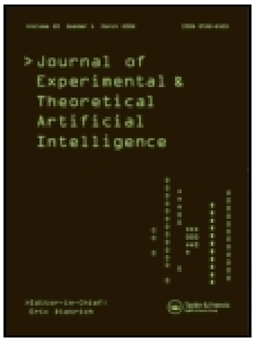


Taylor & Francis
Taylor & Francis Group

Journal of Experimental & Theoretical Artificial Intelligence

ISSN: 0952-813X (Print) 1362-3079 (Online) Journal homepage: http://www.tandfonline.com/loi/teta20

Comparing performance of algorithms for generating concept lattices

Sergei O. Kuznetsov & Sergei A. Obiedkov

To cite this article: Sergei O. Kuznetsov & Sergei A. Obiedkov (2002) Comparing performance of algorithms for generating concept lattices, Journal of Experimental & Theoretical Artificial Intelligence, 14:2-3, 189-216, DOI: 10.1080/09528130210164170

To link to this article: http://dx.doi.org/10.1080/09528130210164170

Published online: 09 Nov 2010.

Comparing performance of algorithms for generating concept lattices

SERGEI O. KUZNETSOV† and SERGEI A. OBIEDKOV‡

† *All Russian Institute for Scientific and Technical Information (VINITI), Moscow, Russia*
e-mail: serge@viniti.ru
‡ *Russian State University for the Humanities, Moscow, Russia*
e-mail: bs-obj@east.ru

Abstract. Recently concept lattices became widely used tools for intelligent data analysis. In this paper, several algorithms that generate the set of all formal concepts and diagram graphs of concept lattices are considered. Some modifications of well-known algorithms are proposed. Algorithmic complexity of the algorithms is studied both theoretically (in the worst case) and experimentally. Conditions of preferable use of some algorithms are given in terms of density/sparseness of underlying formal contexts. Principles of comparing practical performance of algorithms are discussed.

Keywords: concept lattice, algorithms, computation complexity

Received 1 November 2001; accepted 15 January 2002

1. Introduction

Concept (Galois) lattices proved to be a useful tool in many applied domains: machine learning, data mining and knowledge discovery, information retrieval, etc. (Finn 1991, Carpineto and Romano 1996, Stumme *et al.* 1998, Ganter and Kuznetsov 2000, Stumme *et al.* 2000). The problem of generating the set of all concepts and the concept lattice of a formal context is extensively studied in the literature (Chein 1969, Norris 1978, Ganter 1984, Bordat 1986, Guénoche 1990, Dowling 1993, Kuznetsov 1993, Godin *et al.*, 1995, Carpineto and Romano 1996, Mephu Nguifo and Njiwoua 1998, Lindig 1999, Nourine and Raynaud 1999, Stumme *et al.* 2000, Valtchev and Missaoui 2001). It is known that the number of concepts can be exponential in the size of the input context (e.g. when the lattice is a Boolean one) and the problem of determining this number is #P-complete (Kuznetsov 1989, 2001). Therefore, from the standpoint of the worst-case complexity, an algorithm generating all concepts and/or a concept lattice can be considered optimal if it generates the lattice with polynomial time delay and space linear in the number of all concepts (modulo some factor polynomial in the input size). On the other hand, 'dense' contexts, which realise the worst case by bringing about exponential number of concepts, may occur not often in practice. Moreover, various implementation issues, such as dimension of a typical context, specificity of the

Journal of Experimental & Theoretical Artificial Intelligence ISSN 0952–813X print/ISSN 1362–3079 online

DOI: 10.1080/09528130210164170

operating system used, and so on, may be crucial for the practical evaluation of algorithms.

In this article, we consider several algorithms that generate all concepts and/or concept lattice both theoretically and experimentally, for clearly specified data sets. In most cases, it was possible to improve the original versions of the algorithms. We present modifications of some algorithms and indicate conditions when some of them perform better than the others. Only a few known algorithms generating the concept set construct the diagram graphs. We attempted to modify some algorithms so that they can construct diagram graphs.

Algorithms considered in this paper are also employed in the JSM-method of hypothesis generation (Finn 1991). In terms of Formal Concept Analysis (FCA) (Ganter and Wille 1999), the main idea of the JSM-method is as follows: there is a positive context and a negative context relative to a goal attribute. JSM-hypotheses with respect to the goal attribute are intents of the positive context that are not contained in intents of the negative context. Here, we are not going into details of hypothesis generation see (Finn 1991, Ganter and Kuznetsov 2000) for details. It suffices to mention that to generate JSM-hypotyheses, one needs to have an algorithm that generates all concepts.

The paper is organized as follows. In section 2, we give main definitions and an example. In section 3, we discuss the principles of comparing efficiency of algorithms and make an attempt at their classification. In section 4, we give a short review of the algorithms and analyze their worst-case complexity. In section 5, we present the results of experimental comparison.

2. Main definitions

First, we give some standard definitions of Formal Concept Analysis (FCA) (Ganter and Wille 1999).

Definition 1: *A* formal context *is a triple of sets* (G, M, I)*, where G is called a set of objects, M is called a set of attributes, and* $I \subseteq G \times M$.

Definition 2: *For* $A \subseteq G$ *and* $B \subseteq M : A' = \{m \in M | \forall geA(gIm)\}, B' = \{g \in G | \forall m \in B(gIm)\}$.

$X \rightarrow X''$ *is a closure operator, i.e. it is monotonic* $(X \subseteq Y \rightarrow X'' \subseteq Y'')$*, extensive* $(X \subseteq X'')$ *and idempotent* $((X'')'' = X'')$.

Definition 3: *A* formal concept *of a formal context* (G, M, I) *is a pair* (A, B)*, where* $A \subseteq G, B \subseteq M, A' = B$*, and* $B' = A$*. The set A is called the* extent, *and the set B is called the* intent *of the concept* (A, B).

Definition 4: *For a context (G, M, I), a concept* $X = (A, B)$ *is* less general than or equal to *a concept* $Y = (C, D)$ *(or* $X \leq Y$*) if* $A \subseteq C$ *or, equivalently,* $D \subseteq B$.

Definition 5: *For two concepts X and Y such that* $X \leq Y$ *and there is no concept Z with* $Z \neq X, Z \neq Y, X \leq Z \leq Y$*, the concept X is called a* lower neighbour *of Y and Y is called an* upper neighbour *of X. This relationship is denoted by* $X \prec Y$.

Definition 6: *We call the (directed) graph of the relation* $\prec$ *a* diagram graph. *A plane (not necessarily a planar) embedding of a diagram graph where a concept has larger vertical co-ordinate than that of any of its lower neighbours is called a* line *(Hasse)* diagram.

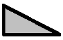
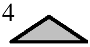
G\M	a = 4 vertices	b = 3 vertices	c = has a right angle	d = all sides are equal
1	X		X	X
2	X		X	
3 		X	X	
4 		X		X

Figure 1. A formal context.

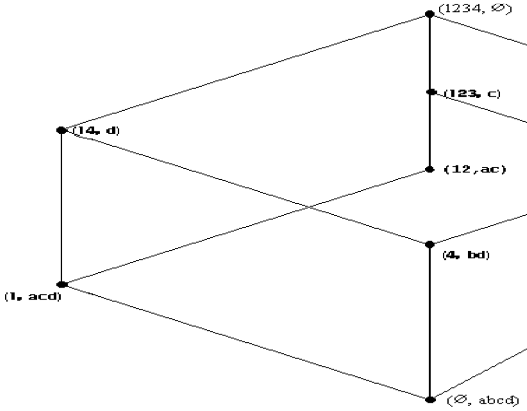


Figure 2. The line diagram for the context from figure 1.

The problem of drawing line diagrams (Ganter and Wille 1999) is not discussed here.

Example 1: Below we present a formal context with some elementary geometric figures and its line diagram. We shall sometimes omit parentheses and write, for example, 12 instead of $\{1,2\}$. We shall also write g' and m' instead of $\{g\}'$ for $g \in G$ and $m \in M$, called object intent and attribute extent, respectively.

3. On principles of comparison

The problem of comparing performance of algorithms for constructing concept lattices and their diagram graphs is a challenging and multifaceted one. The first comparative study of several algorithms constructing the concept set and diagram graphs can be found in Guénoche (1990). However, the formulations of the algorithms are sometimes buggy, and the description of the results of the experimental tests lacks any information about data used for tests. The fact that the choice of the algorithm should be dependent on the input data is not accounted for. Besides, only one of the algorithms considered in Guénoche (1990), namlely that of Bordat (1986), constructs the diagram graph; thus, it is hard to compare its time complexity with that of the other algorithms.

A later review with more algorithms and more information on experimental data can be found in Godin *et al.* (1995). Only algorithms generating diagram graphs are considered. The algorithms that were not originally designed for this purpose are extended by the authors to generate diagram graphs. Unfortunately, such extensions are not always effective: for example, the time complexity of the version of the **NextClosure** algorithm (called **Ganter-Allaoui**) dramatically increases with the growth of the context size. This drawback can be cancelled by the efficient use of binary search in the list produced by the original **NextClosure** algorithm. Tests were conducted only for contexts with small number of attributes per object as compared to the number of all attributes. Our experiments also show that the algorithm proposed in Godin *et al.* (1995) works faster on such contexts than the others do. However, in certain situations not covered in Godin *et al.* (1995), this algorithm is far behind some other algorithms.

In our study, we considered both theoretical (worst-case) and experimental complexity of algorithms. As for the worst-case upper bounds, the algorithms with complexity linear in the number of concepts (modulo a factor polynomial of the input size) are better than those with complexity quadratic in the number of concepts; and the former group can be subdivided into smaller groups with respect to the factor polynomial in input. According to this criterion, the present champion is the algorithm by Nourine and Raynaud (1999). On the other hand, 'dense' contexts, which realize the worst case by bringing about exponential number of concepts, may not occur often in practice.

Starting a comparison of algorithms 'in practice', we face a bunch of problems. First, algorithms, as described by their authors, often allow for different interpretation of crucial details, such as the test of uniqueness of a generated concept. Second, authors seldom describe exactly data structures and their realizations. Third, algorithms behave differently on different databases (contexts). Sometimes authors compare their algorithms with others on specific data sets. We would like to propose that the community should reach a consensus with respect to databases to be used as testbeds. Our idea is to consider two types of test-beds. On the one hand, some 'classical' (well-recognized in data analysis community) databases should be used, with clearly defined scalings if they are many-valued. On the other hand, we propose to use 'randomly generated contexts'. The main parameters of a context $K = (G, M, I)$ seem here to be the (relative to $|M|$) number of objects $|G|$ and the (relative to $|G|$) number of attributes, the (relative, i.e. compared to $|G||M|$) size of the relation I, average number of attributes per object intent (respectively, average number of objects per attribute extent). The community should specify particular type(s) or random context generator(s) that can be tuned by the choice of above (or some other) parameters.

Another major difficulty resides in the choice of a programming language and platform, which strongly affects the performance. A possible way of avoiding this difficulty could be comparing not the time but the number of specified operations (intersections, unions, closures, etc.) from a certain library. However, here one encounters the difficulty of weighting these operations in order to get the overall performance. Much simpler would be comparing algorithms using a single platform.

In this article, we compare performance of several algorithms for clearly specified random data sets (contexts), as well as for real data. As for ambiguities in original pseudo-code formulations of the algorithms, we tried to find their most efficient realizations. Of course, this does not guarantee that more efficient realizations cannot be found.

In most cases, it was possible to improve the original versions of the algorithms. Since only few known algorithms generating the concept set construct also the diagram graph, we attempted to modify some algorithms making them able to construct diagram graphs.

As mentioned above, data structures that realize concept sets and diagram graphs of concept lattices are of great importance. Since their sizes can be exponential w.r.t. the input size, some of their natural representations are not polynomially equivalent, as it is in the case of graphs. For example, the size of the incidence matrix of a diagram graph is quadratic w.r.t. the size of the incidence list of the diagram graph and thus cannot be reduced to the latter in time polynomial w.r.t. the input. Moreover, some important operations, such as finding a concept, are performed for some representations: spanning trees (Bordat 1986, Ganter and Reuter, 1991), ordered lists (Ganter 1984), CbO trees (Kuznetsov 1993), 2–3 trees, see Aho *et al.* (1983) for the definition) in polynomial time, but for some other representations (unordered lists) they can be performed only in exponential time. A representation of a concept lattice can be considered reasonable if its size cannot be exponentially compressed w.r.t. the input and allows the search for a particular concept in time polynomial in the input.

Table 1 presents an attempt at classification of algorithms. Note that this classification refers to our versions of the algorithms rather than to original versions (except for **Titanic** (Stumme *et al.* 2000), which realizes an approach completely different from that of the other algorithms). Here, we do not address techniques for building diagram graphs; the attributes of the context in table 1 describe only construction of the concept set. The algorithm of Carpineto and Romano (1996) and that of Valtchev and Missaoui (2001) do not fit in this classification, since they rely on the structure of the diagram graph to compute concepts.

All the algorithms can be divided into two categories: incremental algorithms (Norris 1978, Dowling 1993, Godin *et al.* 1995, Carpineto and Romano 1996), which, at the *i*th step, produce the concept set or the diagram graph for *i* first objects of the context, and batch ones, which build the concept set and its diagram graph for the whole context from scratch (Chein 1969, Ganter 1984, Bordat 1986, Zabezhailo *et al.* 1987, Kuznetsov 1993, Lindig 1999). Besides, any batch algorithm typically adheres to one of the two strategies: top–down (from the maximal extent to the minimal one) or bottom–up (from the minimal extent to the maximal one). However, it is always possible to reverse the strategy of the algorithm by considering attributes instead of objects and vice versa; therefore, we choose not to include this property into the classification.

Generation of the concept set presents two main problems: (1) how to generate all concepts; (2) how to avoid repetitive generation of the same concept or, at least, to determine whether a concept is generated for the first time. There are several ways to generate a new intent. Some algorithms (in particular, incremental ones) intersect a generated intent with some object intent. Other algorithms compute an intent explicitly intersecting all object intents of the corresponding extent. There are algorithms that, starting from object intents, create new intents by intersecting already obtained intents. Lastly, the algorithm from (Stumme *et al.* 2000) does not use the intersection operation to generate intents. It forms new intents by adding attributes to those already generated and tests some condition on supports of

Table 1. Properties of algorithms constructing concept lattices.

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
Bordat						X	X			
NextClosure		X					X			
Close by One		X							X	
Lindig					X				X	
Chein		X	X					X		
Nourine	X				X				X	
Norris	X	X							X	
Godin	X		X	X					X	
Dowling	X	X							X	
Titanic										X

m1—incremental; m2—uses canonicity based on the lexical order; m3—divides the set of concepts into several parts; m4—uses hash function; m5—maintains an auxiliary tree structure; m6—uses attribute cache; m7—computes intents by subsequently computing intersections of object intents (i.e. $\{g\}' \cap \{h\}'$); m8—computes intersections of already generated intents; m9—computes intersections of nonobject intents and object intents; m10—uses supports of attribute sets.

attribute sets (a support of an attribute set is the number of objects whose intents contain all attributes from this set) to realize whether an attribute set is an intent.

In table 1, attributes m2–m6 correspond to techniques used to avoid repetitive generation of the same concept. This can be done by maintaining specific data structures. For example, the **Nourine** algorithm constructs a tree of concepts and searches in this tree for every newly generated concept. Note that other algorithms (e.g. **Bordat** and **Close by One**) also may use trees for storing concepts, which allows efficient search for a concept when the diagram graph is to be constructed. However, these algorithms use other techniques for identifying the first generation of a concept and, therefore, they do not have the m5 attribute in the context from table 1.

Some algorithms divide the set of all concepts into disjoint sets, which allows narrowing down the search space. For example, the **Chein** algorithm stores concepts in layers, each layer corresponding to some step of the algorithm. The original version of this algorithm looks through the current layer each time a new concept is generated. The version we used for comparison does not involve search to detect duplicate concepts; instead, it employs a canonicity test based on the lexicographical order (similar to that of **NextClosure**), which made it possible to greatly improve the efficiency of the algorithm. We use layers only for generation of concepts: a new intent is produced as the intersection of two intents from the same layer. In our version of the algorithm, layers are much smaller than those in (Chein 1969). The **Godin** algorithm uses a hash function (the cardinality of intents), which makes it possible to distribute concepts among 'buckets' and to reduce the search. Several algorithms (**NextClosure**, **Close by One**) generate concepts in accordance with some version of the lexicographical order of their extents assuming that there is a linear order on the set of objects. At each step of the algorithm, there is a *current object*. The generation of a concept is considered canonical if its extent contains no object preceding the *current object*. Our implementation of the **Bordat** algorithm uses an attribute cache: the uniqueness of a concept is tested by intersecting its intent with the content of the cache.

In many cases, we attempted to improve the efficiency of the original algorithms. Only some of the original versions of the algorithms construct the diagram graph

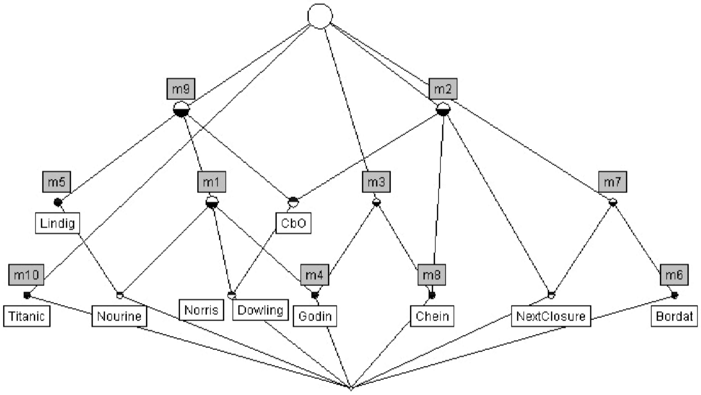


Figure 3. The line diagram of algorithms.

(Bordat 1986, Godin *et al.* 1995, Lindig 1999, Nourine and Raynaud 1999, Valtchev *et al.* 2000); it turned out that the other algorithms could be extended to construct the diagram graph within the same worst-case time complexity bounds.

In the next section, we will discuss worst-case complexity bounds of the considered algorithms. Since the output size can be exponential in the input, it is reasonable to estimate complexity of the algorithms not only in terms of input and output sizes, but also in terms of (cumulative) delay. Recall that an algorithm for listing a family of combinatorial structures is said to have *polynomial delay* (Johnson 1988) if it executes at most polynomially many computation steps before either outputting each next structure or halting. An algorithm is said to have a *cumulative delay d* (Goldberg 1993) if at any point of time in any execution of the algorithm with any input p the total number of computation steps that have been executed is at most $d(p)$ plus $Kd(p)$, where K is the number of structures that have been output so far. If $d(p)$ can be bounded by a polynomial of p, the algorithm is said to have a *polynomial cumulative delay*.

4. Algorithms: a survey

In most cases, top–down algorithms consist of two basic steps:

- Compute the maximal concept (G, G').
- For each computed concept, generate all its lower neighbours.

Top–down algorithms differ in their generation of lower neighbours and preventing concepts from being generated more than once.

The computation of the minimal concept (M', M) is rather straightforward and can be conducted after the termination of the algorithm. In order to avoid unnecessary complications in the discussion of the algorithms, we do not allow for the minimal concept when its extent is empty, and add the minimal element upon construction of the rest of the lattice.

Some top-down algorithms have been proposed in Bordat (1986) and Zabezhailo *et al.* (1987). The algorithm **MI-tree** from Zabezhailo *et al.* (1987) generates the concept set, but does not build the diagram graph. In **MI-tree**, every new concept is searched for in the set of all concepts generated so far.

The algorithm in Bordat (1986) uses a tree (a 'trie', cf. Aho *et al*. (1983)) for fast storing and retrieval of concepts. Our version of this algorithm uses a technique that requires $O(|M|)$ time to realize whether a concept is generated for the first time. We also suggest a procedure *Find* (of complexity $O(|M|^2)$ if the tree is constructed top-down or complexity $O(|G|^2)$ if the tree is constructed bottom-up) that is used to find a concept if it was already generated. It is assumed that there is a linear order on the set G (i.e. we can speak of the first, next and last elements). An auxiliary tree is used to construct the diagram graph; it is implemented by the sets **Ch**: **Ch**((A, B)) is the set of children of the concept (A, B) in this tree. This tree is actually a spanning tree of the diagram graph (i.e. the tree with all diagram graph vertices and some of its edges). The set **Ch** of the currently processed concept consists of its lower neighbours generated for the first time. If the algorithm is used in conjunction with the procedure *LowerNeighbours* presented below, the sets **Ch** have the following form: $\mathbf{Ch}((A,B)) = \{(C,D) \mid (C,D) \prec (A,B) \;\&\; \forall(E,F)((C,D) \prec (E,F) \rightarrow \min(A \backslash E \cup E \backslash A) \in A)\}$. In every set **Ch**, concepts are linearly ordered according to the order in which they were added to the set.

```
Bordat
   0. $L := \emptyset$
   1. Process (($G$, $G'$), $G'$)
   2. $L$ is the concept set.

Process (($A$, $B$), $C$)
   1. $L := L \cup \{(A, B)$
   2. $LN$ := LowerNeighbours (($A$, $B$))
   3. For each ($D$, $E$) $\in LN$
      3.1. If $C \cap E = B$
         3.1.1. $C := C \cup E$
         3.1.2. Process(($D$, $E$), $C$)
         3.1.3. Ch(($A$, $B$)) := Ch(($A$, $B$)) $\cup$ {($D$, $E$)}
      3.2. Else
         3.2.1. Find(($G$, $G'$), ($D$, $E$))
      3.3. ($A$, $B$) is an upper neighbour of ($D$, $E$)

Find ((A, B), (C, D))
   {(A, B) is the concept where the search should be started, and
   (C, D) is the concept to be found}
   1. ($E$, $F$) is the first concept in Ch(($A$, $B$)) such that $F \subseteq D$
   2. If $F \neq D$
      2.1. Find (($E$, $F$), ($C$, $D$))
   3. Else ($E$, $F$) is the desired concept

LowerNeighbours (($A$, $B$))
   1. $LN := \emptyset$
   2. $C := B$
   3. $g$ is the first object in $A$ such that $\neg(\{g\}' \subseteq C)$;
      if there is no such object, $g$ is the last element of $A$
   4. While $g$ is not the last element of $A$
      4.1. $E := \{g\}$
      4.2. $F := \{g\}'$
      4.3. $h := g$
```

```
    4.4. While $h$ is not the last element of $A$
      4.4.1. $h$ is the next element of $A$
      4.4.2. If $\neg(F \cap \{h\}' \subseteq C)$
        4.4.2.1. $E := E \cup \{h\}$
        4.4.2.2. $F := F \cap \{h\}'$
    4.5. If $F \cap C = B$
      4.5.1. $LN := LN \cup \{(E, F)\}$
    4.6. $C := C \cup F$
    4.7. $g$ is the first object in $A$ such that
         $\neg(\{g\}' \subseteq C)$; if there is no such object,
         $g$ is the last element of $A$
5. LN is the set of lower neighbours of $(A, B)$
```

This is just one possible method for generating lower neighbours; a similar version of the algorithm for constructing minimal intersections was proposed by O.M. Anshakov (Zabezhailo *et al.* 1987). The proof of the correctness of this algorithm, as well as of another algorithm designed by Khazanovskii (Zabezhailo *et al.* 1987), can be found in Obiedkov (1999).

If the diagram graph is to be constructed, the spanning tree of the diagram graph given by sets **Ch** should be extended by some more edges. To restore these edges, one should be able to perform the search for a concept in the spanning tree. We show that the search for a concept can be performed in $O(|M|^2)$ time $(O(|M|)$ tests for attribute subset containment). Indeed, suppose that we need to find a concept with intent D. First, we examine concepts from the set $\mathbf{Ch}(G, G')$ in order to find some concept (E, F) such that $F \subseteq D$. It is assumed that the concepts from **Ch** are considered in the order in which they were put into this set. When we find (E, F), such that $F \subseteq D$, we check whether $F = D$; if not, we continue searching in the set **Ch** corresponding to (E, F), and so on.

Statement 1: *If* (S, T) *is a predecessor of* (E, F) *in the tree,* (X, Y) *is a left sibling of* (S, T) *in the tree, and* (V, W) *is the parent of* (S, T) *and* (X, Y) *in the tree, then* $F \cap Y = W$.

Proof: Let $F \cap Y = W_1$. Since $W \subseteq F$ and $W \subseteq Y$, we have $W \subseteq W_1$. As $W_1 \subseteq Y$, three cases are possible: (1) $W \subset W_1 \subset Y$, (2) $W_1 = Y$, and (3) $W_1 = W$. If (1) holds, then there is a concept with the intent W_1 descending from (V, W), and (X, Y) cannot be a child of (V, W). If (2) holds, then (E, F) is less general than (X, Y). But for an arbitrary concept U, all less general concepts are generated and added to the tree prior to the examination of right siblings of U; thus, (E, F) is added to the tree before (S, T) and (S, T) cannot be a predecessor of (E, F) in the tree. Therefore, $W = W_1$.

The *trajectory* of an intent D or $\mathrm{Tr}(D) = \{D_1, \ldots, D_n\}$ with $D_\mathrm{n} = D$, is the set of all intents for which *Find* tests the relation $D_i \subseteq D$ during the search for D. The *trajectory length* is the cardinality of the trajectory. Obviously, $\mathrm{Tr}(D_i) \subseteq \mathrm{Tr}(D_j)$ if the relation $D_i \subseteq D$ was tested before $D_j \subseteq D$. Let $H(D_j) = \bigcup_{i \in Tr(D_j)} D_i$.

Statement 2: *For an arbitrary intent* $D \subseteq M$, *the trajectory length of* D *is not greater than* $|M|$.

Proof: By the definition of $\mathrm{Tr}(D)$, for any $D_j, D_k \in \mathrm{Tr}(D)$ such that $j < k$, either $H(D_j) = H(D_k)$ or $H(D_j) \subset H(D_k)$. We show that $H(D_j) \subset H(D_k)$. Three cases are possible: (1) D_j is a predecessor of D_k in the tree, (2) D_j is a left sibling of D_k in the tree, and (3) D_j is a left sibling of a predecessor of D_k in the tree. For the first two cases, the proof is trivial. Consider the third case. If $H(D_j) = H(D_k)$, $\forall a \in D_k \exists i < k : a \in D_i$. Hence, $a \in D_k \cap D_i$, and, by Statement 1, all elements of D_k belong to P, the parent of D_k in the tree, and $D_k = P$. This contradicts to the fact that each intent corresponds biuniquely to a vertex of the tree.

Thus, $H(D_j) \subset H(D_k)$ and the sequence $H(D_i), i = \{1, \ldots, k\}$ is strictly increasing. On the other hand, $H(D_k) \subseteq M$ for any k. Therefore, $k \leq |M|$.

The second parameter C in *Process* is the set of all attributes contained in the intent of the concept (A, B) being *Process*ed and the intents of all its left siblings and all left siblings of all its predecessors (w. r. t. the tree). Thus, if, for some concept (D, E) that is a lower neighbour of $(A, B), C \cap E \neq B(B \subset C \cap E)$, then ($D$, E) was generated earlier. Indeed, this means that there are concepts (V, W), (X, Y), and (S, T) such that (X, Y) is a left sibling of (S, T); either $(S, T) = (D, E)$ or (S, T) is a predecessor of (D, E); (V, W) is the parent of (X, Y) and (S, T); and $E \cap Y \neq W(W \subset E \cap Y)$. It is impossible that $W \subset E \cap Y \subset Y$ (see the proof of Statement 1). So, $E \cap Y = Y$, which ensures that (D, E) has already been generated as a descendant of (X, Y).

The time complexity of the algorithm that finds the lower neighbours of a concept is $O(|G| \times |M|^2)$. Any concept can be *Process*ed only once; therefore, the time complexity of **Bordat** is $O(|G||M|^2|L|)$, where $|L|$ is the size of the concept lattice. Moreover, this algorithm has a polynomial delay. In fact, at each vertex of the tree it either produces a new concept making at most $|M|$ closures, or backtracks at most min($|M|$, $|G|$) times and either halts at the root of the tree or obtains a new concept before attaining the root. Thus, at most O(|G||M|^2) computation steps are performed before Bordat generates a next concept or halts.

If there is no need to build the diagram graph, everything below the line 3.1.2 in *Process* could be omitted.

Let us use the context from Example 1 to illustrate how the algorithm works. We do not go into details of generation of lower neighbours and their search. The first generation of every concept is given in boldface. Subscripts of literals C are used to distinguish between different recursion levels.

- **(1234, $\varnothing$) the maximal concept**
- $C_0 := \varnothing$
- **(123, c)**, **(14, d)**, **(34, b)** are the lower neighbours of the concept $(1234, \varnothing)$ (generated by the *LowerNeighbours* procedure in this very order)
- $C_0 \cap \mathrm{c} = \varnothing \rightarrow C_0 := \mathrm{c}$, *Process*((123, c), c)
  - $C_1 :=$ c
  - **(12, ac)**, **(3, bc)** are the lower neighbours of (123, c)
  - $C_1 \cap \mathrm{ac} = \mathrm{c} \rightarrow C_1 :=$ ac, *Process* ((12, ac), ac)
    - $C_{11} :=$ ac
    - **(1, acd)** is the lower neighbour of (12, ac)
    - $C_{11} \cap \mathrm{acd} = \mathrm{ac} \rightarrow C_{11} :=$ acd, **Ch**((12, ac)) : = {(1, acd)}
  - **Ch**((123, c)) : = {(12, ac)}
  - $C_1 \cap \mathrm{bc} = \mathrm{c} \rightarrow$ **Ch**((123, c)) := {(12, ac), (3, bc)}
- **Ch**((1234, $\varnothing$) := {(123, c)}
- $C_0 \cap \mathrm{d} = \varnothing \rightarrow C_0 :=$ cd, *Process*((14, d), cd)
  - $C_2 :=$ cd
  - (1, acd), **(4, bd)** are the lower neighbours of (14, d)

  - $C_2 \cap \mathrm{acd} = \mathrm{cd} \neq \mathrm{d} \rightarrow \mathit{Find}(1, \mathrm{acd})$
  - $C_2 \cap \mathrm{bd} = \mathrm{d} \rightarrow C_2 := \mathrm{bcd}, \mathbf{Ch}((14, \mathrm{d})) := \{(4, \mathrm{bd})\}$
- $\mathbf{Ch}((1234, \varnothing)) := \{(123, \mathrm{c}), (14, \mathrm{d})\}$
- $C_0 \cap \mathrm{b} = \varnothing \rightarrow C_0 := \mathrm{bcd}, \mathit{Process}((34, \mathrm{b}), \mathrm{bcd})$
  - $C_3 := \mathrm{bcd}$
  - (3, bc), (4, bd) are the lower neighbours of (34, b)
  - $C_3 \cap \mathrm{bc} = \mathrm{bc} \neq \mathrm{b} \rightarrow \mathit{Find}(3, \mathrm{bc})$
  - $C_3 \cap \mathrm{bd} = \mathrm{bd} \neq \mathrm{b} \rightarrow \mathit{Find}(4, \mathrm{bd})$
- $\mathbf{Ch}((1234, \varnothing) := (123, \mathrm{c}), (14, \mathrm{d}), (34, \mathrm{b})$

Next, we consider the family of algorithms based on computation of closures for subsets of G. They follow the below scheme:

```
1. While some condition is true do
   1.1. For some set $A \subseteq G$, compute $(A'', A')$.
   1.2. If this is the first time the concept $(A'', A')$ is generated (or, in some
        algorithms, if this is definitely the last time it is generated), add it to the
        concept set.
```

Individual algorithms differ in the condition used to exit the loop, the method for selecting subsets to compute the closures, the technique for checking whether a concept was generated earlier (the *canonicity test:* for any concept, there is a unique canonical generation, which is specified in each particular algorithm), and the method for computing closures.

The simplest (naïve) algorithm corresponding to this scheme computes closures of all subsets of G but the empty one. It performs the canonicity test by searching through all concepts generated so far. The loop runs 2^n times, where $n = |G|$. Thus, the number of iterations is equal to or greater than the number of concepts. At each step of the loop, the generated concept is tested for canonicity, which requires the time linear in the number of concepts. Therefore, the algorithm has the time complexity quadratic in the number of concepts.

The algorithm **NextClosure** proposed by Ganter computes closures for only some of subsets of G and uses an efficient canonicity test, which does not address the list of generated concepts and requires little storage space.

The following technique for selecting subsets is used. It is assumed that there is a linear order ( ) on G. The algorithm starts by examining the set consisting of the object maxim≺al with respect to (max(G)), and finishes when the canonically generated closure is equal to G. L≺et A be a currently examined subset of G. The generation of A'' is considered canonical if $A''\backslash A$ contains no g $\max(A)$. If the generation of A'' is canonical (and A'' is not equal to G), the next se≺t to be examined is obtained from A'' as follows:

$$A'' \cup \{g\} \backslash \{h \mid h \in A'' \& g \quad h\}, \text{ where } g = \max(\{h \mid h \in G \backslash A''\}). \qquad (1)$$

Otherwise, the set examined at the n≺ext step is obtained from A in a similar way, but the added object must be less (w.r.t.) than the maximal object in A:

$$A \cup \{g\} \backslash \{h \mid h \in A \& g \quad h\}, \text{ whe≺re } g = \max(\{h \mid h \in G \backslash A \& h \quad \max(A)\}) \qquad (2)$$

The algorithm runs as bel≺ow: ≺

```
NextClosure
```

```
1. $L := \emptyset$, $A := \emptyset$, $g := \max(G)$
2. While $A \neq G$
   2.1. $A := A \cup \{g\} \setminus \{h \mid h \in A \,\&\, g \quad h\}$
   2.2. If $\{h \mid h \in A'' \setminus A \,\&\, h \quad g\} = \emptyset$
      2.2.1. $L := L \cup \{(A'', A')\}$
      2.2.2. $g := \max(\{h \mid h \in G \setminus A''\})$
      2.2.3. $A := A''$
   2.3. Else
      2.3.1. $g := \max(\{h \mid h \in G \setminus A \,\&\, h \quad g\})$
3. $L$ is the concept set
```

The NextClosure algorithm produces the set of all concepts in time $O(|G|^2|M||L|)$ and has polynomial delay $O(|G|^2|M|)$.

The **Close by One** (**CbO**) algorithm uses a similar notion of canonicity and a similar method for selecting subsets (Kuznetsov 1993). However, it employs an intermediate structure that helps to compute closures more efficiently using the generated concepts. The **NextClosure** algorithm computes A'' by subsequently intersecting object intents and checking which objects from $G\backslash$A contain the resulting intent. The **CbO** algorithm obtains each new closure from a concept it generated at a previous step by intersecting its intent with intent of an object that does not belong to its extent. The original version of the **CbO** algorithm uses a tree as an intermediate structure. The tree can be used for the construction of the diagram graph or as an alternative to the diagram graph for storing the concept set allowing one to easily update this set when new objects are to be taken into account. If we regard this structure only as an auxiliary one, a simple stack is sufficient, as any branch of the tree becomes useless after a new branch is started.

First, let us show how a stack can be incorporated into the NextClosure algorithm. If a concept is generated canonically, it is placed into the stack. Having decided on the next set, the algorithm searches the stack for a concept that could be used to compute the closure. Suppose that A is the last set that was examined, B is the set to be examined next, and g is the object added to A or A'' to obtain B. If $B = \{g\}$, we clear the stack. Otherwise, we wish to efficiently compute B' without subsequently intersecting the intents of objects from B. With this in mind, we search the stack for a concept (C, D) such that $B' = D \cap \{g\}'$. If $|B| > 1$, there is such a concept in the stack; its extent $C = (B \backslash \{g\})''$; since $g = \max(B)$, the closure of $B \backslash \{g\}$ must have been computed by the moment when B is considered. The stack consists of pairs $\langle (V, W), h \rangle$, where ($V$, W) is a concept and h is the object used to generate this concept. To find (C, D), we should find the first element of the stack with $h \quad g$ (removing all the preceding elements of the stack).

The last (bottom) element of the stack always has the form $\langle (h'', h'), h \rangle$, where $h \in G$. The set chosen at some iteration of the **NextClosure** algorithm either consists of only one element or is a superset of the last one-element set $\{h\}$ that was examined and does not contain objects less (w.r.t.) than h. Obviously, the stack will never become empty when looking for a concept, and the algorithm will succeed.

If $k = |A|$, it is necessary to perform $\cap$-operation $k - 1$ times (provided that A' is computed by intersection of $\{g\}'$ for all $g \in A$) if the stack is not used. If the stack is used, only one application of $\cap$ is needed, but some additional time is required to test and remove (not more than $k - 1$ and usually much less) elements from the stack. One of the advantages of the original **NextClosure** algorithm is that it needs little storage space, as it does not use generated concepts to generate other ones. The version with the stack uses more memory, however the number of concepts in the stack never exceeds $|G|$.

The algorithm with a tree can be obtained from the algorithm with a stack as follows.

1. The algorithm starts generating the tree from a dummy root with an empty extent; this root is declared to be the current node;
2. When a new node is added to the tree, it is joined with the current node by an edge and declared the current node;
3. Instead of clearing the stack, the root is declared to be the current node;
4. All actions with the head of the stack are changed for similar actions with the current node;
5. Instead of removing the head of the stack, the parent of the current node is declared to be the current node.

Note that there are two possible strategies of using the stack: (1) first, choose g and the set B to be considered next, then search the stack for a concept that could be used to generate the new intent (this strategy is described above); or (2) search the stack for a concept that could be used to generate the new intent; when found, choose the object g and the set to be considered next. The **CbO** algorithm implements the second strategy, which uses the stack more intensively. For this algorithm, unlike for the **NextClosure** algorithm, the set whose closure is computed at some step of **CbO** cannot be described by the formulas (1) and (2): it equals the extent of a generated concept supplemented by one object. Here, we present the simplest recursive version of **CbO**. The algorithm computes concepts according to the lexicographic order defined on the subsets of G (concepts whose extents are lexicographically less are generated first): A is lexicographically less than B if ($A \subset B$) and $\min(A) > \max(A)$ or ($A \not\subset B$) and $\min((A \cup B)\backslash(B \cap A)) \in A$. The **NextClosure** algorithm generates concepts in a different lexicographic order: A is lexicographically less than B if $\min(A \cup B)\backslash(A \cap B)) \in B$. The order in which concepts are generated by **CbO** seems to be preferable if the diagram graph is constructed: the first generation of the concept is always canonical, which makes it possible to find a concept in the tree when it is generated for the second time and to draw appropriate diagram graph edges. **NextClosure**-style lexicographical order allows binary search, which is helpful when the diagram graph is to be constructed after the construction of the set of all concepts.

```
Close by One
   1. $L := \varnothing$
   2.  For each $g \in G$
     2.1. Process $(\{g\}, g, (g'', g'))$
   3. $L$ is the concept set.

Process $(A, g, (C, D))$
   $\{C = A'', D = A'\}$
   1. If $\{h | h \in C\backslash A \& h \quad g\} = \varnothing$
     1.1. $L := h \cup \{(C, D)\}$
     1.2. For each $f \prec\in \{h | h \in G \& g \quad h\}$
       1.2.1. $Z := C \cup \{f\}$
       1.2.2. $Y := D \cap \{f\}'$          $\prec$
```

```
        1.2.3. $X := Y'(= Z \cup \{h \mid h \in G \backslash Z \& Y \subseteq \{h\}'\})$
        1.2.4. Process $(Z, f, (X, Y))$
```

To construct the diagram graph with the **CbO** algorithm, we can use a tree. Unlike the tree from **Bordat**, this tree is not a part of the diagram graph. Assume that child nodes of any tree node are linearly ordered with respect to the lexicographic order. A *path* in the tree is a sequence of its nodes starting from the root such that any node is followed by its child node or right sibling (*a* is the right sibling of *b* if *c* is the parent of *a* and *b*, and $b < a$ with respect to the order on the child nodes of *c*). The length of such a path is limited by $|G| + 1$. The diagram graph can be constructed as follows. Each time some concept is generated non-canonically, the algorithm searches the tree for the canonical generation of this concept (which requires time linear in the number of objects) and the corresponding edge is added. While searching, we make the following path in the tree: the last node of the path is the desired concept; each node of the path is either the first child of the previous node (if the previous node is less general than the desired concept) or its first right sibling (otherwise). The time complexity of **Close by One (CbO)** is $O(|G|^2|M||L|)$, and its polynomial delay is $O(|G|^3|M|)$.

Another bottom-up algorithm is presented in Lindig (1999). The idea is to generate the bottom concept and then, for each concept that is generated for the first time, generate all its upper neighbours. Lindig uses a tree of concepts that allows one to check whether some concept was generated earlier. The description of the tree is not detailed in Lindig (1999), but it seems to be the spanning tree of the inverted diagram graph (i.e. with the root at the bottom of the diagram graph), similar to the tree from **Bordat**. Finding a concept in such a tree takes $O(|G| \times |M|)$ time. In fact, the below algorithm may be regarded as a bottom-up version of the **Bordat** algorithm.

The procedures *Find* and *Next* are not described here for the reasons of space. *Find* returns if it cannot find the specified concept. *Next* returns λ if there is no next concept.

```
Lindig
   1. $C := (M', M)$
   2. $T$ is a concept tree consisting of the root node $C$
   3. While $C \neq \lambda$
      3.1. For each $X \in$ UpperNeighbours $(C)$
         3.1.1. $X :=$ Find$((M', M), X, T)$
         3.1.2. If $X = \lambda$
            3.1.2.1. Add $X$ to $T$ as a child of $C$
         3.1.3. $X$ is an upper neighbour of $C$
      3.2. $C :=$ Next$(C, T)$
   4. The set of nodes of $T$ is the concept set
```

```
UpperNeighbours $((A, B))$
   0. $UN := \varnothing$
   1. $Min := G \backslash A$
   2. For each $g \in G \backslash A$
      2.1. $D := (A \cup \{g\})'$
      2.2. $C := D'$
      2.3. If $Min \cap ((C \backslash A) \backslash \{g\}) = \varnothing$
         2.3.1. $UN := UN \cup \{(C, D)\}$
```

```
    2.4. Else
      2.4.1. $Min := Min \backslash \{g\}$
  3. $Un$ is the set of upper neighbours of $(A, B)$
```

The time complexity of the algorithm is $O(|G|^2|M||L|)$. Its polynomial delay is $O(|G|^2|M|)$.

The **NextClosure** algorithm generates the intent of each new concept by intersecting intents of all objects from its extent. The **CbO** algorithm does the same by intersecting an object intent and intent of a generated concept. This strategy enables one to incrementally process new data by updating and extending the results without performing all the computations from scratch. The **Norris** algorithm given below is basically an incremental version of **CbO**. However, there is another strategy: represent the objects by extent–intent pairs and generate each new concept intent as the intersection of intents of two existent concepts. The **AI-tree** (Zabezhailo *et al.* 1987) and **Chein** (Chein 1969) algorithms, which are very similar to each other, use this strategy.

```
Chein
  1. $L_1 := ((\{g\}, \{g\}') \mid g \in G\}$
  2. $k := 1$
  3. While $|L_k| > 1$
    3.1. $L_{k+1} := \emptyset$
    3.2. For each $\{(A, B), (C, D)\} \subseteq L_k$
      3.2.1. $F := B \cap D$
      3.2.2. If there is $E \subseteq G$ such that $(E, F) \in L_{k+1}$
        3.2.2.1. $E := E \cup A \cup C$
      3.2.3. Else
        3.2.3.1. $L_{k+1} \cup \{(A \cup C, F)\}$
      3.2.4. If $F = B$
        3.2.4.1. Mark $(A, B)$ in $L_k$
      3.2.5. If $F = D$
        3.2.5.1. Mark $(C, D)$ in $L_k$
    3.3. $L_k := L_k \backslash \{(V, W) \mid (V, W)$ is marked in $L_k\}$
    3.4. $k := k + 1$
  4. $\bigcup_k L_k$ is the concept set
```

The algorithm admits slight modifications. In the original versions, concepts with empty intents are ignored; therefore, steps 3.2.2–3.2.5 are performed only when F is not empty. Assume that concepts in L_k are linearly ordered and the choice of the subset $\{(A, B), (C, D)\}$ at Step 3.2 is based on the following order (from the least to the largest): $K \subseteq L_k$ is less than $K_1 \subseteq L_k$ if $\min((K \cup K_1) \backslash (K \cap K_1)) \in K$, where $\min(K)$ is the least element of K with respect to the linear order on L_k). Then it is not necessary to add elements of C to E at Step 3.2.2.1 (provided that (A, B) is less than (C, D) with respect to the linear order on the elements of L_k); the algorithm will still work correctly. Besides, instead of marking concepts at Steps 3.2.4.1 and 3.2.5.1, it is possible to remove them from L_k altogether, thus, excluding them from the following consideration. However, it is not possible to remove both concepts at once (in the case where the context is given by the set of object intents $\{a, a, ab\}$ this would lead to an error). Let us illustrate this algorithm with the context from Example 1 (the output of the algorithm consists of the concepts given in boldface in the lines not marked with *).

L_1:

1.1. **(1, acd)**
1.2. **(2, ac)** *
1.3. **(3, bc)**
1.4. **(4, bd)**

2. L_2:
   2.1. (1, acd) +
      2.1.1. + (2, ac): **(12, ac)**—mark 1.2
      2.1.2. + (3, bc): **(13, c)** *
      2.1.3. + (4, bd): **(14, d)**
   2.2. (3, bc) +
      2.2.1. + (4, bd): **(34. b)**
3. L_3:
   3.1. (12, ac) +
      3.1.1. + (13, c): **(123, c)**—mark 2.1.2
      3.1.2. + (14, d): (124, $\varnothing$) → **(1234, $\varnothing$)**—see 3.1.3 *
      3.1.3. + (34, b): ac ∩ b = $\varnothing$—the same as in 3.1.2
   3.2. (14, d) +
      3.2.1. + (34, b): d ∩ b = $\varnothing$ the same as in 3.1.2
4. L_4:
   4.1. (123, c) +
      4.1.1. + (1234, $\varnothing$): **(1234, $\varnothing$)**—mark 3.1.2

It is rather obvious that the algorithm can be improved by considering only pairs of those concepts from L_k that do not have a common ancestor in $L_{k\text{-}1}$. Concepts with the extents {1, 2} and {2, 3} yield the same new intent as concepts with the extents {1, 2} and {1, 3}, since the unions of extents within each pair are identical. The same holds for a pair of concepts with the extents {1, 2}, {3, 4} and a pair of concepts with the extents {1, 2, 3}, {1, 2, 4}. As we consider the pairs ({1, 2}, {1, 3}) and ({1, 2, 3}, {1, 2, 4}), it is not worth to consider also the pairs ({1, 2}, {2, 3}) and ({1, 2}, {3, 4}). Nevertheless, the search for a concept (Step 3.2.2) must be carried out through the whole L_k set. Such an approach gives the following layers L_3 and L_4:

`Chein-1`

3. L_3:
   3.1. (12, ac) +
      3.1.1. + (13, c): **(123, c)** mark 2.1.2
      3.1.2. + (14, d): (124, $\varnothing$)
4. L_4:
   4.1. (123, c) +
      4.1.1. + (1234, $\varnothing$): **(1234**, $\varnothing$) mark 3.1.2

It is possible to extend the **Chein** algorithm with a canonicity test similar to that of the **CbO** algorithm. Such a test is much faster than the search in L_k, the size of which is usually considerably larger than that of G. For experiments, we used the version with the canonicity test. The time complexity of the modified algorithm is $O(|G|^3|M||L|)$. The algorithm has polynomial delay $O(|G|^3|M|)$.

Due to their incremental nature, the algorithms considered below do not have polynomial delay. Nevertheless, they all have cumulative polynomial delay.

Nourine and Raynaud (1999) propose an algorithm for the construction of the lattice using a lexicographic tree. This algorithm is the best known worst-case complexity bound $O((|G| + |M|)|G||L|)$. Edges of the tree are labelled with attributes, and nodes are labeled with concepts whose intents consist of the attributes that label the edges leading from the root to the node. Clearly, some nodes do not have labels. First, the tree is constructed incrementally (similar to the **Norris** algorithm presented below). An intent of a new concept C is created by intersecting an object intent g' with the intent of a concept D created earlier, and the extent of C is formed by adding g to the extent of D, which takes $O(|M| + |G|)$ time. A new concept is searched for in the tree using the intent of the concept as the key; this search requires $O(|M|)$ time. When the tree is created, it is used to construct the diagram graph. For each concept C, its parents are sought for as follows. Counters are kept for every concept initialized to zero at the beginning of the process. For each object, the intersection of its intent and the concept intent is produced in $O(|M|)$ time. A concept D with the intent equal to this intersection is found in the tree in $O(|M|)$ time and the value in the counter of D increases; if the counter is equal to the difference between the cardinalities of the extents of the concepts C and D (i.e. the intersection of the intent of C and the intent of any object from D outside C is equal to the intent of D), the concept D is a parent of C.

The algorithm proposed by Norris (1978) is essentially an incremental version of the **CbO** algorithm. The **CbO**-tree (which is useful only for diagram graph construction) can be built as follows: first, there is only the dummy root; examine objects from G and for each concept of the tree check whether the object under consideration has all the attributes of the concept intent; if it does, add it to the extent; otherwise, form a new node and declare it a child node of the current one; the extent of the corresponding concept equals the extent of the parent node plus the object being examined; the intent is the intersection of this object intent and the parent intent; next, test the new node for the canonicity; if the test fails, remove it from the tree. The original version of the algorithm from Norris (1978) does not construct the diagram graph, and no data structure is explicitly mentioned.

If it is not necessary to construct the diagram graph, the **Norris** algorithm is preferable to **CbO**, as the latter has to remember how the last concept was generated (in the above version, it is implemented by the recursion; other variants include stack and tree); this involves additional computational resources. The **Norris** algorithm does not maintain any structure but the concept set. Besides, the closure of an object set is never computed explicitly.

```
Norris
1. $L := \varnothing$
2. For each $g \in G$
   2.1. Add ($g$, $L$)
3. $L$ is the concept set

Add ($g$, $L$)
   1. For each $(A, B) \in L$
      1.1 If $B \subseteq \{g\}'$
         1.1.1. $A := A \cup \{g\}$
      1.2. Else
```

```
            1.2.1. $D := B \cap \{g\}'$
            1.2.2. If $\{h \mid h \in G \backslash A \& h$ has already been
                   $Added$ & $D \subseteq \{h\}'\} = \varnothing$
               1.2.2.1. $L := L \cup \{(A \cup \{g\}, D)\}$
    2. If $\{h \mid h \in G \& \ h$ has already been $Added$ & $\{g\}' \subseteq \{h\}'\} = \varnothing$
       2.1. $L := L \cup (\{g\}, \{g\}')$
```

The time complexity of the algorithm is $O(|G|^2|M||L|)$.

The algorithm proposed by Godin *et al.* (1995) has the worst-case time complexity quadratic in the number of concepts. This algorithm uses a heuristic based on the size of attribute sets. The algorithm always computes the bottom concept, which is called *inf*.

```
Godin
    1. $inf := (\varnothing, \varnothing), L := \varnothing$
    2. For each $g \in G$
       2.1. Add $(g, L)$
    3. $L$ is the concept set

Add $(g, L)$
    1. If $inf = (\varnothing, \varnothing)$
       1.1. $inf := (\{g\}, \{g\}')$
       1.2. $L := \{inf\}$
    2. Else
       2.1. If $\neg(\{g\}' \subseteq$ the intent of $inf)$
          2.1.1. If the extent of $inf = \varnothing$
             2.1.1.1. The intent of $inf$ := the intent of $inf \cup \{g\}'$
          2.1.2. Else
             2.1.2.1. $L := L \cup (\varnothing,$ the intent of $inf \cup \{g\}')$
             2.1.2.2. $(\varnothing,$ the intent of $inf \cup \{g\}')$ is an upper
                      neighbour of $inf$
             2.1.2.3. $inf := (\varnothing,$ the intent of $inf \cup \{g\}')$
       2.2. From $i := 0$
            to max$(\{j \mid \exists(A, B)((A, B) \in L \& |B| = j)\})$
          2.2.1. $C_i := \{(A, B) \mid (A, B) \in L \& |B| = i\}$
          2.2.2. $C_i' := \varnothing$
       2.3. From $i := 0$
            to max$(\{j \mid \exists(A, B)((A, B) \in L \& |B| = j)\})$
          2.3.1. For each $(A, B) \in C_i$
             2.3.1.1. If $B \subseteq \{g\}'$
                2.3.1.1.1. $A : A \cup \{g\}$
                2.3.1.1.2. $C_i' := C_i' \cup \{(A, B)\}$
                2.3.1.1.3. If $B = \{g\}'$
                   2.3.1.1.3.1. Exit the algorithm
             2.3.1.2. Else
                2.3.1.2.1. $Int := B \cap \{g\}'$
                2.3.1.2.2. If $\neg\exists(A1, B1)((A1, B1) \in C'_{|Int|} \& B1 = Int)$
                   2.3.1.2.2.1. $L := L \cup \{(A \cup \{g\}, Int)\}$
                   2.3.1.2.2.2. $C'_{|Int/} := C'_{|Int/} \cup \{(A \cup \{g\}, Int)\}$
```

```
2.3.1.2.2.3. $(A \cup \{g\}, Int)$ is an upper neighbour of
             $(Y, y)$
2.3.1.2.2.4. UpdateEdges $((A \cup \{g\}, Int), (A, B))$
2.3.1.2.2.5. If $Int = \{g\}'$
  2.3.1.2.2.5.1. Exit the algorithm
```

The *UpdateEdges* procedure proposed by Godin is quite time-consuming (at least theoretically). Valtchev and Missaoui (2001) suggest another strategy for updating edges, improving theoretical complexity and, apparently, practical performance of the algorithm. However, this change does not improve the performance of the algorithm when the diagram graph is not constructed. In their implementation of **Godin,** Valtchev and Missaoui use a trie for concept lookup, which allows achieving better performance (or at least, better worst-case bounds).

Only the lines 2.1.2.2, 2.3.1.2.2.3, and 2.3.1.2.2.4 of the above algorithm deal with the diagram graph construction. Below, we demonstrate how the algorithm constructs the concept set (without the diagram graph) for the context from Example 1. Each generated concept is shown together with the object (g_i) and the concept (except for the cases where the new concept is generated at Step 2.1.2 as a specification of *inf*) that were used to create it.

- g_1: *inf* := **(1, acd)** c_1
- g_2, c_1: **(12, ac)** c_2
- g_3: *inf* := **(∅, abcd)** c_3
- g_3, c_2: **(123, c)** c_4
- g_3, c_1: (12, c)—in C'_1, there is a concept (c_4) with the intent $\{c\}$
- g_3, c_3: **(3, bc)** c_5
- g_4, c_4: **(1234, ∅)** c_6
- g_4, c_2: (124, ∅)—in C'_0, there is a concept (c_6) with the intent ∅
- g_4, c_5: **(34, b)** c_7
- g_4, c_1: **(14, d)** c_8
- g_4, c_3: **(4, bd)** c_9

To check whether the new concept (with the intent *Int*) has already been generated, $C'_{|int|}$ is searched for a concept with the intent *Int* (Step 2.3.1.2.2). This search is efficient when the context is fairly sparse. Otherwise, a canonicity test similar to that of the above algorithms is preferable (i.e. check whether *Int* is contained in some object from $G \backslash A$ already processed). If it is, the concept with this intent must have been generated (since more general concepts are processed first). If the diagram graph is not constructed, such test ensures in the worst case the time complexity of the algorithm linear in the number of concepts. For the 12 × 12-context of the form $(A, A, \neq)$ (the corresponding binary matrix is filled with ones except for the diagonal), the number of necessary relation test ($\subseteq$ and $=$ for attribute sets) decreases from $\approx$ 485 000 to $\approx$ 30 000. For the context 20 × 20 with from 10 to 20 attributes per object, the canonicity test needs about half less operations than the original **Godin** test (however, it becomes necessary to test whether an object belongs to a set). In fact, this modification yields something very similar to the **Norris** algorithm: the only difference is the order in which concepts are treated; in the case of the **Godin** algorithm, this order makes it possible to avoid some operations resulting in non-canonical concepts by exiting the *Add* procedure. On sparse contexts, the version with the canonicity test performs slower than the original **Godin** algorithm.

However, time difference between various algorithms processing sparse contexts is generally not significant. Even if some algorithm performs several times slower than another one, the difference is several seconds or even fractions of a second. The algorithms with time complexity linear in the number of concepts perform considerably faster than other algorithms on the contexts that need hours or, at least, dozens of minutes to be processed. The **Godin** algorithm uses a cardinality heuristic that may reduce search in some cases. The idea generalizes to the introduction of an efficiently computable hash-function f defined on the set of concepts such that any function value has a relatively small number of pre-images in the given context (and distributing concepts among groups (buckets) so that the concepts A and B fall within the same group if and only if $f(A) = f(B)$). Moreover, when an object is being added, the groups are formed only from concepts containing this object in their extents, which allows one to check promptly whether a concept has been already generated by searching for it in the appropriate group. If the group contains fewer concepts than there are objects processed so far, such test is preferable to the canonicity test. The hash function used by Godin, i.e. the cardinality of concept intents, does not always satisfy this condition. A possible workaround is to choose one of the two tests at run-time: if there are more concepts in the group to be searched than objects processed so far, the canonicity test is performed; otherwise, the group is searched for the concept. However, computing cardinality may be expensive. If M is large, the average size of a group might be significantly less than the number of objects (in the case of a sparse context), which encourages one to use a cardinality heuristic. However, if attribute sets are implemented so that several attributes are encoded by a single number, which is useful from the storage space perspective and allows efficient operations of intersection and union, then computation of the size of a concept intent takes much more time. As mentioned above, the worst-case complexity of **Godin** is quadratic in the number of generated concepts. This algorithm has cumulative polynomial delay, since in the worst case, for each generated concept, it looks through the list of the same order of magnitude as the list of already generated concepts, to find the concept with the same intent.

It is worth noting that the sets C_i may be based on concept extents, as well as intents. The sets C_i' cannot be treated in this way, as, at Step 2.3.1.2.2, a concept with certain intent is searched for; the extent of this concept is unknown.

Dowling (1993) proposed an incremental algorithm for computing knowledge spaces. A dual formulation of the algorithm (with set-theoretical union replaced by intersection) allows generation of the concept set. Despite the fact that the theoretical worst-case complexity of the algorithm is $O(|M||G|^2|L|)$, the constant in this upper bound seems to be too large and in practice the algorithm performs worse than other algorithms.

Recently, Valtchev *et al.* (2000) proposed an algorithm (called **Divide and Conquer**) generalizing the incremental approach towards lattice creation. This algorithm divides the context into two parts, either horizontally (i.e. splitting G) or vertically (i.e. splitting M), constructs the diagram graph for each part, and assembles them into the global lattice. Thus, this algorithm is particularly suitable for parallel computations. For details, we refer the reader to Valtchev *et al.* (2000). The theoretical complexity of this algorithm is difficult to estimate in terms of input and output sizes. The complexity of the procedure assembling lattices L_1 and L_2 into the global lattice L is $O((|G| + |M|)(|L_1||L_1| + |L||M|)$.

5. Results of experimental tests

The algorithms were implemented in C + +. The tests were run on a Celeron®–1000 computer, 384 MB RAM under Windows 2000®. Here, we present a number of charts that show how the execution time of the algorithms depends on various parameters. More charts can be found in Kuznetsov and Obiedkov (2000).

For tests, we used randomly generated data. Contexts were generated based on three parameters: $|G|$, $|M|$, and the number of attributes per object (denoted below as $|g'|$; all objects of the same context had equal numbers of attributes). Given $|g'|$, every row of the context (i.e. every object intent) was generated by successively calling the *rand* function from the standard C library to obtain the numbers of attributes constituting the object intent, which lead to uniform distribution.

We implemented the following algorithms:

- Those that build only the set of concepts:
  - **Chein** (with modifications described above and the **CbO**-style canonicity test, which performs better than the original **Chein** algorithm);
  - **Close by One**;
  - **NextClosure;**
  - **Norris** (with buckets from the **Godin** algorithm that allow avoiding some useless operations);
  - **Bordat** (actually, the improved version described above);
  - Two versions of the **Godin** algorithm: **GodinEx**, where the sets C are based on sizes of extents, and **Godin**, where they are based on sizes of intents; their results are almost identical;
  - **Dowling** (a dual version; see above);
- Those that build the diagram graph:
  - **Close by One**;
  - **NextClosure** (which uses binary search to find the canonical generation of a concept);

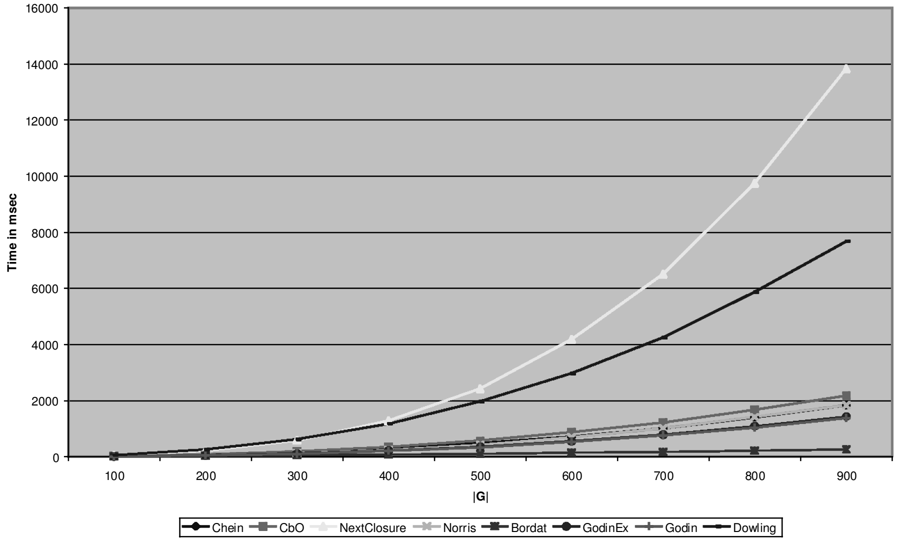


Figure 4. Concept set: $|M| = 100$; $|g'| = 4$.

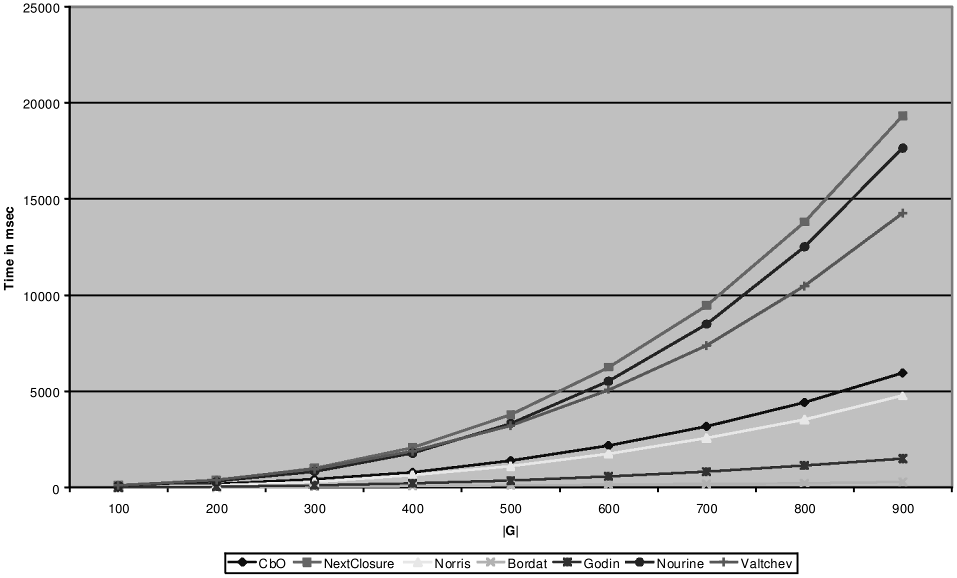


Figure 5. Diagram graph: $|M| = 100$; $|g'| = 4$.

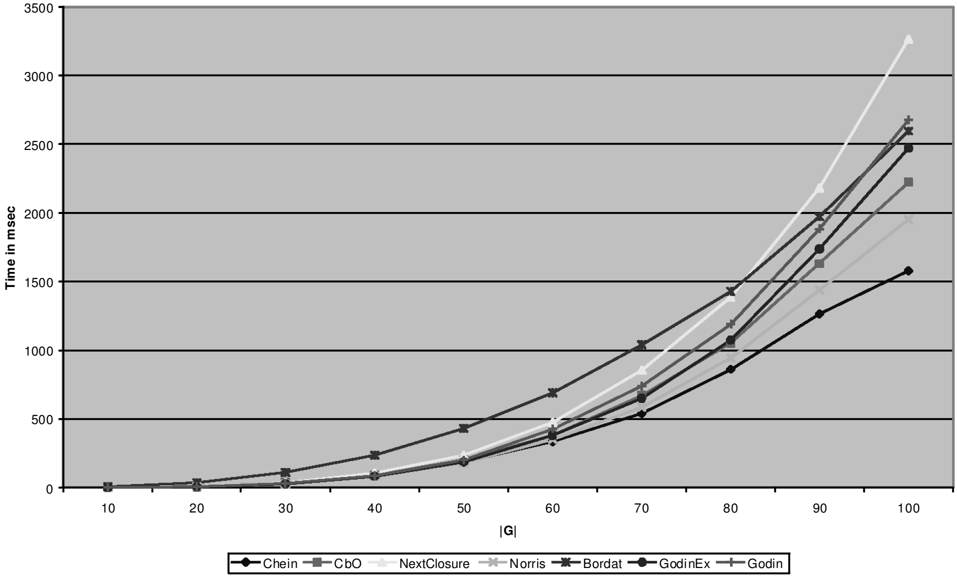


Figure 6. Concept set: $|M| = 100$; $|g'| = 25$.

- **Norris** (which uses a tree);
- **Bordat** (again, our version);
- **Godin** (where sets C are based on the concept extents);
- **Lindig**;
- **Nourine;**
- **Valtchev** (using horizontal splitting of the object set)

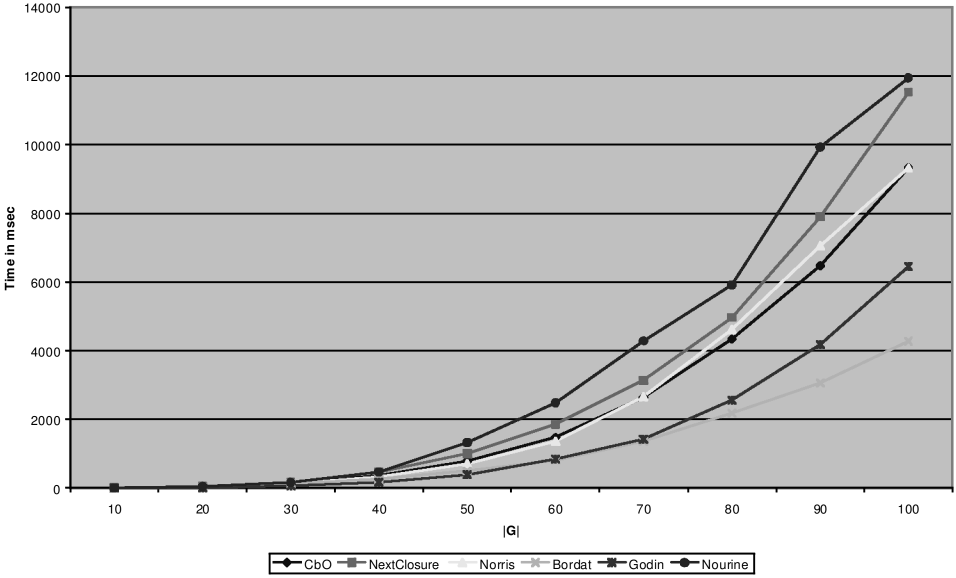


Figure 7. Diagram graph: $|M| = 100$; $|g'| = 25$.

To improve the readability of charts, we sometimes omit algorithms that perform much slower than the others.

The **Godin** algorithm (and **GodinEx**) is a good choice in the case of sparse contexts. However, when contexts become denser, its performance decreases dramatically. The **Bordat** algorithm seems most suitable for large contexts, especially if it is necessary to build the diagram graph. When $|G|$ is small, the **Bordat** algorithm runs several times slower than other algorithms, but, as $|G|$ grows, the difference between **Bordat** and other algorithms becomes smaller, and, in many cases, **Bordat** finally turns out to be the leader. For large and dense contexts, the fastest algorithms are bottom-up canonicity-based algorithms (**Norris**, **CbO**, **NextClosure**).

It should be noted that the **Nourine** algorithm featuring the smallest time complexity, has not been the fastest algorithm: even when diagonal contexts of the form $(G, G, \neq)$ (which corresponds to the worst case) are processed, its performance was inferior to the **Norris** algorithm. Probably, this can be accounted to the fact that we represent attribute sets by bit strings, which allows very efficient implementation of set-theoretical operations (32 attributes per one processor cycle); whereas searching in the Nourine-style lexicographic tree, one still should individually consider each attribute labelling edges.

Figures 8–9 show the execution time for the contexts of the form $(G, G, \neq)$, which yield $2^{|G|}$ concepts. It is noteworthy that the algorithm of Valtchev is almost the best in the worst case of $(G, G, \neq)$; being, sometimes, inferior to **Norris**. However, in other situations, even in case of other dense contexts, it is far from being the fastest one. Valtchev *et al.* (2000) suggest that this algorithm works better on database-like contexts (obtained by nominal scaling). This must hold for the version using vertical splitting (of the attribute set), but not for the version using horizontal splitting (the one we implemented), since, even in database-like contexts, there are no special relationships between objects (similar to the relationship between attributes obtained by scaling from the same multi-valued attribute). Nevertheless, it can be possible to

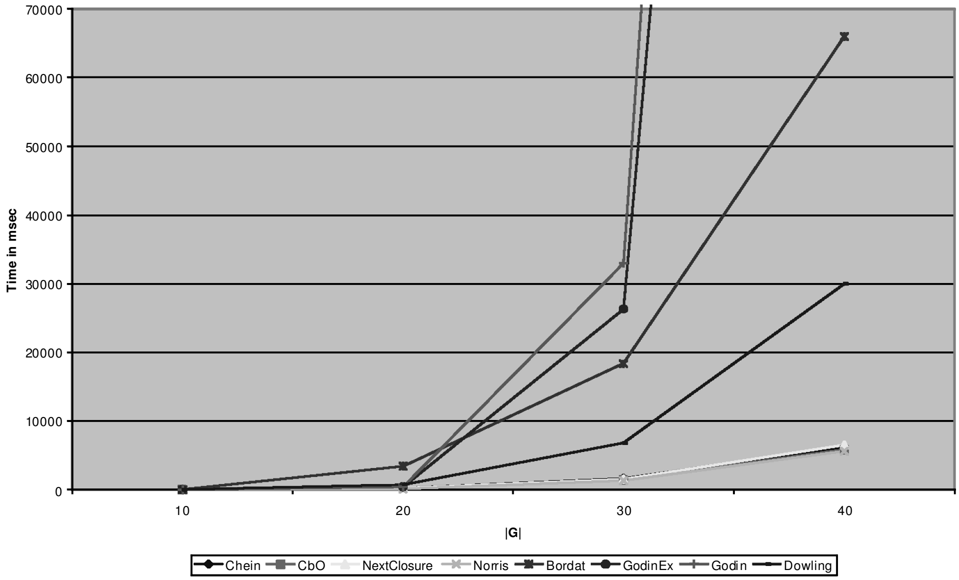


Figure 8. Concept set: $|M| = 100$; $|g'| = 50$.

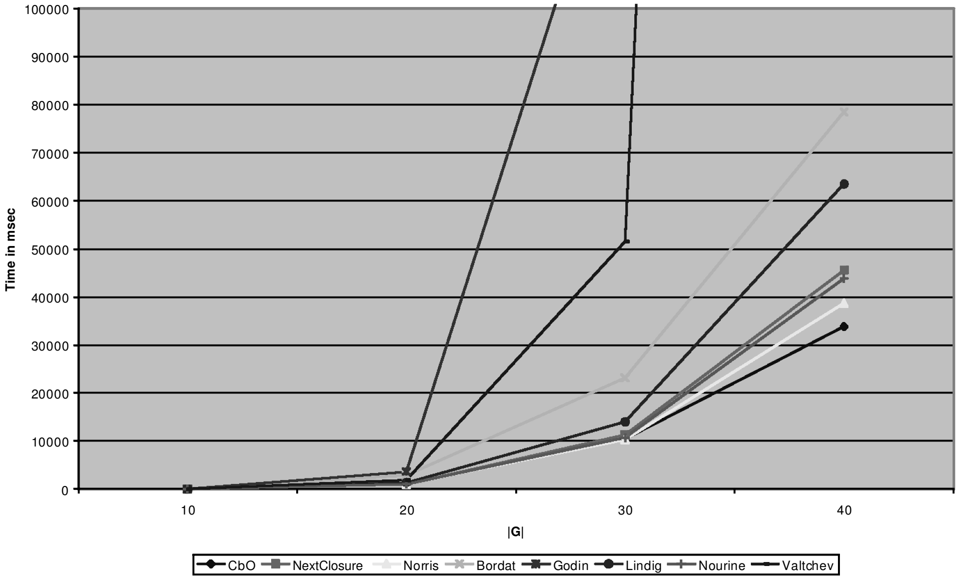


Figure 9. Diagram graph: $|M| = 100; |g'| = 50$.

introduce modifications such as sophisticated preliminary sorting that would make this algorithm a fair choice in case of large databases with large number of implications on attributes.

We also performed comparisons on the SPECT heart database (267 objects, 23 attributes, 21549 concepts, 110589 edges in the diagram graph) available from the UCI Machine Learning Repository (Blake and Merz 1998). The results are given in figures 10–11.

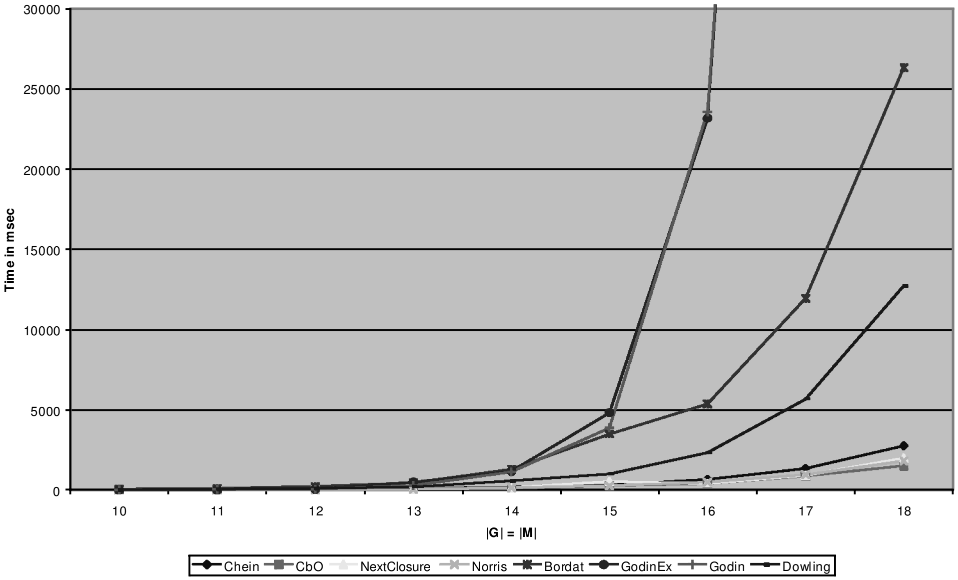


Figure 10. Concept set: contexts of theform $(G, G, \neq)$.

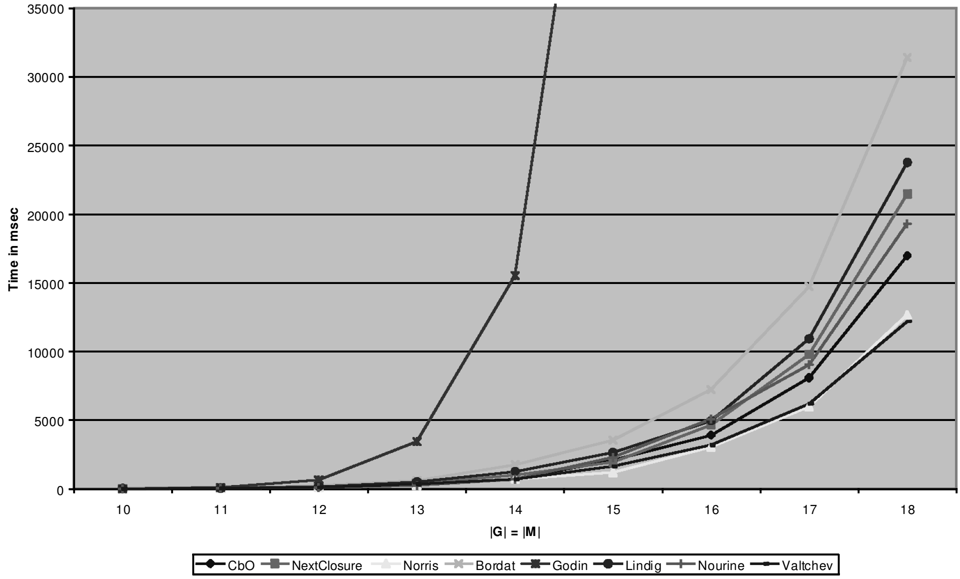


Figure 11. Diagram graph: contexts of the form $(G, G, \neq)$.

6. Conclusions

In this work, we attempted to compare, both theoretically and experimentally, performance (time complexity) of some well-known algorithms for constructing concept lattices. We discussed principles of experimental comparison of complexity, making evaluation of other useful properties of algorithms a subject of further research.

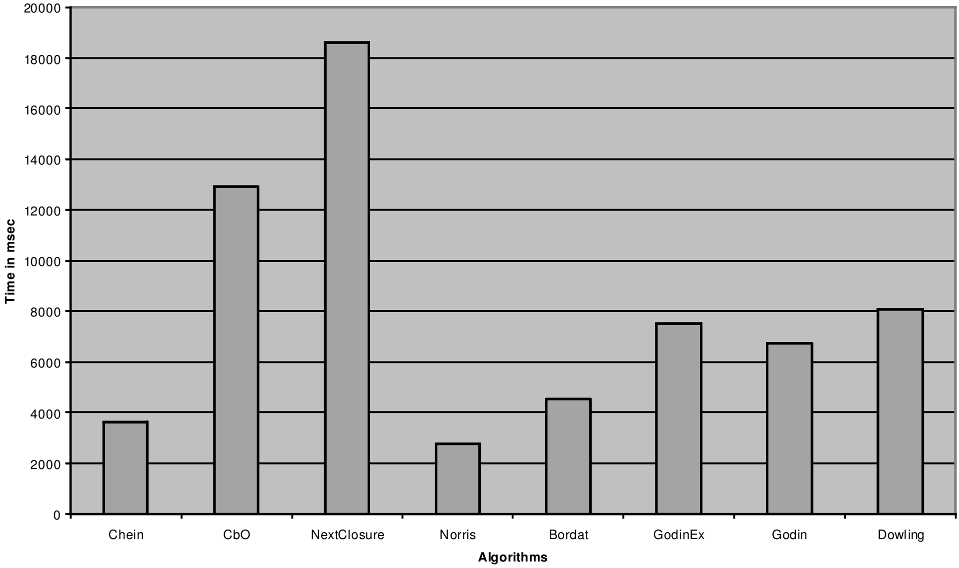


Figure 12. Concept set: SPECT database.

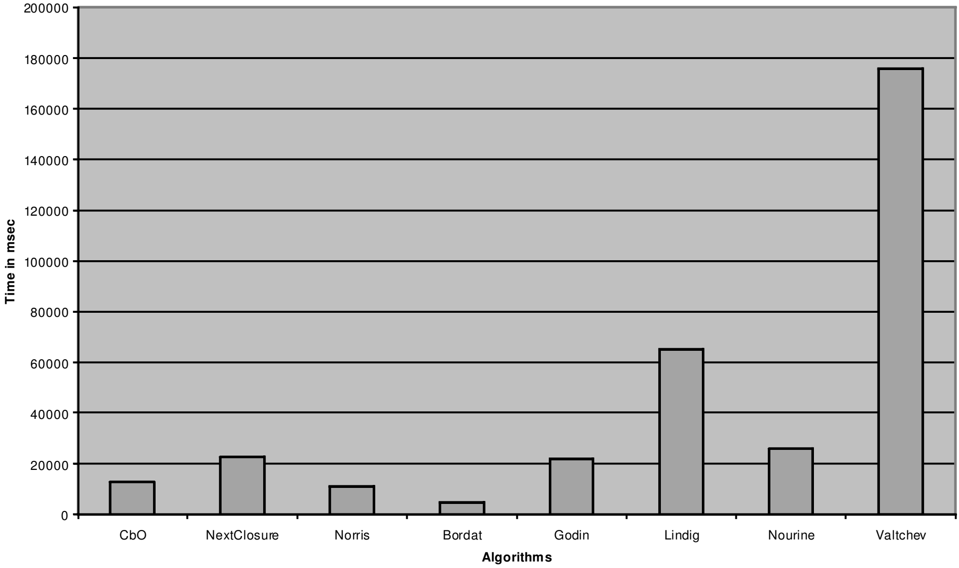


Figure 13. Diagram graph: SECT database.

A new algorithm (Stumme *et al.* 2000) was proposed quite recently. Its worst time complexity is not better than that of the algorithms described above, but the authors report on its good practical performance for databases with very large number of objects (such as MUSHROOMS). Comparing the performance of this algorithm with those considered above and testing the algorithms on large databases, including 'classical' ones, will be the subject of the further work. We can also mention works (Carpineto and Romano 1996, Mephu Nguifo and Njiwoua 1998) where similar algorithms were applied for machine learning and data analysis, e.g. in Mephu Nguifo and Njiwoua (1998) a Bordat-type algorithm was used. The incremental algorithm proposed by Van Der Merwe and Kourie (2001) uses special 'intent operations' (approximate and exact intent representatives) to determine the only canonical generator of a concept before generating it. The algorithm in Yevtushenko (2002) using so-called binary decision diagrams is reported to work fast for very dense contexts. The algorithms of V. Kaibler for computing the lattice of polyhedron faces is similar to **CbO**-algorithm.

The choice of an algorithm for construction of the concept lattice should be based on the properties of input data. Recommendations based on our experiments are as follows: the **Godin** algorithm should be used for small and sparse contexts; for dense contexts, the algorithms based on the canonicity test, linear in the number of input objects, such as **Norris, Close by One,** and **NextClosure** should be used. **Bordat** performs well on contexts of average density, especially, when the diagram graph is to be constructed. Experiments with real data suggest that, when only concepts are needed, the simple and intuitive algorithm of Norris is the best choice. When the diagram graph should be constructed, it is better to use our modification of **Bordat.** Of course, these recommendations should not be considered as the final judgement; more experiments with various types of real data are to be done. By this work, we would like rather to provoke further interest in well-substantiated comparison of algorithms that generate concept lattices.

Acknowledgments

This work was supported by the 'Computer Intelligent System' Program of the Russian Academy of Sciences, project no. 2.2. The first author was also supported by the Alexander von Humboldt Foundation. We thank Vincent Duquenne and Bernhard Ganter for helpful discussions and Sergei Yevtushenko for his diagram-drawing program Concept Explorer.

References

Aho, A. V., Hopcroft, J. E., and Ullmann, J. D., 1983, *Data Structures and Algorithms* (Reading: Addison-Wesley).

Blake, C. L., and Merz, C. J., 1998, UCI Repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html], Irvine, CA: University of California, Department of Information and Computer Science.

Bordat, J. P., 1986, Calcul pratique du treillis de Galois d'une correspondance. *Math. Sci. Hum.*, **96**: 31–47.

Carpineto, C., and Romano, G., 1996, A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, **24**: 95–122.

Chein, M., 1969, Algorithme de recherche des sous-matrices premières d'une matrice. *Bull. Math. Soc. Sci. Math. R.S. Roumanie*, **13**: 21–25.

Dowling, C. E., 1993, On the irredundant generation of knowledge spaces. *J. Math. Psych.*, **37** (1): 49–62.

Finn, V. K., 1991, Plausible reasoning in systems of JSM type. *Itogi Nauki i Tekhniki, Ser. Informatika*, **15**: pp. 54–101.

Ganter, B., 1984, Two Basic Algorithms in Concept Analysis, FB4-Preprint No. 831, TH Darmstadt.

Ganter B., and Kuznetsov, S., 2000, Formalizing hypotheses with concepts. In *Proceedings 8th International Conference on Conceptual Structures, ICCS 2000, Lecture Notes in Artificial Intelligence*, **1867**, Darmstadt, Germany, pp. 342–356.

Ganter, B., and Reuter, K., 1991, Finding all closed sets: a general approach. *Order*, **8**: 283–290.

Ganter, B., and Wille, R., 1999, *Formal Concept Analysis: Mathematical Foundations* (Heidelberg: Springer).

Godin, R., Missaoui, R., and Alaoui, H., 1995, Incremental concept formation algorithms based on Galois lattices. *Computation Intelligence*, **11** (2): 246–267.

Goldberg, L. A., 1993, *Efficient Algorithms for Listing Combinatorial Structures* (Cambridge: Cambridge University Press).

Guénoche, A., 1990, Construction du treillis de Galois d'une relation binaire. *Math. Inf. Sci. Hum.*, **109**: 41–53.

Johnson, D. S., Yannakakis, M., and Papadimitriou, C. H., 1988, On generating all maximal independent sets. *Inf. Proc. Let.*, **27**: 119-123.

Kuznetsov, S. O., 1989, Interpretation on graphs and complexity characteristics of a search for specific patterns. *Automatic Documentation and Mathematical Linguistics*, **24** (1): 37–45.

Kuznetsov, S. O., 1993, A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic Documentation and Mathematical Linguistics*, **27** (5): 11–21.

Kuznetsov, S. O., 2001, On computing the size of a lattice and related decision problems. *Order*, **18** (4): 13–21.

Kuznetsov, S. O., and Obiedkov S. A., 2000, Algorithm for the Construction of the Set of All Concepts and Their Line Diagram, Preprint MATH-Al-05, TU-Dresden.

Lindig, C., 1999, Algorithmen zur begriffsanalyse und ihre anwendung bei softwarebibliotheken, (Dr.-Ing.) Dissertation, Techn. Univ. Braunschweig.

Mephu Nguifo, E., and Njiwoua, P., 1998, Using lattice-based framework as a tool for feature extraction, in Feature Extraction. In H. Liu and H. Motoda (eds) *Construction and Selection: A Data Mining Perspective* (Boston, MA: Kluwer), pp. 205–216.

Norris, E. M., 1978, An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de Mathématiques Pures et Appliquées*, **23** (2): 243–250.

Nourine L., and Raynaud O., 1999 A fast algorithm for building lattices. *Information Processing Letters*, 71: 199–204.

Obiedkov, S. A., 1999, Algorithmic problems of the JSM-method of automatic hypothesis generation. *Nauch. Tekh. Inf., Ser. 2*, 1–2: 64–75.

Stumme G., Taouil R., Bastide Y., Pasquier N., and Lakhal L., 2000, Fast computation of concept lattices using data mining techniques. In *Proceedings, 7th Int. Workshop on Knowledge Representation Meets Databases (KRDB* 2000), Berlin, Germany, pp. 129–139.

Stumme G., Wille R., and Wille U., 1998, Conceptual knowledge discovery in databases using formal concept analysis methods. In *Proceedings, 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98)*, Nantes, France, pp. 450–458.

Valtchev P., and Missaoui, R., 2001, Building concept (Galois) lattices from parts: generalizing the incremental methods. In *Proceedings, 9th International Conference on Conceptual Structures, ICCS 2001, Lecture Notes in Artificial Intelligence*, **2120**, Stanford, CA, USA, pp. 290–303.

Valtchev, P., Missaoui, R., and Lebrun, P., 2000, A Partition-Based Approach Towards Building Galois (Concept) Lattices. Rapport de recherche no. 2000-08, Département d'Informatique, UQAM, Montréal, Canada.

Van Der Merwe, F.J., and Kourie, D.G., 2002, AddAtom: an incremental algorithm for constructing concept lattices and concept sublattices. Technical report, Department of Computer Science, University of Pretoria.

Yevtushenko S., 2002, BDD-based Algorithms for the Construction of the Set of All Concepts. In *Contributions to ICCS 2002*, Borovets, Bulgaria, pp. 61–73.

Zabezhailo, M. I., Ivashko, V. G., Kuznetsov, S. O., Mikheenkova, M. A., Khazanovskii, K. P., and Anshakov, O. M., 1987, Algorithms and programs of the JSM-method of automatic hypothesis generation. *Automatic Documentation and Mathematical Linguistics*, **21**(5): 1–14.