Comparing Performance of Algorithms for Generating the Duquenne–Guigues Basis

Konstantin Bazhanov and Sergei Obiedkov

Higher School of Economics, Moscow, Russia,
kostyabazhanov@mail.ru, sergei.obj@gmail.com

Abstract. In this paper, we take a look at algorithms involved in the computation of the Duquenne–Guigues basis of implications. The most widely used algorithm for constructing the basis is Ganter's Next Closure, designed for generating closed sets of an arbitrary closure system. We show that, for the purpose of generating the basis, the algorithm can be optimized. We compare the performance of the original algorithm and its optimized version in a series of experiments using artificially generated and real-life datasets. An important computationally expensive subroutine of the algorithm generates the closure of an attribute set with respect to a set of implications. We compare the performance of three algorithms for this task on their own, as well as in conjunction with each of the two versions of Next Closure.

1 Introduction

Implications are among the most important tools of formal concept analysis (FCA) [9]. The set of all attribute implications valid in a formal context defines a closure operator mapping attribute sets to concept intents of the context (this mapping is surjective). The following two algorithmic problems arise with respect to implications:

1. Given a set $\mathcal{L}$ of implications and an attribute set A, compute the closure $\mathcal{L}(A)$.
2. Given a formal context $\mathbb{K}$, compute a set of implications equivalent to the set of all implications valid in $\mathbb{K}$, i.e., the *cover* of valid implications.

The first of these problems has received considerable attention in the database literature in application to functional dependencies [14]. Although functional dependencies are interpreted differently than implications, the two are in many ways similar: in particular, they share the notion of semantic consequence and the syntactic inference mechanism (Armstrong rules [1]). A linear-time algorithm, LinClosure, has been proposed for computing the closure of a set with respect to a set of functional dependencies (or implications) [3], i.e., for solving the first of the two problems stated above. However, the asymptotic complexity estimates may not always be good indicators for relative performance of algorithms in practical situations. In Sect. 3, we compare LinClosure with two

other algorithms—a "naïve" algorithm, Closure [14], and the algorithm proposed in [20]—both of which are non-linear. We analyze their performance in several particular cases and compare them experimentally on several datasets.

For the second problem, an obvious choice of the cover is the Duquenne–Guigues, or canonical, basis of implications, which is the smallest set equivalent to the set of valid implications [11]. Unlike for the other frequently occurring FCA algorithmic task, the computation of all formal concepts of a formal context [12], only few algorithms have been proposed for the calculation of the canonical basis. The most widely-used algorithm was proposed by Ganter in [10]. Another, attribute-incremental, algorithm for the same problem was described in [17]. It is claimed to be much faster than Ganter's algorithm for most practical situations. The Concept Explorer software system [21] uses this algorithm to generate the Duquenne–Guigues basis of a formal context. However, we do not discuss it here, for we choose to concentrate on the computation of implications in the lectic order (see Sect. 4). The lectic order is important in the interactive knowledge-acquisition procedure of attribute exploration [8], where implications are output one by one and the user is requested to confirm or reject (by providing a counterexample) each implication.

Ganter's algorithm repeatedly computes the closure of an attribute set with respect to a set of implications; therefore, it relies heavily on a subprocedure implementing a solution to the first problem. In Sect. 4, we describe possible optimizations of Ganter's algorithm and experimentally compare the original and optimized versions in conjunction with each of the three algorithms for solving the first problem. A systematic comparison with the algorithm from [17] is left for further work.

2 The Duquenne–Guigues Basis of Implications

Before proceeding, we quickly recall the definition of the Duquenne–Guigues basis and related notions.

Given a *(formal) context* $\mathbb{K} = (G, M, I)$, where G is called a set of *objects*, M is called a set of *attributes*, and the binary relation $I \subseteq G \times M$ specifies which objects have which attributes, the derivation operators $(\cdot)^I$ are defined for $A \subseteq G$ and $B \subseteq M$ as follows:

$$A' = \{m \in M \mid \forall g \in A : gIm\} \qquad B' = \{g \in G \mid \forall m \in B : gIm\}$$

In words, A' is the set of attributes common to all objects of A and B' is the set of objects sharing all attributes of B. The double application of $(\cdot)'$ is a closure operator, i.e., $(\cdot)''$ is extensive, idempotent, and monotonous. Therefore, sets A'' and B'' are said to be *closed*. Closed object sets are called *concept extents* and closed attribute sets are called *concept intents* of the formal context $\mathbb{K}$.

In discussing the algorithms later in the paper, we assume that the sets G and M are finite.

An *implication* over M is an expression $A \to B$, where $A, B \subseteq M$ are attribute subsets. It *holds* in the context if $A' \subseteq B'$, i.e., every object of the context that has all attributes from A also has all attributes from B.

An attribute subset $X \subseteq M$ *respects* (or is a *model* of) an implication $A \to B$ if $A \not\subseteq X$ or $B \subseteq X$. Obviously, an implication holds in a context (G, M, I) if and only if $\{g\}'$ respects the implication for all $g \in G$.

A set $\mathcal{L}$ of implications over M defines the closure operator $X \mapsto \mathcal{L}(X)$ that maps $X \subseteq M$ to the smallest set respecting all the implications in $\mathcal{L}$:

$$\mathcal{L}(X) = \bigcap \{Y \mid X \subseteq Y \subseteq M, \forall (A \to B) \in \mathcal{L} : A \not\subseteq Y \text{ or } B \subseteq Y\}.$$

We discuss algorithms for computing $\mathcal{L}(X)$ in Sect. 3. Note that, if $\mathcal{L}$ is the set of all valid implications of a formal context, then $\mathcal{L}(X) = X''$ for all $X \subseteq M$.

Two implication sets over M are *equivalent* if they are respected by exactly the same subsets of M. Equivalent implication sets define the same closure operator. A *minimum cover* of an implication set $\mathcal{L}$ is a set of minimal size among all implication sets equivalent to $\mathcal{L}$. One particular minimum cover described in [11] is defined using the notion of a pseudo-closed set, which we introduce next.

A set $P \subseteq M$ is called *pseudo-closed* (with respect to a closure operator $(\cdot)''$) if $P \neq P''$ and $Q'' \subset P$ for every pseudo-closed $Q \subset P$.

In particular, all minimal non-closed sets are pseudo-closed. A pseudo-closed attribute set of a formal context is also called a *pseudo-intent*.

The *Duquenne–Guigues* or *canonical basis* of implications (with respect to a closure operator $(\cdot)''$) is the set of all implications of the form $P \to P''$, where P is pseudo-closed. This set of implications is of minimal size among those defining the closure operator $(\cdot)''$. If $(\cdot)''$ is the closure operator associated with a formal context, the Duquenne–Guigues basis is a minimum cover of valid implications of this context. The computation of the Duquenne–Guigues basis of a formal context is hard, since even recognizing pseudo-intents is a coNP-complete problem [2], see also [13, 7]. We discuss algorithms for computing the basis in Sect. 4.

3 Computing the Closure of an Attribute Set

In this section, we compare the performance of algorithms computing the closure of an attribute set X with respect to a set $\mathcal{L}$ of implications. Algorithm 1 [14] checks every implication $A \to B \in \mathcal{L}$ and enlarges X with attributes from B if $A \subseteq X$. The algorithm terminates when a fixed point is reached, that is, when the set X cannot be enlarged any further (which always happens at some moment, since both $\mathcal{L}$ and M are assumed finite).

The algorithm is obviously quadratic in the number of implications in $\mathcal{L}$ in the worst case. The worst case happens when exactly one implication is applied at each iteration (but the last one) of the **repeat** loop, resulting in $|\mathcal{L}|(|\mathcal{L}|+1)/2$ iterations of the **for all** loop, each requiring $O(|M|)$ time.

Example 1. A simple example is when $X = \{1\}$ and the implications in $\mathcal{L} = \{\{i\} \to \{i+1\} \mid i \in \mathbb{N}, 0 < i < n\}$ for some n are arranged in the descending order of their one-element premises.

Algorithm 1 CLOSURE$(X, \mathcal{L})$

Input: An attribute set $X \subseteq M$ and a set $\mathcal{L}$ of implications over M.
Output: The closure of X w.r.t. implications in $\mathcal{L}$.

```
repeat
    stable := true
    for all A → B ∈ L do
        if A ⊆ X then
            X := X ∪ B
            stable := false
            L := L \ {A → B}
until stable
return X
```

In [3], a linear-time algorithm, LINCLOSURE, is proposed for the same problem. Algorithm 2 is identical to the version of LINCLOSURE from [14] except for one modification designed to allow implications with empty premises in $\mathcal{L}$. LINCLOSURE associates a counter with each implication initializing it with the size of the implication premise. Also, each attribute is linked to a list of implications that have it in their premises. The algorithm then checks every attribute m of X (the set whose closure must be computed) and decrements the counters for all implications linked to m. If the counter of some implication $A \to B$ reaches zero, attributes from B are added to X. Afterwards, they are used to decrement counters along with the original attributes of X. When all attributes in X have been checked in this way, the algorithm stops with X containing the closure of the input attribute set.

It can be shown that the algorithm is linear in the length of the input assuming that each attribute in the premise or conclusion of any implication in $\mathcal{L}$ requires a constant amount of memory [14].

Example 2. The worst case for LINCLOSURE occurs, for instance, when $X \subset \mathbb{N}$, $M = X \cup \{1, 2, \ldots, n\}$ for some n such that $X \cap \{1, 2, \ldots, n\} = \varnothing$ and $\mathcal{L}$ consists of implications of the form

$$X \cup \{i \mid 0 < i < k\} \to \{k\}$$

for all k such that $1 \leq k \leq n$. During each of the first $|X|$ iterations of the **for all** loop, the counters of all implications will have to be updated with only the last iteration adding one attribute to X using the implication $X \to \{1\}$. At each of the subsequent $n - 1$ iterations, the counter for every so far "unused" implication will be updated and one attribute will be added to X. The next, $(|X| + n)$th, iteration will terminate the algorithm.

Note that, if the implications in $\mathcal{L}$ are arranged in the superset-inclusion order of their premises, this example will present the worst case for Algorithm 1 requiring n iterations of the main loop. However, if the implications are arranged in the subset-inclusion order of their premises, one iteration will be sufficient.

Inspired by the mechanism used in LINCLOSURE to obtain linear asymptotic complexity, but somewhat disappointed by the poor performance of the

Algorithm 2 LinClosure$(X, \mathcal{L})$

Input: An attribute set $X \subseteq M$ and a set $\mathcal{L}$ of implications over M.
Output: The closure of X w.r.t. implications in $\mathcal{L}$.

```
for all A → B ∈ L do
    count[A → B] := |A|
    if |A| = 0 then
        X := X ∪ B
    for all a ∈ A do
        add A → B to list[a]
update := X
while update ≠ ∅ do
    choose m ∈ update
    update := update \ {m}
    for all A → B ∈ list[m] do
        count[A → B] = count[A → B] − 1
        if count[A → B] = 0 then
            add := B \ X
            X := X ∪ add
            update := update ∪ add
return X
```

algorithm relative to Closure, which was revealed in his experiments, Wild proposed a new algorithm in [20]. We present this algorithm (in a slightly more compact form) as Algorithm 3. The idea is to maintain implication lists similar to those used in LinClosure, but get rid of the counters. Instead, at each step, the algorithm combines the implications in the lists associated with attributes not occurring in X and "fires" the remaining implications (i.e., uses them to enlarge X). When there is no implication to fire, the algorithm terminates with X containing the desired result.

Wild claims that his algorithm is faster than both LinClosure and Closure, even though it has the same asymptotic complexity as the latter. The worst case for Algorithm 3 is when $\mathcal{L} \setminus \mathcal{L}_1$ contains exactly one implication $A \to B$ and $B \setminus X$ contains exactly one attribute at each iteration of the **repeat ... until** loop. Example 1 presents the worst case for 3, but, unlike for Closure, the order of implications in $\mathcal{L}$ is irrelevant. The worst case for LinClosure (see Example 2) is also the worst case for Algorithm 3, but it deals with it, perhaps, in a more efficient way using n iterations of the main loop compared to $n + |X|$ iterations of the main loop in LinClosure.

Experimental Comparison

We implemented the algorithms in C++ using Microsoft Visual Studio 2010. For the implementation of attribute sets, as well as sets of implications in Algorithm 3, we used dynamic bit sets from the Boost library [6]. All the tests described in the following sections were carried out on an Intel Core i5 2.67 GHz computer with 4 Gb of memory running under Windows 7 Home Premium x64.

Algorithm 3 WILD'S CLOSURE$(X, \mathcal{L})$

Input: An attribute set $X \subseteq M$ and a set $\mathcal{L}$ of implications over M.
Output: The closure of X w.r.t. implications in $\mathcal{L}$.

```
for all m ∈ M do
    for all A → B ∈ L do
        if m ∈ A then
            add A → B to list[m]
repeat
    stable := true
    L1 := ⋃_{m∈M\X} list[m]
    for all A → B ∈ L \ L1 do
        X := X ∪ B
        stable := false
    L := L1
until stable
return X
```

Figure 1 shows the performance of the three algorithms on Example 1. Algorithm 2 is the fastest algorithm in this case: for a given n, it needs n iterations of the outer loop—the same as the other two algorithms, but the inner loop of Algorithm 2 checks exactly one implication at each iteration, whereas the inner loop of Algorithm 1 checks $n - i$ implications at the ith iteration. Although the inner loop of Algorithm 3 checks only one implication at the ith iteration, it has to compute the union of $n - i$ lists in addition.

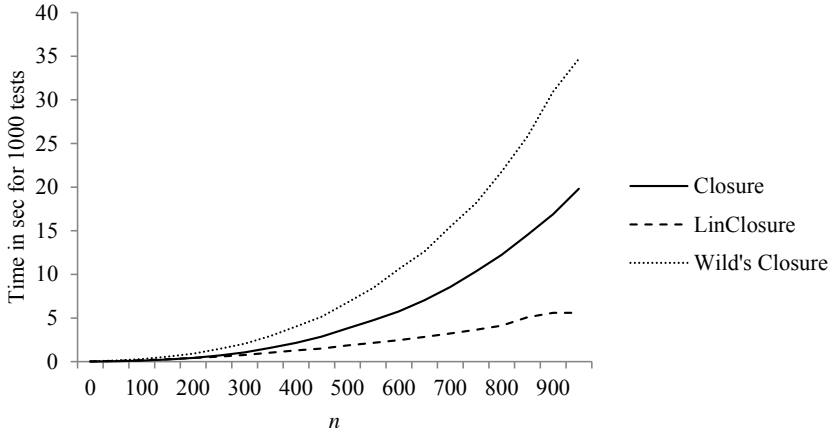


Fig. 1. The performance of Algorithms 1–3 for Example 1.

Figure 2 shows the performance of the algorithms on Example 2. Here, the behavior of Algorithm 2 is similar to that of Algorithm 1, but Algorithm 2 takes more time due to the complicated initialization step.

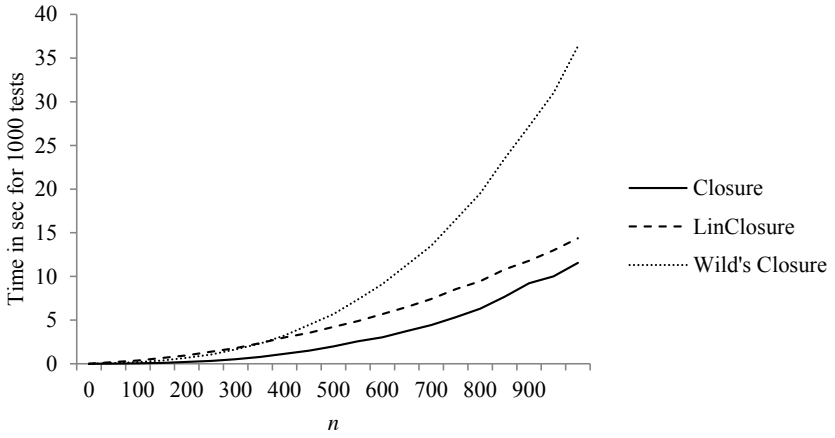


Fig. 2. The performance of Algorithms 1–3 for Example 2 with implications in $\mathcal{L}$ arranged in the superset-inclusion order of their premises and $|X| = 50$.

Interestingly, Algorithm 1 works amost twice as fast on Example 2 as it does on Example 1. This may seem surprising, since it is easy to see that the algorithm performs essentially the same computations in both cases, the difference being that the implications of Example 1 have single-element premises. However, this turns out to be a source of inefficiency: at each iteration of the main loop, all implications but the last fail to fire, but, for each of them, the algorithm checks if their premises are included in the set X. Generally, when $A \not\subseteq X$, this can be established easier if A is large, for, in this case, A is likely to contain more elements outside X. This effect is reinforced by the implementation of sets as bit strings: roughly speaking, to verify that $\{i\} \not\subseteq \{1\}$, it is necessary to check all bits up $\{i\}$, whereas $\{i \mid 0 < i < k\} \not\subseteq \{k+1\}$ can be established by checking only one bit (assuming that bits are checked from left to right). Alternative data structures for set implementation might have less dramatic consequences for performance in this setting. On the other hand, the example shows that performance may be affected by issues not so obviously related to the structure of the algorithm, thus, suggesting additional paths to obtain an optimal behavior (e.g., by rearranging attributes or otherwise preprocessing the input data).

We have experimented with computing closures using the Duquenne–Guigues bases of formal contexts as input implication sets. Table 1 shows the results for randomly generated contexts. The first two columns indicate the size of the attribute set and the number of implications, respectively. The remaining three columns record the time (in seconds) for computing the closures of 1000 ran-

domly generated subsets of M by each of the three algorithms. Table 3 presents similar results for datasets taken from the UCI repository [5] and, if necessary, transformed into formal contexts using FCA scaling [9].[1] The contexts are described in Table 2, where the last four columns correspond to the number of objects, number of attributes, number of intents, and number of pseudo-intents (i.e., the size of the canonical basis) of the context named in the first column.

Table 1. Performance on randomly generated tests (time in seconds per 1000 closures)

		Algorithm		
$\|M\|$	$\|\mathcal{L}\|$	1	2	3
30	557	0.0051	0.2593	0.0590
50	1115	0.0118	0.5926	0.1502
100	380	0.0055	0.2887	0.0900
100	546	0.0086	0.4229	0.1350
100	2269	0.0334	1.5742	0.5023
100	3893	0.0562	2.6186	0.8380
100	7994	0.1134	5.3768	1.7152
100	8136	0.1159	5.6611	1.8412

Table 2. Contexts obtained from UCI datasets

Context	$\|G\|$	$\|M\|$	# intents	# pseudo-intents
Zoo	101	28	379	141
Postoperative Patient	90	26	2378	619
Congressional Voting	435	18	10644	849
SPECT	267	23	21550	2169
Breast Cancer	286	43	9918	3354
Solar Flare	1389	49	28742	3382
Wisconsin Breast Cancer	699	91	9824	10666

In these experiments, Algorithm 1 was the fastest and Algorithm 2 was the slowest, even though it has the best asymptotic complexity. This can be partly explained by the large overhead of the initialization step (setting up counters and implication lists). Therefore, these results can be used as a reference only when the task is to compute one closure for a given set of implications. When

[1] The breast cancer domain was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia (now, Slovenia). Thanks go to M. Zwitter and M. Soklic for providing the data.

a large number of closures must be computed with respect to the same set of implications, Algorithms 2 and 3 may be more appropriate.

Table 3. Performance on the canonical bases of contexts from Table 2 (time in seconds per 1000 closures)

Context	Algorithm 1	2	3
Zoo	0.0036	0.0905	0.0182
Postoperative Patient	0.0054	0.2980	0.0722
Congressional Voting	0.0075	0.1505	0.0883
SPECT	0.0251	0.9848	0.2570
Breast Cancer	0.0361	1.7912	0.5028
Solar Flare	0.0370	2.1165	0.6317
Wisconsin Breast Cancer	0.1368	8.4984	2.4730

4 Computing the Basis in the Lectic Order

The best-known algorithm for computing the Duquenne–Guigues basis was developed by Ganter in [10]. The algorithm is based on the fact that intents and pseudo-intents of a context taken together form a closure system. This makes it possible to iteratively generate all intents and pseudo-intents using Next Closure (see Algorithm 4), a generic algorithm for enumerating closed sets of an arbitrary closure operator (also proposed in [10]). For every generated pseudo-intent P, an implication $P \to P''$ is added to the basis. The intents, which are also generated, are simply discarded.

Algorithm 4 Next Closure(A, M, $\mathcal{L}$)

Input: A closure operator $X \mapsto \mathcal{L}(X)$ on M and a subset $A \subseteq M$.
Output: The lectically next closed set after A.

```
for all m ∈ M in reverse order do
    if m ∈ A then
        A := A \ {m}
    else
        B := L(A ∪ {m})
        if B \ A contains no element < m then
            return B
return ⊥
```

Next Closure takes a closed set as input and outputs the next closed set according to a particular *lectic* order, which is a linear extension of the subset-

inclusion order. Assuming a linear order $<$ on attributes in M, we say that a set $A \subseteq M$ is *lectically smaller* than a set $B \subseteq M$ if

$$\exists b \in B \setminus A \; \forall a \in A (a < b \Rightarrow a \in B).$$

In other words, the lectically largest among two sets is the one containing the smallest element in which they differ.

Example 3. Let $M = \{a < b < c < d < e < f\}$, $A = \{a, c, e\}$ and $B = \{a, b, f\}$. Then, A is lectically smaller than B, since the first attribute in which they differ, b, is in B. Note that if we represent sets by bit strings with smaller attributes corresponding to higher-order bits (in our example, $A = 101010$ and $B = 110001$), the lectic order will match the usual less-than order on binary numbers.

To be able to use Next Closure for iterating over intents and pseudo-intents, we need access to the corresponding closure operator. This operator, which we denote by $^\bullet$, is defined via the Duquenne–Guigues basis $\mathcal{L}$ as follows.[2] For a subset $A \subseteq M$, put

$$A^+ = A \cup \bigcup \{P'' \mid P \to P'' \in \mathcal{L}, P \subset A\}.$$

Then, $A^\bullet = A^{+ + \cdots +}$, where $A^{\bullet +} = A^\bullet$; i.e., $^\bullet$ is the transitive closure of $^+$.

The problem is that $\mathcal{L}$ is not available when we start; in fact, this is precisely what we want to generate. Fortunately, for computing a pseudo-closed set A, it is sufficient to know only implications with premises that are proper subsets of A. Generating pseudo-closed sets in the lectic order, which is compatible with the subset-inclusion order, we ensure that, at each step, we have at hand the required part of the basis. Therefore, we can use any of the three algorithms from Sect. 3 to compute $A^\bullet$ (provided that the implication $A^\bullet \to A''$ has not been added to $\mathcal{L}$ yet). Algorithm 5 uses Next Closure to generate the canonical basis. It passes Next Closure the part of the basis computed so far; Next Closure may call any of the Algorithms 1–3 to compute the closure, $\mathcal{L}(A \cup \{m\})$, with respect to this set of implications.

After Next Closure computes $A^\bullet$, the implication $A^\bullet \to A''$ may be added to the basis. Algorithm 5 will then pass $A^\bullet$ as the input to Next Closure, but there is some room for optimizations here. Let i be the maximal element of A and j be the minimal element of $A'' \setminus A$. Consider the following two cases:

$j < i$: As long as $m > i$, the set $\mathcal{L}(A^\bullet \cup \{m\})$ will be rejected by Next Closure, since it will contain j. Hence, it makes sense to skip all $m > i$ and continue as if $A^\bullet$ had been rejected by Next Closure. This optimization has already been proposed in [17].

$i < j$: It can be shown that, in this case, the lectically next intent or pseudo-intent after $A^\bullet$ is A''. Hence, A'' could be used at the next step instead of $A^\bullet$.

Algorithm 6 takes these considerations into account.

[2] We deliberately use the same letter $\mathcal{L}$ for an implication set and the closure operator it defines.

Algorithm 5 CANONICAL BASIS(M, $''$)

Input: A closure operator $X \mapsto X''$ on M, e.g., given by a formal context (G, M, I).
Output: The canonical basis for the closure operator.

```
L := ∅
A := ∅
while A ≠ M do
    if A ≠ A'' then
        L := L ∪ {A → A''}
    A := NEXT CLOSURE(A, M, L)
return L
```

Algorithm 6 CANONICAL BASIS(M, $''$), an optimized version

Input: A closure operator $X \mapsto X''$ on M, e.g., given by a formal context (G, M, I).
Output: The canonical basis for the closure operator.

```
L := ∅
A := ∅
i := the smallest element of M
while A ≠ M do
    if A ≠ A'' then
        L := L ∪ {A → A''}
    if A'' \ A contains an element < i then
        A := {m ∈ A | m ≤ i}
    else if A'' = M then
        exit while
    else
        A := A''
        i := the largest element of M
    for all j ≤ i ∈ M in reverse order do
        if j ∈ A then
            A := A \ {j}
        else
            B := L(A ∪ {j})
            if B \ A contains no element < j then
                A := B
                i := j
                exit for
return L
```

Experimental Comparison

We used Algorithms 5 and 6 for constructing the canonical bases of the contexts involved in testing the performance of the algorithms from Sect. 3, as well as the context $(M, M, \neq)$ with $|M| = 18$, which is special in that every subset of M is closed (and hence there are no valid implications). Both algorithms have been tested in conjunction with each of the three procedures for computing closures (Algorithm 1–3). The results are presented in Table 4 and Fig. 3. It can be seen that Algorithm 6 indeed improves on the performance of Algorithm 5. Among the three algorithms computing the closure, the simpler Algorithm 1 is generally more efficient, even though, in our implementation, we do not perform the initialization step of Algorithms 2 and 3 from scratch each time we need to compute a closure of a new set; instead, we reuse the previously constructed counters and implication lists and update them incrementally with the addition of each new implication. We prefer to treat these results as preliminary: it still remains to see whether the asymptotic behavior of LinClosure will give it an advantage over the other algorithms on larger contexts.

Table 4. Time (in seconds) for building the canonical bases of artificial contexts

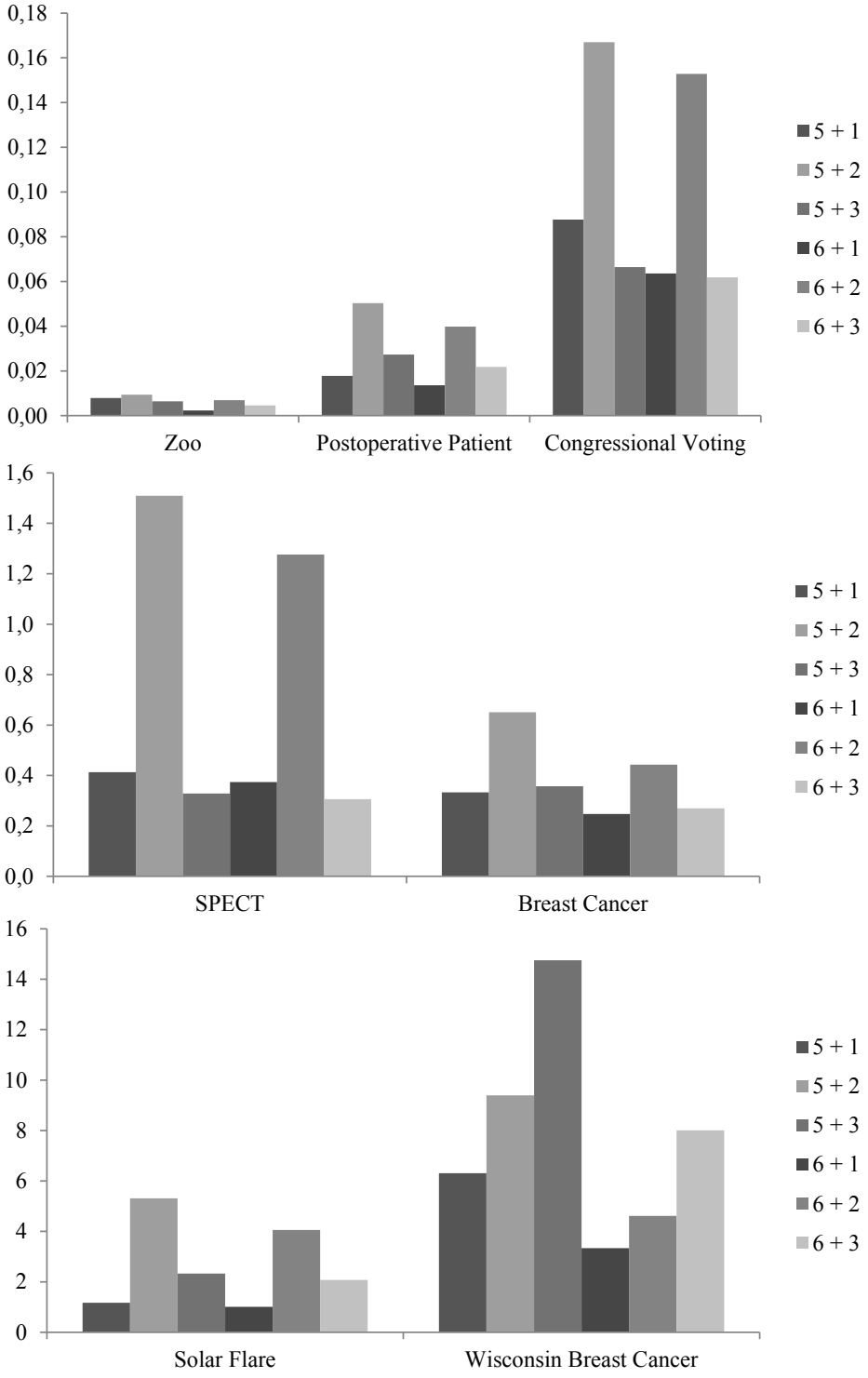
Context	# intents	# pseudo-intents	Algorithm 5 + 1	5 + 2	5 + 3	6 + 1	6 + 2	6 + 3
$100 \times 30, 4$	307	557	0.0088	0.0145	0.0119	0.0044	0.0065	0.0059
$10 \times 100, 25$	129	380	0.0330	0.0365	0.0431	0.0073	0.0150	0.0169
$100 \times 50, 4$	251	1115	0.0442	0.0549	0.0617	0.0138	0.0152	0.0176
$10 \times 100, 50$	559	546	0.0542	0.1312	0.1506	0.0382	0.0932	0.0954
$20 \times 100, 25$	716	2269	0.3814	0.3920	0.7380	0.1219	0.1312	0.2504
$50 \times 100, 10$	420	3893	1.1354	0.7291	1.6456	0.1640	0.1003	0.2299
$900 \times 100, 4$	2472	7994	4.6313	2.7893	6.3140	1.5594	0.8980	2.0503
$20 \times 100, 50$	12394	8136	7.3097	8.1432	14.955	5.1091	6.0182	10.867
$(M, M, \neq)$	262144	0	0.1578	0.3698	0.1936	0.1333	0.2717	0.1656

5 Conclusion

In this paper, we compared the performance of several algorithms computing the closure of an attribute set with respect to a set of implications. Each of these algorithms can be used as a (frequently called) subroutine while computing the Duquenne–Guigues basis of a formal context. We tested them in conjunction with Ganter's algorithm and its optimized version.

In our future work, we plan to extend the comparison to algorithms generating the Duquenne–Guigues basis in a different (non-lectic) order, in particular, to incremental [17] and divide-and-conquer [19] approaches, probably, in conjunction with newer algorithms for computing the closure of a set [16]. In addition,

Fig. 3. Time (in seconds) for building the canonical bases of contexts from Table 2



we are going to consider algorithms that generate other implication covers: for example, direct basis [15, 20, 4] or proper basis [18]. They can be used as an intermediate step in the computation of the Duquenne–Guigues basis. If the number of intents is much larger than the number of pseudo-intents, this two-step approach may be more efficient than direct generation of the Duquenne–Guigues basis with Algorithms 5 or 6, which produce all intents as a side effect.

Acknowledgements

The second author was supported by the Academic Fund Program of the Higher School of Economics (project 10-04-0017) and the Russian Foundation for Basic Research (grant no. 08-07-92497-NTsNIL_a).

References

1. Armstrong, W.: Dependency structure of data base ralationship. Proc. IFIP Congress pp. 580–583 (1974)
2. Babin, M.A., Kuznetsov, S.O.: Recognizing pseudo-intents is coNP-complete. In: Kryszkiewicz, M., Obiedkov, S. (eds.) Proceedings of the 7th International Conference on Concept Lattices and Their Applications. pp. 294–301. University of Sevilla, Spain (2010)
3. Beeri, C., Bernstein, P.: Computational problems related to the design of normal form relational schemas. ACM TODS 4(1), 30–59 (March 1979)
4. Bertet, K., Monjardet, B.: The multiple facets of the canonical direct unit implicational basis. Theor. Comput. Sci. 411(22-24), 2155–2166 (2010)
5. Blake, C., Merz, C.: UCI repository of machine learning databases (1998), http://archive.ics.uci.edu/ml
6. Demming, R., Duffy, D.: Introduction to the Boost C++ Libraries. Datasim Education Bv (2010), see http://www.boost.org
7. Distel, F., Sertkaya, B.: On the complexity of enumerating pseudo-intents. Discrete Appl. Math. 159, 450–466 (March 2011)
8. Ganter, B.: Attribute exploration with background knowledge. Theor. Comput. Sci. pp. 215–233 (1999)
9. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Berlin (1999)
10. Ganter, B.: Two basic algorithms in concept analysis. Preprint 831, Technische Hochschule Darmstadt, Germany (1984)
11. Guigues, J.L., Duquenne, V.: Familles minimales d'implications informatives resultant d'un tableau de donnees binaires. Math. Sci. Hum. 95(1), 5–18 (1986)
12. Kuznetsov, S., Obiedkov, S.: Comparing performance of algorithms for generating concept lattices. Journal of Experimental and Theoretical Artificial Intelligence 14(2/3), 189–216 (2002)
13. Kuznetsov, S.O., Obiedkov, S.: Some decision and counting problems of the Duquenne–Guigues basis of implications. Discrete Appl. Math. 156(11), 1994–2003 (2008)
14. Maier, D.: The theory of relational databases. Computer software engineering series, Computer Science Press (1983)

15. Mannila, H., Räihä, K.J.: The design of relational databases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1992)
16. Mora, A., Aguilera, G., Enciso, M., Cordero, P., de Guzman, I.P.: A new closure algorithm based in logic: SLFD-Closure versus classical closures. Inteligencia Artificial, Revista Iberoamericana de IA 10(31), 31–40 (2006)
17. Obiedkov, S., Duquenne, V.: Attribute-incremental construction of the canonical implication basis. Annals of Mathematics and Artificial Intelligence 49(1-4), 77–99 (April 2007)
18. Taouil, R., Bastide, Y.: Computing proper implications. In Proc. ICCS-2001 International Workshop on Concept Lattices-Based Theory, Methods and Tools for Knowledge Discovery in Databases pp. 290–303 (2001)
19. Valtchev, P., Duquenne, V.: On the merge of factor canonical bases. In: Medina, R., Obiedkov, S. (eds.) ICFCA. Lecture Notes in Computer Science, vol. 4933, pp. 182–198. Springer (2008)
20. Wild, M.: Computations with finite closure systems and implications. In: Computing and Combinatorics. pp. 111–120 (1995)
21. Yevtushenko, S.A.: System of data analysis "Concept Explorer" (in Russian). In: Proceedings of the 7th national conference on Artificial Intelligence KII-2000. pp. 127–134. Russia (2000), http://conexp.sourceforge.net/