A Partial-Closure Canonicity Test to Increase the Efficiency of CbO-Type Algorithms

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

Published version

ANDREWS, Simon (2014). A Partial-Closure Canonicity Test to Increase the Efficiency of CbO-Type Algorithms. In: HERNANDEZ, Nathalie, JÄSCHKE, Robert and CROITORU, Madalina, (eds.) Graph-Based Representation and Reasoning. Lecture Notes in Computer Science (8577). Springer, 37-50.

A partial-closure canonicity test to increase the efficiency of CbO-type algorithms

Simon Andrews

Conceptual Structures Research Group
Communication and Computing Research Centre
Faculty of Arts, Computing, Engineering and Sciences
Sheffield Hallam University, Sheffield, UK
s.andrews@shu.ac.uk

Abstract. Computing formal concepts is a fundamental part of Formal Concept Analysis and the design of increasingly efficient algorithms to carry out this task is a continuing strand of FCA research. Most approaches suffer from the repeated computation of the same formal concepts and, initially, algorithms concentrated on efficient searches through already computed results to detect these repeats, until the so-called canonicity test was introduced. The canonicity test meant that it was sufficient to examine the attributes of a computed concept to determine its newness: searching through previously computed concepts was no longer necessary. The employment of this test in Close-by-One type algorithms has proved to be highly effective. The typical CbO approach is to compute (close) a concept and then test its canonicity. This paper describes a more efficient approach, whereby a concept need only be partially closed in order to carry out the test. Only if it passes the test does its closure need to be completed. This paper presents this partial-closure canonicity test in the In-Close algorithm and compares this to a traditional CbO algorithm to demonstrate the increase in efficiency.
Keywords: Formal Concept Analysis; FCA; FCA algorithms; Computing formal concepts; Canonicity test; Partial-closure canonicity test; Close-by-One; In-Close; CbO

1 Introduction

The emergence of Formal Concept Analysis (FCA) as a data analysis technique [5, 14, 29] has increased the need for algorithms that compute formal concepts quickly. A problem in computing these formal concepts 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 [20]. Furthermore, computation of formal concepts typically involves repeated computation of the same concept [8], which is unwanted as we are normally interested only in obtaining the unique concepts. Older algorithms relied on ever more efficient search techniques to find repeated concepts, but these algorithms were superseded by the discovery of the so-called

'canonicty test' [11], whereby the attributes concept could be examined to determine its uniqueness in the computation. This test has given rise to a number of algorithms based on the Close-by-One (CbO) algorithm [19] and the focus of research has been to develop improvements of the original CbO algorithm. This paper presents one such improvement in the form of a more efficient canonicity test: the partial-closure canonicity test.

The next two sections of the paper briefly describe formal concepts and the main issues involved in their efficient computation, showing how testing a computed concept's canonicity is an efficient means of detecting the computation of repeated results. Section 4 shows a CbO algorithm incorporating a canonicity test that involves the full closure of concept before testing. Section 5 presents a partial-closure canonicity test that avoids the need for full closure before testing and in Section 6 the partial-closure test is realised in the In-CloseI algorithm. Section 7 compares and evaluates the efficiency of the algorithms by carrying out 'level playing field' implementations and experiments on variety of data sets. Another recent advance in CbO-type algorithms, that allows the inheritance of failed canonicity tests in the recursion, is included in the evaluations. Finally, the paper makes its conclusions and suggestions for further work in Section 8.

2 Formal Concepts

A description of formal concepts [12] 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:

$$A^{\uparrow} := \{\ y \in Y \mid \forall x \in A : xIy\ \}. \tag{1}$$

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:

$$B^{\downarrow} := \{\ x \in X \mid \forall y \in B : xIy\ \}. \tag{2}$$

(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:

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

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 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, compounded by the fact 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: $\{b,d\} = \{1\}^{\downarrow} = \{1,2\}^{\downarrow} = \{1,2,3\}^{\downarrow} = \{1,4\}^{\downarrow} = \{2,4\}^{\downarrow} = \{4\}^{\downarrow}$.

Thus determining that a concept is a repeat is a key halting condition of the algorithm if we are only interested in obtaining unique concepts. The algorithm `ComputeConceptsOnce`, below, illustrates this approach by intersecting a current extent A with successive attribute closures (successive 'columns' in the context), closing the resulting 'candidate' new extent C and then testing the newness of the resulting concept. If the concept passes the test it is processed in some way (stored for example) and the algorithm called again, passing the new extent and the next attribute to the next level of recursion. The algorithm is invoked with an extent and a starting attribute, initially $(X, 0)$.

```
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 [23], 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 [13]. The problem with these approaches is that even an efficient search becomes expensive when there is a large number of repeated concepts.

Ganter noticed [11] that the basic algorithm recursively iterated attribute combinations in the natural combinatorial order, or *canon*. A concept is thus canonical if, when it is computed, its rank comes after the previous canonical concept. For example, the concepts in Section 2, above, are 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. If the previous concept to be computed had the intent $\{0,3,4\}$ and the next concept had the intent $\{0,2\}$, it would be rejected as a repeat: it will have been generated earlier in the computation. It is this canonicity, and the testing thereof, that has become fundamental 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) [19, 21] an efficient means of achieving this with a time complexity of $O(|G||M|^2|L|)$, where L is the set of concepts, or $O(|M|^2|G||L|)$ if the main cycle of the algorithm is over attributes instead of objects, as in the CbO-type algorithms presented here.

CbO has thus formed the basis for several variants and improvements [1, 2, 16, 17, 26] and it has been shown, in numerous tests, that these CbO-type algorithms are significantly faster than previous algorithms [1, 2, 6, 15–17, 22, 26, 28], outperforming algorithms including Chein [9], Norris [24], Next-Closure [11], Bordat [7], Godin [13], Nourine [25] and Add-Intent [30].

4 CbO Algorithm [16]

In the CbO algorithm of Krajca, Outrata and Vychodil [16], 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:

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

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. The correctness of the original CbO algorithm is given in [19], and proof of the canonicity test has been shown in [16].

The algorithm is written below and is called `ComputeConceptsFrom`. The procedure is invoked with the initial concept $(A, B) = (X, X^{\uparrow})$ and 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 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.

CbO call tree - Figure 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). The lower part of a box shows the intersections carried out in the computation of the corresponding concept: the empty-square arrows are the extent intersections, $A \cap \{j\}^{\downarrow}$, and the filled-circle arrows are the closure intersections in $C^{\uparrow}$. In each case the number pointed to represents the attribute involved. Note that each concept, new or repeat, requires a full closure. The notation $< C_x, j >$ represents the invocation of the next level in the form of concept and corresponding initial attribute. The numbered lines connecting the boxes represent the iteration of j in the main cycle.

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.

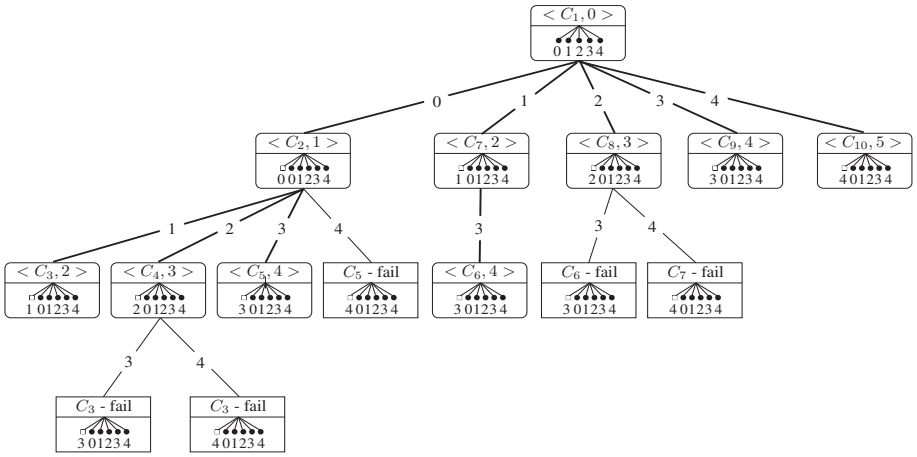

Fig. 1. CbO Call Tree

5 A partial-closure canonicity test

In the CbO algorithm, above, a concept is fully closed before the canonicity test is applied to it. However, it is sufficient to close a concept only up to the current attribute j to determine its canonicity. Remember that the canonicity test in CbO compares the previous and new intent to see that they agree in all attributes *up to the current attribute*. Attributes after the current one have no bearing on the canonicity.

Thus in designing a partial-closure canonicity test we define, from 1 and 4, a *partial-closure* operator $^{\uparrow_j}$ as a modification of the original closure operator:

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

The partial-closure canonicity test will thus determine if the attributes in the intent B, up to j, agree with the attributes in the closure of C up to j and can be defined from 5 and 3 as:

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

If there is an attribute in $C^{\uparrow_j}$ that is in B before j then the concept is not canonical and is a repeat.

The efficiency is 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 $< n$.

In fact, the implementation of the partial closure test can be even more efficient because the closure can be halted *as soon as* an attribute is found that is in B before j, so the cost of the closure will actually be $< j$ for each test failure.

Of course, if the canonicity test is passed, new concepts still need to be fully closed and this can be carried out in the In-Close type of CbO algorithms.

6 In-CloseI algorithm with partial-closure test

This section gives an updated version of the algorithm given in [1]. The key difference to the CbO algorithm in Section 4, above, is that the canonicity test is applied *before* a concept is fully closed. The closure of a concept that passes the test is completed at the next level of recursion, by adding the current attribute j to the intent B whenever the current extent A is found: i.e. whenever $A \cap \{j\}^{\downarrow} = A$. This test of equality can be implemented with almost zero additional cost, given that the intersection $A \cap \{j\}^{\downarrow}$ is carried out in any case. Whenever a new concept is detected, the current, partial, intent is passed to the next level. The intent is then fully closed when the main cycle at the next level is completed.

The algorithm, given as `In-CloseI` below, is invoked with the initial concept $(A, B) = (X, \emptyset)$ and initial attribute $y = 0$.

In-CloseI

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

```
1 for j ← y upto n − 1 do
2 |   C ← A ∩ {j}↓
3 |   if A = C then
4 |   |   B ← B ∪ {j}
5 |   else
6 |   |   if B = C^↑j then
7 |   |   |   D ← B ∪ {j}
8 |   |   |   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.

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 partial 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 the closure of D can be completed.

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-CloseI - CbO and In-CloseI 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-CloseI 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-CloseI 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):

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

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-CloseI call tree - Figure 2 shows the In-CloseI 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). The notation $< C_x, j >$ is the invocation of the next level in the form of concept (extent and *partial* intent) and corresponding initial attribute. The numbered lines connecting the boxes represent the iteration of j in the main cycle.

The computations are now partial with intents being completed at the next level. The lower part of a box, as in the CbO tree, shows the intersections carried out in the computation of the corresponding concept: the empty-square arrows are the extent intersections $A \cap \{j\}^{\downarrow}$ and the filled-circle arrows are the closure intersections in $C^{\uparrow_j}$. In each case the number pointed to represents the attribute involved. Note that, now, each concept is closed only up to j. Thus the In-Close call tree shows fewer closure intersections than CbO. If the canonicity test is passed the intent is closed at the next level, requiring some additional extent intersections. Altogether there are 59 intersections in the tree, 30 fewer than for CbO.

Table 1 summarises 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 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 and create the call-trees, 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].

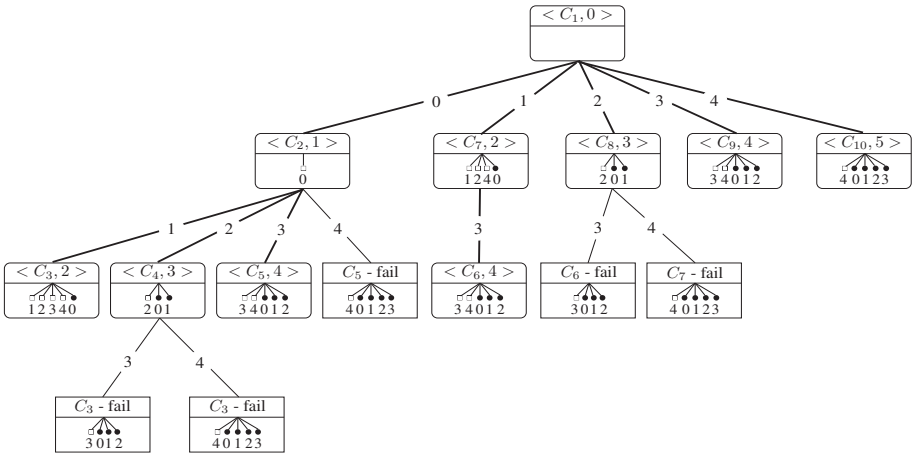

Fig. 2. In-Close Call Tree

Table 1. 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-CloseI		14 (37)	22	59

7 Performance Evaluation

7.1 Incorporating the partial-closure test into FCbO

Another significant advance in CbO-type algorithms was made with the FCbO algorithm [17, 26]. FCbO is an enhancement of CbO where a failed canonicity test is inherited by the next level of recursion. Thus an attribute that has caused a previous failure can be skipped in subsequent levels, avoiding unnecessary closures. To provide a performance evaluation that included this feature it was implemented in In-CloseI to produce In-CloseII.

7.2 Implementations

Implementations were carried out using C++ and a series of tests were performed 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 2.39GHz processor and 3GB 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 comparison with times presented elsewhere would have been 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 [8]. 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 [16].

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 the first attribute that is not canonical. Thus the partial-closure can be halted as soon as such an attribute is found.

The inherited canonicity failure required the creation of a two-dimensional array to implement the failed intents that are passed to the next level of recursion. Although the use of pointers reduces the need for copying arrays in memory, the updating and accessing of the arrays gives rise to some additional complexity in the computation.

The notional `ProcessConcept` procedure was implemented simply by storing the computed concepts. Extents were stored as 'end-to-end' lists of integers in a one-dimensional array to make it possible to store them in the memory available. 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 implemented simply as `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 the equality of C and A. In effect the test becomes $if\ |A| = |C|$, incurring no additional overhead during the completion of partial-closures.

For the closure $C^{\uparrow}$ and partial-closure $C^{\uparrow_j}$, 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, as soon as C was found in a column before j and not in B, the partial-closure was halted.

7.3 Data set experiments

A series of experiments were carried out to compare the performance of the algorithm implementations using a variety of real, artificial and randomised data sets. These provided a wide range of size and density of formal context to test the implementations under a variety of conditions. Three of the real data sets are from the UCI Machine Learning Repository [10]: *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 results are given in Table 2.

Artificial data sets were used that, although partly randomised, were constrained by properties of real data sets, such as many valued attributes with a fixed number of possible values. The results of the artificial data set experiments are given in Table 3.

Three series of random data experiments were carried out, testing the affect of changes in the number of attributes, context density, and number of objects. The results are shown in Figure 3.

Table 2. 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	2,276,0243
CbO	0.66	3.06	0.56	32.68
In-CloseI	0.40	1.65	0.12	11.42
FCbO	0.35	2.06	0.21	17.20
In-CloseII	**0.29**	**1.62**	**0.10**	**9.38**

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

	M7X10G120K	M10X30G120K	T10I4D100K
$\|G\| \times \|M\|$	$120,000 \times 70$	$120,000 \times 300$	$100,000 \times 1,000$
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-CloseI	**1.26**	18.95	16.02
FCbO	1.67	22.33	29.41
In-CloseII	1.39	**10.42**	**11.04**

8 Conclusions and Further Work

The results of the performance experiments suggest that the partial-closure canonicity test significantly improves the efficiency of CbO-type algorithms. Not only was In-CloseI faster than the basic CbO algorithm, in most cases it was faster than FCbO. Although FCbO can significantly reduce the number of closures carried out [26] this appears to be outweighed by the savings in intersections made by partial-closure.

The results also show that further efficiency is possible by combining advances in CbO: combining the inherited canonicity test failure of FCbO with the partial closure canonicity test of In-CloseI to create In-CloseII. In most cases In-CloseII was faster than In-CloseI, but it is interesting that for the objects series of randomised experiments the saving was relatively small and for the artificial data set M7X10G120K, In-CloseI was actually faster. It is quite possible that the complexity of the inherited failure feature is resulting in some significant overheads in the computation, but further work is required to confirm and investigate this suspicion. It may be enlightening to compare the effort saved by the reduction of intersection operations with the increased effort of passing, updating and accessing the inherited failed intents.

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) [16, 17] but not as yet for the In-Close variants.

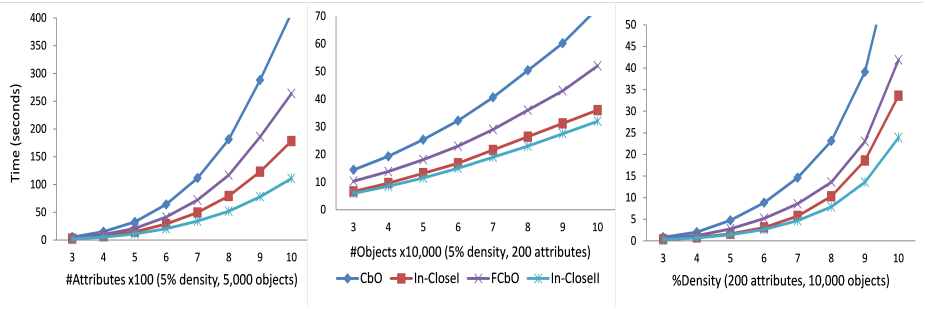

Fig. 3. Comparison of performance with varying number of attributes, density and objects.

Implementations of In-Close and FCbO are available open-source at *SourceForge* [4, 27].

References

1. S. Andrews. In-close, a fast algorithm for computing formal concepts. In S. Rudolph, F. Dau, and S. O. Kuznetsov, editors, *ICCS 2009*, volume 483 of *CEUR WS*. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-483/, 2009.
2. S. Andrews. In-close2, a high performance formal concept miner. In S. Andrews, S. Polovina, R. Hill, and B. Akhgar, editors, *Conceptual Structures for Discovering Knowledge - Proceedings of the 19th International Conference on Conceptual Structures (ICCS)*, pages 50–62. Springer, 2011.
3. S. Andrews. Appendix to: A partial-closure canonicity test to increase the efficiency of cbo-type algorithms, 2013.
4. S. Andrews. In-close program, 2013.
5. S. Andrews and C. Orphanides. Analysis of large data sets using formal concept lattices. In Kryszkiewicz and Obiedkov [18], pages 104–115.
6. D. Borchman. A generalized next-closure algorithm - enumerating semilattice elements from a generating set. pages 9–20. Universidad de Malaga, 2012.
7. J. P. Bordat. Calcul pratique du treillis de galois dune correspondance. *Math. Sci. Hum.*, 96:31–47, 1986.
8. C. Carpineto and G. Romano. *Concept Data Analysis: Theory and Applications*. J. Wiley, 2004.
9. M. Chein. Algorithme de recherche des sous-matrices premires dune matrice. *Bull. Math. Soc. Sci. Math. R.S. Roumanie*, 13:21–25, 1969.
10. A. Frank and A. Asuncion. UCI machine learning repository: http://archive.ics.uci.edu/ml, 2010.
11. B. Ganter. Two basic algorithms in concept analysis. FB4-Preprint 831, TH Darmstadt, 1984.
12. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1998.

13. R. Godin, R. Missaoui, and H Alaoui. Incremental concept formation algorithms based on galois lattices. *Computational Intelligence*, 11(2):246–267, 1995.
14. M. Kaytoue, S. Duplessis, S.O. Kuznetsov, and A. Napoli. Two fca-based methods for mining gene expression data. In S. Ferre and S. Rudolph, editors, *ICFCA 2009*, volume 5548 of *LNAI*. Springer, 2009.
15. M. Kirchberg, E. Leonardi, Y. S. Tan, S. Link, R. K. L. Ko, and B. S. Lee. Formal concept discovery in semantic web data. volume 7278 of *LNAI*. Springer-Verlag Berlin Heidelberg, 2012.
16. P. Krajca, J. Outrata, and V. Vychodil. Parallel recursive algorithm for fca. In R. Belohavlek and S.O. Kuznetsov, editors, *CLA 2008*, 2008.
17. P. Krajca, V. Vychodil, and J. Outrata. Advances in algorithms based on cbo. In Kryszkiewicz and Obiedkov [18], pages 325–337.
18. M. Kryszkiewicz and S. Obiedkov, editors. *7th International Conference on Concept Lattices and Their Applications, CLA 2010, Seville*. University of Sevilla, 2010.
19. O. Kuznetsov, S. A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic Documentation and Mathematical Linguistics*, 27(5):11–21, 1993.
20. S. O. Kuznetsov. Interpretation on graphs and complexity characteristics of a search for specific patterns. *Automatic Documentation and Mathematical Linguistics*, 24(1):37–45, 1989.
21. S.O. Kuznetsov. Learning of simple conceptual graphs from positive and negative examples. In J.M. Zytkow and J. Rauch, editors, *PKDD'99*, volume 1704 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 1999.
22. S.O. Kuznetsov and S.A. Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14:189–216, 2002.
23. C. Lindig. Fast concept analysis. In *Working with conceptual structures: Contributions to ICCS 2000*, pages 152–161. Aachen: Shaker Verlag, 2000.
24. M. Norris, E. An algorithm for computing the maximal recatangles in a binary relation. *Revue Roumanine de Mathématiques Pures et Appliquées*, 23(2):243–250, 1978.
25. L. Nourine and O. Raynaud. A fast algorithm for building lattices. *Information Procesing Letters*, 71:199–204, 1999.
26. Jan Outrata and Vilem Vychodil. Fast algorithm for computing fixpoints of galois connections induced by object-attribute relational data. *Inf. Sci.*, 185(1):114–127, February 2012.
27. Krajca P., Outrata J., and Vychodil V. Fcbo program, 2012.
28. F. Strok and 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*, pages 417–420, 2010.
29. T. Tanabata, K. Sawase, H. Nobuhara, and B. Bede. Interactive data mining for image databases based on fca. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 14(3):303–308, 2010.
30. D. Van der Merwe, S.A. Obiedkov, and Kourie D.G. Addintent: A new incremental algorithm for constructing concept lattices. In P. Eklund, editor, *ICFCA 2004*, volume 2961 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2004.