and Add-Intent [32].

This article, for the first time, presents the CbO variants together, in a comparable manner, and implements them on a 'level playing field' to compare their performance. This article also presents a new CbO variant, In-Close3.

A key aspect of the efficiency of these CbO algorithms has been the design of the canonicity test. Some have required the complete closure of a concept before applying the test of canonicity [18] or have introduced a new test of inherited canonicity failure that can be applied before closure (FCbO) [20,29]. The author's In-Close algorithm introduced a means of testing canonicity *before* a concept is closed [1]. This 'partial-closure' canonicity test was implicit in the original In-Close algorithm but was not fully realised due to the concreteness of its specification. The nature of the canonicity test was obscured by presenting it as a separate 'back-tracking' procedure. In this article, however, the partial-closure test is named, clearly realised and formally presented for the first time.

In FCbO, Outra and Vychodil also introduced a means by which a concept is closed before its descendants are computed, thus allowing the descendants to fully inherit the attributes of the parent. In the spirit of 'best-of-breed' research, the author combined this with his In-Close algorithm to create In-Close2 [2].

This article illustrates, after a brief description of formal concepts, how formal concepts are computed using naïve algorithms that demonstrate the basic recursive process. The main advances in CbO are then described followed by the algorithms in which they appear and combine: CbO, In-Close, FCbO and In-Close2 are presented, along with a final, new, variant called In-Close3. The algorithms are presented using the same notation and style so that the similarities and differences can easily be discerned. Using a simple example, the operation of the CbO algorithm is shown, line-by-line. The key differences between the variants are then compared, using the same example, to highlight where efficiencies occur. Each algorithm is then implemented to compare their performance using a number of real and artificial data sets. The algorithms are implemented using the same optimisations and on the same hardware so that a fair comparison is ensured. Additional experiments are conducted by instrumenting the implemented algorithms to count the number of key operations carried out and thus corroborate the findings of the timed tests. The article concludes by considering what features make the 'Best-of-Breed' CbO algorithm and considering what further steps might be taken.

2. Formal concepts

A description of formal concepts [13] begins with a set of objects X and a set of attributes Y. A binary relation $I \subseteq X \times Y$ is called the *formal context*. If $x \in X$ and $y \in Y$ then xIy says that object x has attribute y. For a set of objects $A \subseteq X$, a derivation operator $^{\uparrow}$ is defined to obtain the set of attributes common to the objects in A as follows:

$$A^{\uparrow} := \{y \in Y | \forall x \in A : xIy\}. \quad (1)$$

Similarly, for a set of attributes $B \subseteq Y$, the $^{\downarrow}$ operator is defined to obtain the set of objects common to the attributes in B as follows:

$$B^{\downarrow} := \{x \in X | \forall y \in B : xIy\}. \quad (2)$$

(A, B) is a formal concept *iff* $A^{\uparrow} = B$ and $B^{\downarrow} = A$. The relations $A \times B$ are then a closed set of pairs in I. In other words, a formal concept is a set of attributes and a set of objects such that all of the objects have all of the attributes and there are no other objects that have all of the attributes. Similarly, there are no other attributes that all the objects have. A is called the *extent* of the formal concept and B is called the *intent* of the formal concept.

A formal context is typically represented as a cross table, with crosses indicating binary relations between objects (rows) and attributes (columns). The following is a simple example of a formal context:

	0	1	2	3	4
a	×			×	×
b		×	×	×	×
c	×		×		
d		×	×		×

Formal concepts in a cross table can be visualised as closed rectangles of crosses, where the rows and columns in the rectangle are not necessarily contiguous. The formal concepts in the example context are:

$C_1 = (\{a,b,c,d\}, \emptyset) \quad C_6 = (\{b\}, \{1,2,3,4\})$
$C_2 = (\{a,c\}, \{0\}) \quad C_7 = (\{b,d\}, \{1,2,4\})$
$C_3 = (\emptyset, \{0,1,2,3,4\}) \quad C_8 = (\{b,c,d\}, \{2\})$
$C_4 = (\{c\}, \{0,2\}) \quad C_9 = (\{a,b\}, \{3,4\})$
$C_5 = (\{a\}, \{0,3,4\}) \quad C_{10} = (\{a,b,d\}, \{4\})$

3. Computation of formal concepts

A formal concept can be computed by applying the $^{\downarrow}$ operator to a set of attributes to obtain its extent, and then applying the $^{\uparrow}$ operator to the extent to obtain the intent. This is the notion of concept closure. For example, taking an arbitrary set of attributes, say $\{1,2\}$, from the context above, $\{1,2\}^{\downarrow} = \{b,d\}$ and $\{b,d\}^{\uparrow} = \{1,2,4\}$. $(\{b,d\},\{1,2,4\})$ is concept C_7 in the list above. If this procedure is applied to every possible combination of attributes, then all the concepts in the context will be computed.

Thus, if there are n attributes in a formal context there are, potentially, 2^n concepts. It is the exponential nature of the problem that provides the computational challenge.

A naïve approach is to systematically generate all the combinations of attributes and perform the closure for each combination. If there are n attributes, $0 \ldots n-1$, then a naïve algorithm to compute all concepts is as follows, where J is a set of attributes, initially empty, and y is the current attribute, initially 0. Note that the attributes are iterated in the natural combinatorial order through a depth-first process of recursion:

```
NaiveComputeConceptsOne(J, y)
  for j ← y upto n − 1 do
      K ← J ∪ {j}
      A ← K↓
      B ← A↑
      ProcessConcept(A, B)
      NaiveComputeConceptsOne(K, j + 1)
```

The notional procedure, `ProcessConcept`, is a means of showing that a concept is recorded, perhaps storing it somewhere for later analysis.

A simple refinement to the naïve algorithm, `NaiveComputeConceptsTwo` below, is possible by realising that it is not necessary to compute the whole of $K^{\downarrow}$ each time. Instead the previous extent A can be intersected with $\{j\}^{\downarrow}$ to form the next extent C. The algorithm is invoked with an initial extent $A = X$ (the set of all objects) and initial attribute 0.

```
NaiveComputeConceptsTwo(A, y)
  for j ← y upto n − 1 do
      C ← A ∩ {j}↓
      B ← C↑
      ProcessConcept(C, B)
      NaiveComputeConceptsTwo(C, j + 1)
```

Applying the closure operator to the extent, C, is implemented by intersecting the extent with each $\{j\}^{\downarrow}$. Whenever the extent is found, the corresponding attribute is added to the intent. In implementation terms, the extent is intersected with each column of the formal context to find every column that contains the extent.

A problem is that the same concept can be computed more than once – in the worst case, exponentially many times. For example, in the context above, the concept C_7 will be computed six times because the extent $\{b,d\}$ can be obtained from the closure of six different combinations of attributes: $\{b,d\} = \{1\}^{\downarrow} = \{1,2\}^{\downarrow} = \{1,2,3\}^{\downarrow} = \{1,4\}^{\downarrow} = \{2,4\}^{\downarrow} = \{4\}^{\downarrow}$. Furthermore, the exponential nature of the computation means that it soon becomes impossible for an implementation of the naïve algorithm to compute the concepts in a practical amount of time.

However, by determining that a concept is a repeat, it is possible to significantly reduce the computation as all further concepts that would be generated by successive intersections must also be repeats. Thus finding a repeated concept is a key halting condition of the algorithm:

```
ComputeConceptsOnce(A, y)
  for j ← y upto n − 1 do
      C ← A ∩ {j}↓
      B ← C↑
      if NewConcept(C, B) then
          ProcessConcept(C, B)
          ComputeConceptsOnce(C, j + 1)
```

Early algorithms concentrated on finding repeats by efficiently searching the previously generated concepts. Lindig's algorithm [26], and others like it, use a search tree to quickly find repeated results. Others use a hash function where the cardinality of results is used to divide them into groups, thus narrowing the search [14]. The problem with these approaches is that even an efficient search becomes expensive when there is a large number of repeated concepts.

4. Advances in CbO

Ganter [12] noticed that the basic algorithm recursively iterated attribute combinations, and thus generated concepts, in the natural combinatorial order. A concept is thus in order if its rank comes after the previous concept. The concepts in the example above have been listed in the natural combinatorial order of their intents: $\emptyset < \{0\} < \{0,1,2,3,4\} < \{0,2\} < \{0,3,4\} < \{1,2,3,4\}$, and so on. It is this canonicity that is fundamental to CbO-type algorithms [24,23] in detecting the repeated computation of a concept. If concepts are computed in this order, then if a concept is generated that has a rank lower or equal to that of its predecessor, it must be a repeat. Kuznetsov specified in Close-by-One (CbO) [24,22] an efficient means of achieving this with a worst-case complexity of $O(|X||Y|^2|L|)$, where L is the set of concepts (or $O(|X|^2|Y||L|)$ if the main iteration of the algorithm is over objects instead of attributes, as in the CbO variants presented here). Ganter's algorithm [12], the CbO variants of Krajca, Outrata and Vychodil [18,20,29] and the author's CbO variants (the In-Close algorithms [1,2]) all have this same worst-case complexity. Comparisons can be found in [23,18,1,20,30,2,17,29]. Each advance or variant of Kuznetsov's original CbO has introduced or combined a number of features to improve efficiency, including:

- A 'full closure' canonicity test to quickly determine repeated concepts [18].
- A more efficient, 'partial closure', canonicity test [1].
- Skipping or inheriting attributes of a parent concept [18,2].
- Inheritance of failed canonicity tests [20,29], so that an attribute that has caused a previous failure can be skipped in subsequent levels.

The features of the CbO variants described in this article are summarised in Table 1.

5. CbO variants

The following sections present these CbO variants together for the first time (including a new variant, In-Close3) and present them using the same notation, abstraction and style, so that the algorithms and their key features can be clearly identified and compared.

For the first time, each of the variants has been implemented, using the same optimisation, compiler and hardware, to provide a level playing field for performance comparisons.

Paper runs of each of the algorithms, using the example context above, are in an on-line appendix [3]. The paper runs are complete, line by line, executions of the algorithms and show that each of the ten concepts, C_1 to C_{10}, is computed once and only once by each algorithm. They provide empirical evidence of how each of the algorithms work and allow their key features to be explored and compared in detail.

5.1. CbO algorithm [18,24,22]

The CbO algorithm of Krajca, Outrata and Vychodil [18] is an efficient realisation of Kuznetsov's CbO [24,22], where the canonicity test for a new concept is carried out by comparing a newly computed intent, D, with its predecessor, B. If they agree in all attributes up to the current attribute, j, then the new concept is canonical. If, however, there is an attribute in D that is not in B and that comes before j then the concept is not canonical (it will have been computed earlier). Thus a new concept is canonical if:

$$B \cap Y_j = D \cap Y_j \tag{3}$$

where Y_j is the set of attributes up to but not including j:

$$Y_j := \{y \in Y | y < j\} \tag{4}$$

The algorithm also passes the intent of a parent concept down to the next level, so that its attributes can be skipped. In effect this is a simple test to avoid repeatedly closing the parent concept.

Table 1
Comparing the features of the CbO variants.

Algorithm	Canonicity test	Skip/inherit attributes	Inherited failure
CbO [18,24,22]	Full closure	✓	×
In-Close [1]	Partial closure	×	×
FCbO [20,29]	Full closure	✓	✓
In-Close2 [2]	Partial closure	✓	×
In-Close3	Partial closure	✓	✓

The algorithm is written below as a single procedure called `Compute-ConceptsFrom`. The procedure is invoked with the initial concept $(A, B) = (X, X^{\uparrow})$ and initial attribute $y = 0$.

CbO

```
ComputeConceptsFrom((A, B), y)
1 ProcessConcept((A, B))
2 for j ← y upto n − 1 do
3     if j ∉ B then
4         C ← A ∩ {j}↓
5         D ← C↑
6         if B ∩ Y_j = D ∩ Y_j then
7             ComputeConceptsFrom((C, D), j + 1)
```

A line-by-line explanation of the algorithm is as follows:

Line 1 – Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts).
Line 2 – Iterate across the context, from starting attribute y up to attribute $n - 1$.
Line 3 – Test if the next attribute is in the current intent, B. If it is, skip it to avoid computing the same concept again.
Line 4 – Otherwise, form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.
Line 5 – Close the extent to form an intent, D. Thus the concept (C, D) is computed ('fully closed') before the canonicity test is carried out to determine if it is a new one:
Line 6 – Perform the canonicity test by checking that attributes in B and D agree up to the current attribute. If they do then the concept (C, D), is a new one so:
Line 7 – Recursively compute concepts from the new one, starting from the next attribute in the context.

Correctness of CbO – The correctness of the original CbO algorithm is given in [24], and proof of the canonicity test has been shown in [18].

CbO call tree – Fig. 1 shows the CbO call tree for the simple example. A rounded box represents the computation of a concept and a square box represents the computation of a repeated concept that then fails the canonicity test. In the upper part of a box, the notation $\langle C_x, y \rangle$ represents a call to CbO: `ComputeConceptsFrom`(C_x, y), where C_x is the concept (to be processed in Line 1) and y is the initial attribute for the cycle in Line 2. The lower part of a box represents the intersections carried out in the computation of C_x. The 'round headed arrows' represent intersections carried out by the closure operator $\uparrow$. In the case of the topmost box in the diagram, these intersections are the ones carried out prior to the initial invocation of CbO to compute the initial concept $C_1 : (X, X^{\uparrow})$. In all the other boxes, the intersections are those carried out in computing $D \leftarrow C^{\uparrow}$ in Line 5. An 'empty square-headed arrow' represents the intersection carried out in Line 4 to form the extent C. The numbers in the lines connecting the boxes represents the current attribute in the cycle in Line 2.

Thus, for example, the box containing $\langle C_2, 1 \rangle$ shows the computation of C_2, with the empty square-headed arrow showing an intersection involving attribute 0 in Line 4: $C \leftarrow A \cap \{0\}^{\downarrow}$ and the round-headed arrows showing the subsequent closure intersections in Line 5, involving attributes $0, 1, 2, 3$ and 4.

There are a total of 15 closures in the tree, with five intersections required for each closure, plus 14 extent intersections, making a total of 89 intersections for CbO. A comparative count of closures and intersections for all the algorithms is given in Table 2.

5.2. In-Close [1] with 'partial closure' canonicity test

This section is an updated version of the algorithm given in [1]. In [1] the canonicity test was described as a backtracking approach but here it is realised for the first time as a 'partial closure' test. The key difference to the first CbO algorithm above is that the canonicity test is applied *before* a concept is fully closed. It is sufficient to examine the context *up to* the current attribute j to determine its canonicity. For this a *partial closure* operator $^{\uparrow_j}$ is defined as a modification of the original closure operator:

$$A^{\uparrow_j} := \{y \in Y_j | \forall x \in A : xIy\}. \tag{5}$$

The 'partial closure' canonicity test determines if the attributes in the intent B, up to j, agree with the attributes in the closure of C up to j:

$$B \cap Y_j = C^{\uparrow_j}. \tag{6}$$

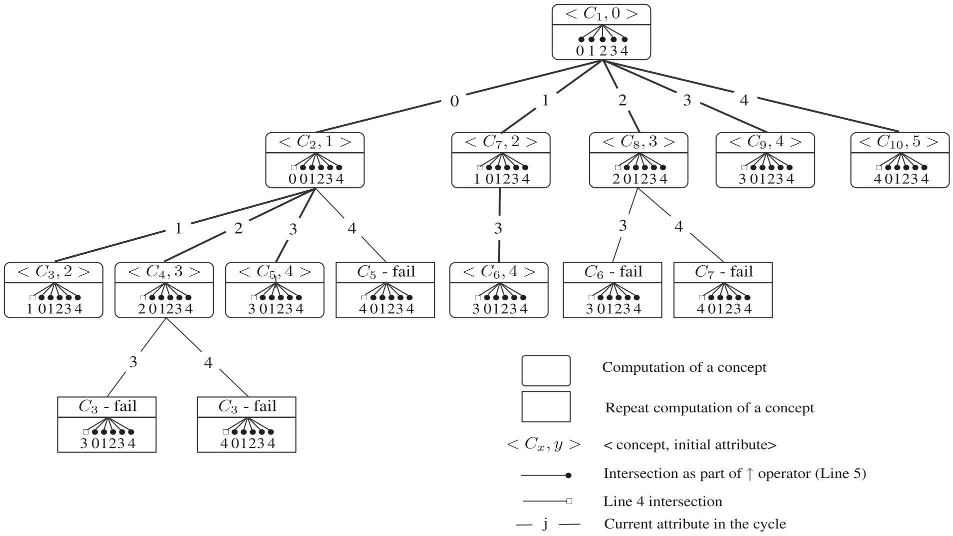

Fig. 1. CbO call tree.

The efficiency is thus that a partial closure, $C^{\uparrow j}$, is clearly less expensive to compute than a complete closure, $C^{\uparrow}$. The cost of the complete closure is a number of intersections equal to the number of attributes, n, whereas the cost of the partial closure is always $< n$. To calculate the average cost of a partial closure, the average value of j during the computation needs to be considered. It is tempting to assume that this average is simply $n/2$ but, because a combinatorial order is being followed, j will always be the largest value in each combination and it is thus necessary to calculate the average largest value in all combinations. This value is dependent on n and, as n becomes large, approaches n. Thus the saving is only some fraction of n, reducing as n increases. However, in practice there will be a further significant saving since it is sufficient to find a non-canonical attribute only once for the test to fail. Thus the partial closure can be halted as soon as such an attribute is found, turning the test into a linear search from attribute 0 up to $j - 1$. This costs $(j - 1)/2$ for a failed test and $j - 1$ for a successful test, approaching $n/2$ and n respectively as n becomes large. Clearly, the failure rate of the canonicity test will depend on the nature of the data. In practice, experiments with data sets [20,29] have shown the failure rate to be typically around 80%, making the average cost of the partial closure test approximately $3n/5$, if n is large. If n is small, then the cost can be taken as $(n - 1)/2$.

Of course, if the canonicity test is passed, new concepts still need to be fully closed and this is carried out incrementally at the next level (hence the algorithm's name: In(*cremental*)-Close(*ure*)), by adding the current attribute j to the intent B whenever the current extent A is found: i.e. whenever $A \cap \{j\}^{\downarrow} = A$. The intent is fully closed when the main cycle is completed. Whenever a new concept is detected, the current (not fully closed intent) is passed to the next level so that some parent attributes can be inherited.

The algorithm, shown in In-Close below, is invoked with the initial pair $(A, B) = (X, \emptyset)$ and initial attribute $y = 0$.

In-Close

ComputeConceptsFrom($(A, B), y$)

1 **for** $j \leftarrow y$ **upto** $n - 1$ **do**
2 $\quad C \leftarrow A \cap \{j\}^{\downarrow}$
3 $\quad$ **if** $A = C$ **then**
4 $\quad\quad B \leftarrow B \cup \{j\}$
5 $\quad$ **else**
6 $\quad\quad$ **if** $B = C^{\uparrow j}$ **then**
7 $\quad\quad\quad D \leftarrow B \cup \{j\}$
8 $\quad\quad\quad$ ComputeConceptsFrom($(C, D), j + 1$)
9 ProcessConcept((A, B))

Line 1 – Iterate across the context, from starting attribute y up to attribute $n - 1$.
Line 2 – Form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.
Line 3 – If the extent formed, C, equals the extent, A, of the concept whose intent is currently being closed, then…
Line 4 – …add the current attribute j to the intent being closed, B.

Table 2
Comparison of closures and intersections for the simple context example.

Algorithm	Full closures (intersections)	Partial closures (intersections)	Extent intersections: $A \cap \{j\}^{\downarrow}$	Total intersections
CbO	15 (75)		14	89
In-Close		14 (37)	22	59
FCbO	14 (70)		13	83
In-Close2		14 (37)	21	58
In-Close3		13 (33)	20	**53**

Line 6 – Otherwise the new partial closure canonicity test is applied. A small simplification to the canonicity test can be made because B is being completed incrementally with j. In other words, at the time of the test, $B = B_j = B \cap Y_j$. Therefore, in the canonicity test, $B \cap Y_j$ can be replaced with B. So if the attributes in B agree with those in the partial closure $C^{\uparrow j}$ the extent must be a new one so…
Line 7 – Create a new intent D that inherits the attributes of B, plus the current attribute j.
Line 8 – Pass the new extent C, the partial intent D and the next location $j+1$ to the next level so that concepts from there can be computed and so that D can be incrementally closed.
Line 9 – Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts). Note that in In-Close this happens at the end of the procedure, once the main cycle has completed the closure of the intent, B.

Correctness of In-Close – CbO and In-Close use the same cycle to form the same extents, C. The only pertinent difference up to this point is the skipping of attributes in CbO when $j \in B$ (Line 3). However, this test is only to avoid forming an extent that will fail the canonicity test anyway. Thus to show that In-Close is correct it is sufficient to show that the partial closure canonicity test is equivalent to the original test, in other words, given the same extent C, show that the tests produce the same result:

$$(B = C^{\uparrow j}) \equiv (B \cap Y_j = D \cap Y_j)$$

As previously stated, in In-Close the intent B is incrementally closed up to the current attribute j, thus $B = B \cap Y_j$, so it is sufficient to show that:

$$C^{\uparrow j} \equiv D \cap Y_j$$

Replacing D with $C^{\uparrow}$, from Line 5 of CbO:

$$C^{\uparrow j} \equiv C^{\uparrow} \cap Y_j$$

Thus, from (1) and (5):

$$\{y \in Y_j | \forall x \in C : xIy\} \equiv \{y \in Y | \forall x \in C : xIy\} \cap Y_j.$$

So on the left side are all the attributes up to j that are related to the extent and on the right are all the attributes that are related to the extent, intersected with all the attributes up to j. Both sides are thus clearly equivalent.

In-Close call tree – Fig. 2 shows the In-Close call tree for the simple example. As in the CbO call tree, a rounded box represents the computation of a concept and a square box represents the computation of a repeated concept that then fails the canonicity test. However, a concept C_x now inherits the intent of the parent concept and has its intent completed as part of the main cycle. For example, the box with $\langle C_2, 1\rangle$ shows the computation of concept C_2. The empty square-headed arrow shows the intersection involving attribute 0 in Line 2: $C \leftarrow A \cap \{0\}^{\downarrow}$. Attribute 0 is subsequently added to C_2's partial intent (inherited from its parent concept C_1) in Line 4. In fact, C_2's inherited partial intent was empty since the intent of C_1 is empty. Adding 0 completes the intent of $C_2 : \{0\}$. There are no intersections involved in the partial closure in Line 6 (no round-headed arrows) because the partial closure is required only up to, but not including, attribute 0.

Similarly, the box with $\langle C_5, 4\rangle$ shows the computation of C_5. Starting the main cycle at attribute 3, the Line 2 intersections for attributes 3 and 4 result in them both being added to the partial intent $\{0\}$, inherited from C_2, thus completing the intent of $C_5 : \{0, 3, 4\}$. The partial closure require to test the canonicity is then up to but not including attribute 3, indicated by the round-headed arrows.

Altogether there are 59 intersections in the In-Close tree, 30 fewer than for CbO. A full comparative count of closures and intersections is given in Table 2.

5.3. FCbO [20,29] with inherited test failure

In [20,29] a new feature of CbO was added by showing that if the original canonicity test in CbO, $B \cap Y_j = D \cap Y_j$, fails for an attribute $j \notin B$, then the test will also fail for each $B' \supseteq B$ where $j \notin B'$, as long as $((D \setminus B) \cap Y_j)$ contains an attribute which is not in $B\prime$. This information can be passed from one level to the next by recording the intent D computed before a failed

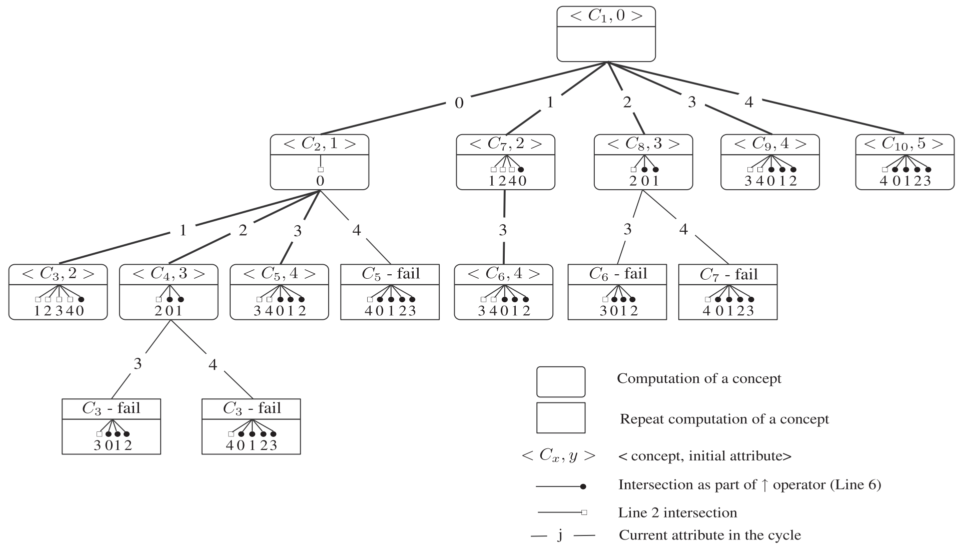

Fig. 2. In-close call tree.

canonicity test for an attribute j. If this record is denoted by N^j, then if $B \cap Y_j = D \cap Y_j$ fails then $N^j = D$. Ns for each attribute are passed to the next level as inherited test failures. For a particular attribute j, if there is an attribute in N^j before j that is not in B before j, then there is no need to carry out the original canonicity test.

To obtain all the failed intents, before passing them to the next level, a combined breadth-first, depth-first approach is used. All attributes are iterated and tested for failure before going to the next level. New concepts, as they are found, are stored in a queue for later processing. Once all attributes have been iterated in the main cycle, the stored concepts are processed by calling the algorithm recursively and passing the corresponding set of failed intents to the next level.

The algorithm is given below. It is invoked with the initial concept $(A, B) = (X, X^{\uparrow})$, initial attribute $y = 0$ and a set of empty Ns, $\{N^y = \emptyset | y \in Y\}$.

FCbO

ComputeConceptsFrom$((A, B), y, \{N^y \mid y \in Y\})$

```
1  ProcessConcept((A, B))
2  for j ← y upto n − 1 do
3      M^j ← N^j
4      if j ∉ B and N^j ∩ Y_j ⊆ B ∩ Y_j then
5          C ← A ∩ {j}↓
6          D ← C↑
7          if B ∩ Y_j = D ∩ Y_j then
8              PutInQueue ((C, D), j)
9          else
10             M^j ← D
11 while GetFromQueue((C, D), j) do
12     ComputeConceptsFrom((C, D), j + 1, {M^y | y ∈ Y})
```

Line 1 – Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts).
Line 2 – Iterate across the context (breadth-first), from starting attribute y up to attribute $n - 1$.
Line 3 – The record of failure for attribute j, M_j, to be inherited by the next level, is set to the current record of failure N_j.
Line 4 – Skip attributes in B and those that have an inherited record of failure.
Line 5 – Otherwise, form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.
Line 6 – Close the extent to form an intent, D.
Line 7 – Perform the original canonicity test.
Line 8 – If the concept is a new one, store it in a queue along with the attribute it was computed at.

Line 10 – Otherwise set the record of failure for attribute j, M_j, to the intent that failed the canonicity test.
Line 11 – Now do the depth-first part by getting each stored concept from the queue...
Line 12 – and passing it to the next level, along with the stored starting attribute for the next level and the records of failure from this level.

Correctness of FCbO – The correctness of FCbO and proof of the inheritance of canonicity failure is given in [20,29].

FCbO call tree – Fig. 3 shows the call tree for FCbO. It is the same as the tree for CbO except that a repeat computation of concept C_3 is avoided because of an inherited canonicity failure. The inherited failure is from the first C_3 canonicity test failure which occurred after the computation of C_4 (see the listing of the paper run of FCbO for a line by line demonstration of the inherited failure [3]). This gives a saving of six intersections compared to CbO. A comparative count of closures and intersections is given in Table 2.

5.4. In-Close2 [2] with 'partial closure' test and fully inherited intents

The algorithm presented here is an updated version of that presented in [2]. In-Close2 has been 'bred' from In-Close and FCbO to combine the efficiencies of the partial closure canonicity test with full inheritance of the parent intent. It achieves this by adapting and adopting the breadth-first and depth-first approach of FCbO [20,29]. The main cycle is completed before passing to the next level, so that all the attributes of a parent intent can be passed down to the next level rather than just some of them. Like In-Close, child intents only have to be 'finished off' by adding attributes when $A = C$, but now additional attributes after j are also inherited and can be skipped. During the main cycle, whilst the current intent is being closed, new extents that pass the canonicity test are stored in a queue, similar to the queue in FCbO, to be processed after the main cycle has completed.

The In-Close2 algorithm, given below, is invoked the same way as In-Close, with an initial $(A, B) = (X, \emptyset)$ and initial attribute $y = 0$.

In-Close2

ComputeConceptsFrom$((A, B), y)$

```
1  for j ← y upto n − 1 do
2      if j ∉ B then
3          C ← A ∩ {j}↓
4          if A = C then
5              B ← B ∪ {j}
6          else
7              if B ∩ Y_j = C↑j then
8                  PutInQueue(C, j)
9  ProcessConcept((A, B))
10 while GetFromQueue(C, j) do
11     D ← B ∪ {j}
12     ComputeConceptsFrom((C, D), j + 1)
```

Line 1 – Iterate across the context, from starting attribute y up to attribute $n - 1$.
Line 2 – Skip attributes already in B. Because intents now inherit all of their parent's attributes, these can be skipped.
Line 3 – Form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.
Line 4 – If the extent formed, C, equals the extent, A, of the concept whose intent is currently being closed, then...
Line 5 – ...add the current attribute j to the intent being closed, B.
Line 7 – Otherwise, test the canonicity. The *unsimplified* version of the partial closure test (6) is now used because intents now contain inherited attributes after j.
Line 8 – If the canonicity test is passed, place the new extent, C, and the location where it was found, j, in a queue for later processing.
Line 9 – Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts).
Lines 10 – The queue is processed by obtaining each new extent and associated location from the queue.
Line 11 – Each new partial intent, D, inherits all the attributes from its completed parent intent, B, along with the attribute, j, where its extent was found.
Line 12 – Call `ComputeConceptsFrom` to compute child concepts from $j + 1$ and to complete the intent D.

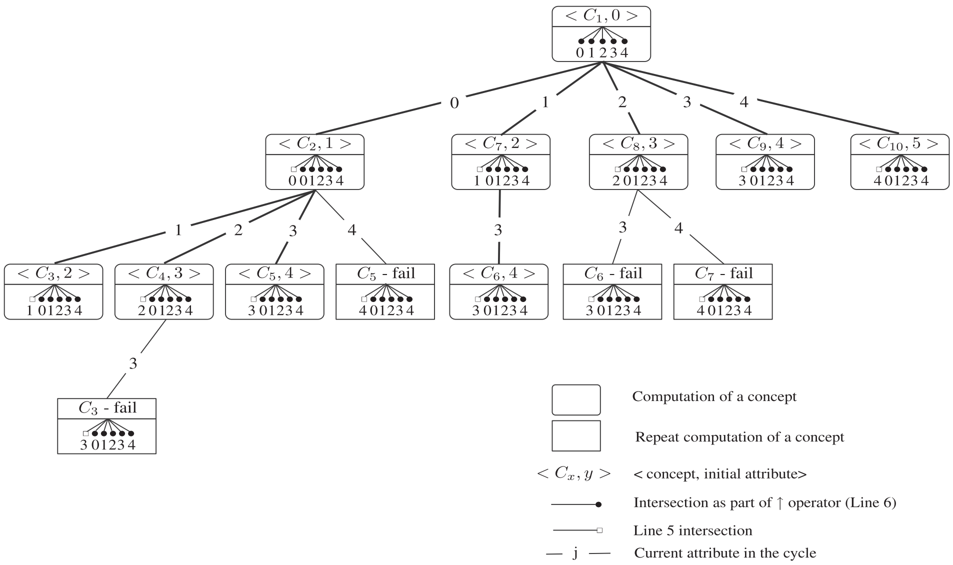

Fig. 3. FCbO call tree.

Correctness of In-Close2 – The process of incremental closure and canonicity testing (lines 3–7) is unchanged from CbO version 2, apart from the new initial test $j \notin B$. To show that CbO version 3 is correct it is thus sufficient to show that the new test does not exclude any required attributes from this process. The possible outcomes after forming C are either adding the attribute to B in $B \leftarrow B \cup \{j\}$ or carrying out the canonicity test $B \cap Y_j = C^{\uparrow j}$. It is thus necessary to show that any attributes excluded by the test (by failing $j \notin B$) do not affect these outcomes.

For the first outcome, if $j \in B$ then $B \leftarrow B \cup \{j\} = B$. In other words, j is already in B, so adding it would have no effect.

For the second outcome, if $j \in B$ then $B \cap Y_j$ would be unchanged since Y_j by definition does not include j.

Thus it is not possible for the new test to exclude any required attributes from the computation.

In-Close2 call tree – Fig. 4 shows the call tree for In-Close2. It is the same as the tree for In-Close except that now concept C_6 inherits an additional attribute, 4, from concept C_7. This is because the main cycle is now completed before passing to the next level, so that attribute 4 is added to the parent intent before the next level is invoked. Thus one intersection is saved compared to In-Close. A comparative count of closures and intersections is given in Table 2.

5.5. In-Close3 with 'partial closure' test, inherited intents and inherited failure test

This is a new variant of CbO that 'breeds' In-Close2 and FCbO to include the partial closure canonicity test and the full inheritance of intents of In-Close2, along with the inherited failed canonicity tests of FCbO.

The In-Close3 algorithm is given below.

In-Close3

ComputeConceptsFrom$((A,B), y, \{N^y \mid y \in Y\})$

for $j \leftarrow y$ **upto** $n-1$ **do**
  $M^j \leftarrow N^j$
  if $j \notin B$ **and** $N^j \cap Y_j \subseteq B \cap Y_j$ **then**
    $C \leftarrow A \cap \{j\}^{\downarrow}$
    if $A = C$ **then**
      $B \leftarrow B \cup \{j\}$
    else
      if $B \cap Y_j = C^{\uparrow j}$ **then**
        PutInQueue(C, j)
      else
        $M^j \leftarrow C^{\uparrow j}$
ProcessConcept$((A,B))$
while GetFromQueue(C, j) **do**
  $D \leftarrow B \cup \{j\}$
  ComputeConceptsFrom$((C,D), j+1, \{M^y \mid y \in Y\})$

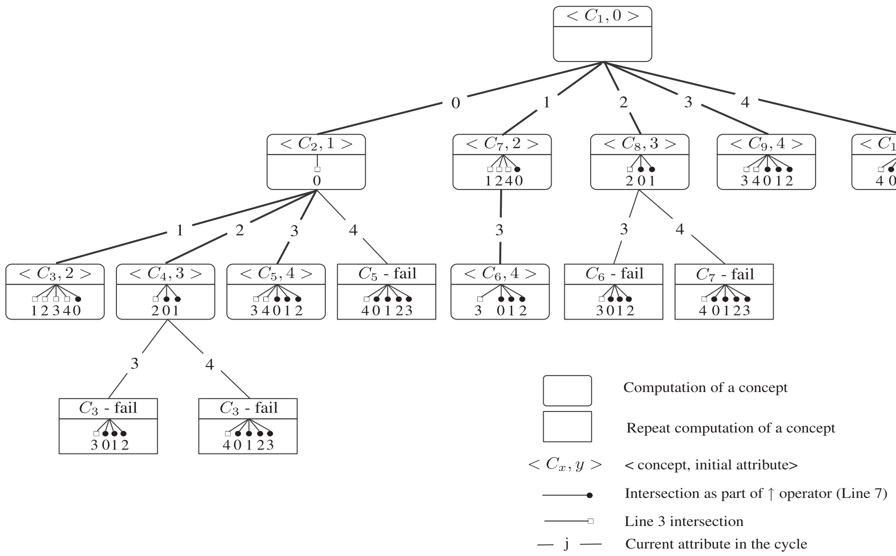

Fig. 4. In-Close2 call tree.

Line 1 – Iterate across the context (breadth-first), from starting attribute y up to attribute $n-1$.
Line 2 – The record of failure for attribute j, M_j, to be inherited by the next level, is set to the current record of failure N_j.
Line 3 – Skip attributes already in B and those that have an inherited record of failure.
Line 4 – Otherwise, form an extent, C, by intersecting the current extent, A, with the next column of objects in the context.
Line 5 – If the extent formed, C, equals the extent, A, of the concept whose intent is currently being closed, then...
Line 6 – ...add the current attribute j to the intent being closed, B.
Line 8 – Otherwise, perform the partial closure canonicity test and, if the extent is new...
Line 9 – ...store it in a queue along with the attribute it was computed at.
Line 11 – Otherwise set the record of failure for attribute j, M_j, to the intent that failed the canonicity test.
Line 11 – Now do the depth-first part by getting each stored concept from the queue...
Line 11 – and passing it to the next level, along with the stored starting attribute for the next level and the records of failure from this level.
Line 11 – if the partial closure canonicity test fails for an attribute j, the attributes in the partial closure are recorded by M^j.
Line 12 – Pass concept (A, B) to notional procedure `ProcessConcept` to process it in some way (for example, storing it in a set of concepts).
Lines 13 – The queue is processed by obtaining each new extent and associated location from the queue.
Line 14 – Each new partial intent, D, inherits all the attributes from its completed parent intent, B, along with the attribute, j, where its extent was found.
Line 15 – Call `ComputeConceptsFrom` to compute child concepts from $j+1$ and to complete the intent, D, also passing down the records of failure from this level.

Correctness of In-Close3 – The correctness of the canonicity test has been shown above as has the completeness of the closure. However, the record of failure in In-Close3 is $M^j \leftarrow C^{\uparrow j}$ whereas in FCbO it is $M^j \leftarrow D = C^{\uparrow}$. Thus it must be shown that the records are equivalent in the inherited canonicity failure test $N^j \cap Y_j \subseteq B \cap Y_j$.

Since $C^{\uparrow} \cap Y_j = C^{\uparrow j}$, then for In-Close3, $N^j \cap Y_j = C^{\uparrow} \cap Y_j \cap Y_j = C^{\uparrow} \cap Y_j$, which is the test for FCbO. Thus the two tests are equivalent.

In-Close3 call tree – Fig. 5 shows the call tree for In-Close3. It is the same as the tree for In-Close2 except that now the second repeat of C_3 is avoided, as it was with FCbO. The first canonicity failure of C_3 is inherited during the computation of C_4. Thus, a saving of five intersections is made compared to In-Close2. A comparative count of closures and intersections is given in Table 2.

6. Performance evaluation

6.1. Paper runs using the simple example

Table 2 compares the performance of each algorithm using the simple context example from Section 2. The table counts and compares the number of full and partial closures and the number of extent intersections, $A \cap \{j\}^{\downarrow}$. Because closure itself involves repeated intersections of an extent with columns of the context, it is convenient to use the intersection as a measure

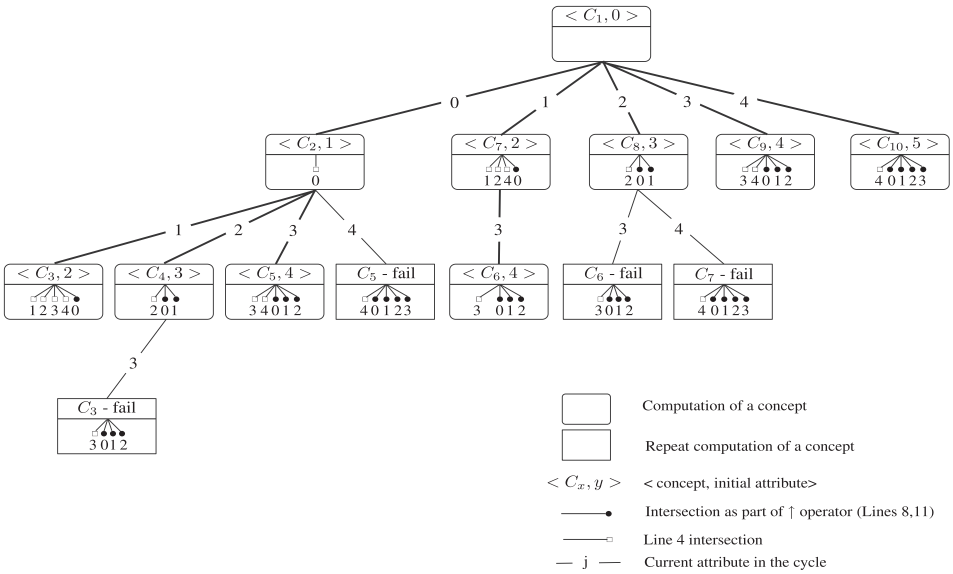

Fig. 5. In-Close3 call tree.

of performance. For a full closure, $C^{\uparrow}$, there are n intersections (in the example, $n = 5$) and for a partial closure, $C^{\uparrow j}$, there are $j - 1$ intersections. To perform the evaluation, a paper run of each algorithm was carried out, line by line. These are too lengthy to be presented here but are available as an on-line appendix [3].

In-Close requires one less closure than CbO. This is because CbO (and FCbO) require an initial closure, $X^{\uparrow}$, that In-Close, and subsequent In-Close variants, do not. The partial closure test saves 38 intersections for In-Close, compared to the full closures in CbO. In-Close, however, requires an additional 8 extent intersections to complete the closure of intents. Thus overall, In-Close required 30 fewer intersections than CbO. FCbO saved six intersections compared to CbO through inherited failure avoiding a repeated concept computation. In-Close2 saved an intersection compared to In-Close by inheriting an attribute in a parent intent. In-Close3 saved a further five intersections through inherited canonicity failure.

6.2. Implementations

Each algorithm was been implemented in C++ and a series of tests were carried out using contexts created from real data sets, artificial data sets and randomised data sets. The experiments were carried out using a standard Windows PC with an Intel E4600 2.39 GHz processor and 3 GB of RAM. The times for the programs include data pre-processing, such as sorting, but exclude administrative aspects, such as data file input.

To create a level playing field for testing, the algorithms were implemented with the same two optimisations. Although it would have been possible to create un-optimised implementations, times for real data sets would be prohibitively slow and comparisons with times presented elsewhere for similar experiments would be unhelpful. The two optimisations used were:

- Sorting context columns in order of density
- Implementing the context as a bit-array

The practice of column-sorting to improve concept computation is well known [9]. By doing so, there are fewer canonicity test failures. This is because there is less chance of finding A before attribute j since the context is less dense before attribute j.

The use of bit-arrays is well known in computation, allowing a SIMD (Single Operation – Multiple Data) approach, where, for example, 32 context rows can be intersected simultaneously using standard 32-bit operations [18].

For the In-Close variants, the full savings of the partial closure canonicity test were realised in the implementations: for the test to fail it is only necessary to find one attribute that is not canonical. Thus the partial closure test can be ended as soon as such an attribute is found, even if that is before $j - 1$.

The inherited canonicity failure required the creation of a two-dimensional array to implement the set of failure records, $\{N^y | y \in Y\}$, one failed intent for each attribute. This was required to be passed to each level of recursion. Although the use of pointers reduces the need for copying arrays in memory, the manipulation and accessing of the arrays gives rise to some additional complexity in the computation.

Extents were stored as 'end-to-end' lists of integers in a one-dimensional array to save memory. Storing extents in a two-dimensional array may, in theory, give rise to some time savings, being a simpler data structure. However, in practice, with typical data sets, the large number of objects and concepts makes the required size of the array prohibitive.

Intents were stored in a two-dimensional bit-array, each being stored as n bits. The use of bits and the fact that the number of attributes is typically much smaller than objects, makes the memory requirements tractable. For testing $j \notin B$, the Boolean nature of the bit-array version of intents is an efficient structure, with the test becoming: `if not(B[j])`.

For carrying out the extent intersection, $C \leftarrow A \cap \{j\}^{\downarrow}$, it was a simple case of testing the bit-position in column j of the context for each integer in A. For the In-Close variants, the size of C (produced as a by-product of the extent intersection) was used to test equality between C and A. In effect the test becomes: $|A| = |C|$, incurring no additional overhead.

For the closure/partial closure of C, a bit-wise Boolean *and* operator was used to 'parse' 32 columns of the context at a time, using the integers in A to identify the rows to test. For the In-Close variants, of course, as soon as C was found in a column before j and not in B, the closure was halted.

6.2.1. Real data set experiments

Three of the data sets are from the UCI Machine Learning Repository [11]: *Mushroom*, *Adult* and *Internet Ads*. A fourth data set, *Student*, is a set of results from a student questionnaire used to obtain course feedback at Sheffield Hallam University, UK in 2010. The data sets provide a range of size and density to test the implementations under a variety of real conditions. Formal contexts were created from these data sets using well known FCA scaling techniques where many-valued data is converted into Boolean form [33,13,6]. The results are given in Table 3. The results bear out the analysis and comparison of the algorithms given earlier, showing that the partial closure canonicity test is a significant factor in the efficiency of the implementations of the algorithms, but also indicate that attribute inheritance/skipping and inherited canonicity test failure contribute to time savings.

Table 3
Real data set results (timings in seconds).

	Mushroom	Adult	Internet Ads	Student
$\|G\| \times \|M\|$	$8,124 \times 125$	$32,561 \times 124$	$3,279 \times 1,565$	587×145
Density	17.36%	11.29%	0.97%	24.50%
#Concepts	226,921	1,388,469	16,570	22,760,243
CbO	0.66	3.06	0.56	32.68
In-Close	0.40	1.65	0.12	11.42
FCbO	0.35	2.06	0.21	17.20
In-Close2	0.35	**1.48**	**0.08**	9.89
In-Close3	**0.29**	1.62	0.10	**9.38**

Table 4
Comparison of closures and intersections for the real data sets.

	Full/partial closures (intersections)	Extent intersections	Total intersections
Adult			
CbO	300,674 (37,283,576)	300,673	37,584,249
InClose	300,673 (26,994,788)	356,967	27,351,755
FCbO	153,745 (19,064,380)	153,745	19,218,125
InClose-II	300,673 (26,994,788)	331,267	27,326,055
InClose-III	153,745 (13,834,074)	184,339	**14,018,413**
Ad			
CbO	1,867,690 (2,922,934,850)	1,867,689	2,924,802,539
InClose	1,867,689 (2,190,704,598)	1,924,704	2,192,629,302
FCbO	333,626 (522,124,690)	333,626	522,458,316
InClose-II	1,867,689 (2,190,704,598)	1,897,257	2,192,601,855
InClose-III	333,626 (386,500,353)	363,194	**386,863,547**
Mushroom			
CbO	1,164,839 (145,604,875)	1,164,838	146,769,713
InClose	1,164,838 (129,543,924)	2,727,529	132,271,453
FCbO	282,165 (35,270,625)	282,165	35,552,790
InClose-II	1,164,838 (129,543,924)	1,191,891	130,735,815
InClose-III	282,164 (31,386,369)	309,217	**31,695,586**
Student			
CbO	55,038,423 (7,980,571,335)	55,038,422	8,035,609,757
InClose	55,038,422 (7,632,184,009)	78,630,232	7,710,814,241
FCbO	33,003,581 (4,785,519,245)	33,003,581	4,818,522,826
InClose-II	55,038,422 (7,632,184,009)	62,045,062	7,694,229,071
InClose-III	33,003,581 (4,605,561,369)	40,010,221	**4,645,571,590**

The programs were then instrumented to count the number of closures and intersections carried out (see Table 4) to compare the number of these operations with the timings. The ranking of algorithms is similar except for FCbO which is now second best after In-Close3, in terms of fewest intersections. It is clear that, in practice, there is higher number of inherited canonicity failures than that shown in the simple example. Thus, the most striking difference is in the large savings of closures made by the inherited failure tests of FCbO and In-Close3, significantly reducing the number of intersections required. However, the savings in time are not proportionate to the savings in intersections. It would seem that the overheads of the additional computational complexity of inherited failure were having a significant impact on the performance of these implementations, even though efficient coding was used. Similar results have been obtained elsewhere [30,17,29].

6.2.2. Artificial data set experiments

The following artificial data sets were used:

- M7X10G120K – a data set based on simulating many-valued attributes. The scaling of many-valued attributes is simulated by creating 'blocks' in the context containing disjoint columns. There are 7 blocks, each containing 10 disjoint columns. The data set was created by writing a simple C++ program with a random number generator.
- M10X30G120K – a similar data set, this time with a context containing 10 blocks, each with 30 disjoint columns.
- T10I4D100K – an artificial data set from the FIMI data set repository [15].

The results of the artificial data set experiments are given in Table 5. The results tend to replicate the findings of the real data set experiments and corroborate the previous comparison of the algorithms.

Table 5
Artificial data set results (timings in seconds). Figures in bold are the fastest times.

	M7X10G120K	M10X30G120K	T10I4D100K
$\lvert G\rvert \times \lvert M\rvert$	$120,000 \times 70$	$120,000 \times 300$	$100,000 \times 1,000$
Density	10.00%	3.33%	1.01%
#Concepts	1,166,343	4,570,498	2,347,376
CbO	2.51	31.26	49.45
In-Close	1.26	18.95	16.02
FCbO	1.67	22.33	29.41
In-Close2	**1.21**	12.10	11.23
In-Close3	1.39	**10.42**	**11.04**

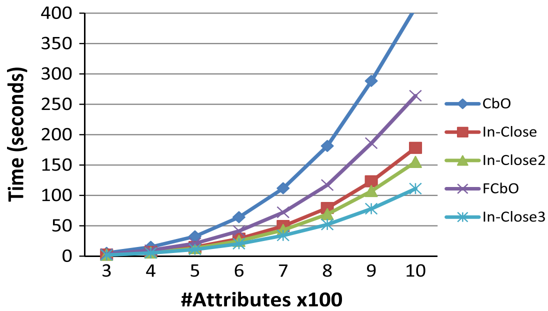

Fig. 6. Comparison of performance with varying number of attributes. 5% density, 5000 objects. #Concepts range from approx. 1,000,000–22,000,000.

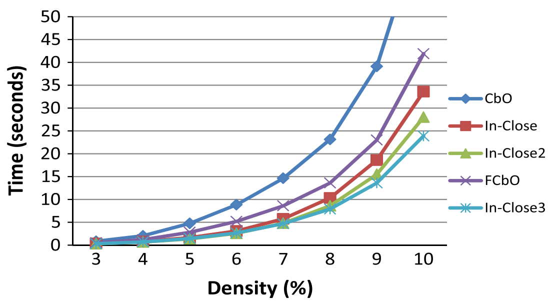

Fig. 7. Comparison of performance with varying context density. 200 attributes, 10,000 objects. #Concepts range from approx. 4,000,000–22,000,000.

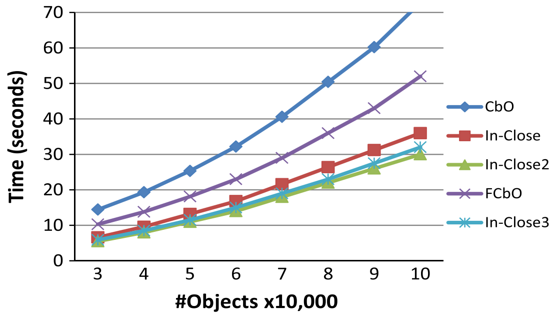

Fig. 8. Comparison of performance with varying number of objects. 5% density, 200 attributes. #Concepts range from approx. 200,000–19,000,000.

6.2.3. Random data set experiments

Three series of random data experiments were carried out to compare the performance of In-Close2 and FCbO, testing the affect of changes in the number of attributes, context density, and number of objects. Note that randomly assigning crosses in a table gives rise to a much larger number of concepts compared to real data sets of similar size and density.

- *Attributes series* – with 5% density and 5000 objects, the number of attributes was varied between 300 and 1000 (Fig. 6). The number of concepts varied from approximately 1 million to 22 million.
- *Density series* – with 200 attributes and 10,000 objects, the density of 1s in the context was varied between 3% and 10% (Fig. 7). The number of concepts varied from approximately 200 thousand to 19 million.
- *Objects series* – with 5% density and 200 attributes, the number of objects was varied between 30,000 and 100,000 (Fig. 8). The number of concepts varied from approximately 4 million to 22 million.

The results of randomised data set series also tend to bear out the analysis and comparison of the algorithms given earlier, showing the partial closure canonicity test to be the most significant factor in the efficiency of the algorithms, but again also indicating that attribute inheritance/skipping and inherited canonicity test failure add to time savings. This is particularly clear in the comparison of In-Close and In-Close2 with respect to FCbO. In the density series, where one might expect a high degree of inheritance, FCbO does better than In-Close, but not In-Close2 which 'bred' the inheritance from FCbO. In the case of the attribute and density series, In-Close3 performed best, but in the object series In-Close2 was slightly quicker. Both versions incorporate the partial closure test and skip inherited attributes but In-Close3 has the inherited canonicity test failure, 'bred' from FCbO, whereas In-Close2 does not. Again it seems that the complexity overhead of maintaining the records of failure balances the savings in intersections, particularly where the number of objects increases.

7. Conclusions and further work

The analysis and comparison of the CbO algorithms presented here, and the results of the experiments carried out, show that the partial closure canonicity test is the most significant factor in the efficiency of CbO algorithms. However, in terms of the number of closures required, the inherited test failure approach of FCbO is the most significant. Breeding In-Close with FCbO, to create In-Close2, incorporating attribute inheritance with partial closure, produced a significant improvement in efficiency, resulting from a significant increase in attributes skipped. Further breeding with FCbO to create In-Close3, incorporating inherited canonicity failure, produced an algorithm that usually, but not always, performed slightly better than In-Close2. It is debatable, therefore, which of In-Close2 and In-Close3 is the 'best of breed'. Although inherited failure made a significant difference to CbO, it had much less impact on In-Close2. Although there were far fewer intersections when inherited failure was incorporated, this appeared to be masked by the extra time taken with the inherited failure mechanisms. In case this was due to an inefficient implementation by the author, the source code provided by the original authors of FCbO was also timed, with the same results. Whilst it is hard to instrument the inherited failure mechanisms directly, as much is to do with memory use and the associated code is rather dispersed, it was possible, by removing all parts involved with inherited failure and the associated structures, to compare FCbO and In-CloseIII, with and without the feature. This experiment confirmed the timings: although there was a large increase in closures with the parts removed, there was only comparatively small slowdown in the overall computation. Thus it is questionable that the added complexity of inherited failure feature provides enough benefit to the performance of these algorithms to make it worthwhile. However, the large reduction in closures resulting from inherited failure does make further work in this area tempting. Can an algorithm be found that incorporates this feature of inherited canonicity failure but without the current complexity?

Future work is also required in increasing performance through parallel processing. Work has been carried out to develop parallel versions of CbO (PCbO) and FCbO (PFCbO) [18,20] but not as yet for the In-Close variants.

Besides work on further improving the efficiency of the computation of formal concepts, what is also required is the development of more applications that harness this power, such as in the areas of classification and machine learning. These improvements in efficiency make FCA available to a larger scale of data, but new tools and techniques are required to provide useful and meaningful results from the large numbers of concepts typically produced by such computations.

Implementations of these algorithms (In-Close variants and FCbO) are available open-source at *SourceForge* [4,19].

Acknowledgments

This work is part of the CUBIST project ("Combining and Uniting Business Intelligence with Semantic Technologies"), funded by the European Commission's 7th Framework Programme of ICT, under topic 4.3: Intelligent Information Management. Project ID: CUBIST-257403.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.ins.2014.10.011.

References

[1] S. Andrews, In-close, a fast algorithm for computing formal concepts, in: S. Rudolph, F. Dau, S.O. Kuznetsov, (Eds.), ICCS 2009, CEUR WS, vol. 483, 2009. <http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-483/>.
[2] S. Andrews, In-close2, a high performance formal concept miner, in: S. Andrews, S. Polovina, R. Hill, B. Akhgar (Eds.), Conceptual Structures for Discovering Knowledge – Proceedings of the 19th International Conference on Conceptual Structures (ICCS), Springer, 2011, pp. 50–62.
[3] S. Andrews, Appendix to a Best of Breed Approach to Designing a Fast Algorithm for Computing Fixpoints of Galois Connections, 2013. <https://dl.dropboxusercontent.com/u/3318140/bob_appendix.pdf>.
[4] S. Andrews, In-Close Program, 2013. <http://sourceforge.net/projects/inclose/>.
[5] S. Andrews, C. Orphanides, Analysis of large data sets using formal concept lattices, in: [21], 2010, pp. 104–115.
[6] S. Andrews, C. Orphanides, Fcabedrock, a formal context creator, in: M. Croitoru, S. Ferre, D. Lukose (Eds.), ICCS 2010, LNCS, vol. 6208/2010, Springer, 2010.
[7] D. Borchman, A generalized next-closure algorithm – enumerating semilattice elements from a generating set, in: L. Szathmary, U. Priss, (Eds.), Proceedings of Concept Lattices and thie Applications (CLA) 2012, Universidad de Malaga, 2012, pp. 9–20.
[8] J.P. Bordat, Calcul pratique du treillis de Galois dune correspondance, Math. Sci. Hum. 96 (1986) 31–47.
[9] C. Carpineto, G. Romano, Concept Data Analysis: Theory and Applications, J. Wiley, 2004.
[10] M. Chein, Algorithme de recherche des sous-matrices premires dune matrice, Bull. Math. Soc. Sci. Math. R.S. Roumanie 13 (1969) 21–25.
[11] A. Frank, A. Asuncion, UCI Machine Learning Repository, 2010. <http://archive.ics.uci.edu/ml>.
[12] B. Ganter, Two Basic Algorithms in Concept Analysis, FB4-Preprint 831, TH Darmstadt, 1984.
[13] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer-Verlag, 1998.
[14] R. Godin, R. Missaoui, H. Alaoui, Incremental concept formation algorithms based on Galois lattices, Comput. Intell. 11 (2) (1995) 246–267.
[15] B. Goethals, Frequent Itemset Implementations (fimi) Repository, 2010. <http://fimi.cs.helsinki.fi/>.
[16] M. Kaytoue, S.O. Kuznetsov, A. Napoli, S. Duplessis, Mining gene expression data with pattern structures in formal concept analysis, Inf. Sci. 181 (10) (2011) 1989–2001.
[17] M. Kirchberg, E. Leonardi, Y.S. Tan, S. Link, R.K.L. Ko, B.S. Lee, Formal concept discovery in semantic web data, LNAI, vol. 7278, Springer-Verlag, Berlin Heidelberg, 2012.
[18] P. Krajca, J. Outrata, V. Vychodil, Parallel recursive algorithm for FCA, in: R. Belohavlek, S. Kuznetsov, (Eds.), Proceedings of Concept Lattices and their Applications, 2008.
[19] P. Krajca, J. Outrata, V. Vychodil, FCbO Program, 2012. <http://fcalgs.sourceforge.net/>.
[20] P. Krajca, V. Vychodil, J. Outrata, Advances in Algorithms Based on CbO, In: [21], 2010, pp. 325–337.
[21] M. Kryszkiewicz, S. Obiedkov, (Eds.), Proceeding of 7th International Conference on Concept Lattices and Their Applications, CLA 2010, Seville, University of Sevilla, 2010.
[22] S. Kuznetsov, Learning of simple conceptual graphs from positive and negative examples, in: J. Zytkow, J. Rauch (Eds.), PKDD'99, Lecture Notes in Computer Science, vol. 1704, Springer, 1999, pp. 384–391.
[23] S. Kuznetsov, S. Obiedkov, Comparing performance of algorithms for generating concept lattices, J. Exp. Theor. Artif. Intell. 14 (2002) 189–216.
[24] S.O. Kuznetsov, Mathematical aspects of concept analysis, Math. Sci. 80 (2) (1996) 1654–1698.
[25] S.O. Kuznetsov, On computing the size of a lattice and related decision problems, Order 18 (4) (2001) 313–321.
[26] C. Lindig, Fast concept analysis, in: Working with Conceptual Structures: Contributions to ICCS 2000, Shaker Verlag, Aachen, 2000, pp. 152–161.
[27] E. Norris, Maximal rectangular relations, in: Proceedings of the 1st Conference on Fundamentals of Computing Theory, Lecture Notes in Computer Science, vol. 56, Springer, 1977, pp. 476–481.
[28] L. Nourine, O. Raynaud, A fast algorithm for building lattices, Inf. Process. Lett. 71 (1999) 199–204.
[29] J. Outrata, V. Vychodil, Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data, Inf. Sci. 185 (1) (2012) 114–127.
[30] F. Strok, A. Neznanov, Comparing and analyzing the computational complexity of fca algorithms, in: Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, 2010, pp. 417–420.
[31] T. Tanabata, K. Sawase, H. Nobuhara, B. Bede, Interactive data mining for image databases based on FCA, J. Adv. Comput. Intell. Intell. Inf. 14 (3) (2010) 303–308.
[32] D. Van der Merwe, S. Obiedkov, D. Kourie, Addintent: a new incremental algorithm for constructing concept lattices, in: P. Eklund (Ed.), ICFCA 2004, Lecture Notes in Computer Science, vol. 2961, Springer, 2004, pp. 372–385.
[33] K.E. Wolff, A first course in formal concept analysis: how to understand line diagrams, Adv. Stat. Software 4 (1993) 429–438.

Simon Andrews is a Reader in Computer Science at Sheffield Hallam University, UK. He is the Technical Lead of CENTRIC (Centre of excellence in Terrorism, Resilience, Intelligence and Organised Crime Research) at Sheffield Hallam University, where his main roles are to provide technical visions for new project ideas, and manage the technical work in current projects, such as the European ePOOLICE project (environmental scanning for organised crime detection) and the European ATHENA project (crisis management through the use of social media). He was also a Principle Investigator for the European CUBIST project (Combining and Uniting Business Intelligence with Semantic Technologies).

He is an international expert on Formal Concept Analysis (FCA) and is co-Editor in Chief of the International Journal of Conceptual Structures and Smart Applications (IJCSSA). He was Program Chair of the 19th International Conference on Conceptual Structures and has presented his work on fast concept-mining algorithms world-wide, including at events in Moscow and Japan.