A New Algorithm for Computing Formal Concepts

Vilem Vychodil*
Dept. Systems Science and Industrial Engineering,
Binghamton University—SUNY
Binghamton, NY 13902, U. S. A.
vychodil@binghamton.edu

Abstract

This paper introduces new and efficient algorithm for computing formal concepts. Formal concepts are particular conceptual clusters hidden in object-attribute data tables. The main distinguishing features of our algorithm compared to the algorithms known to date are its speed and memory efficiency. We describe the algorithm, representation of the input data, implementation of the algorithm, its computational efficiency, and provide an initial theoretical insight. The efficiency of algorithm implementation is demonstrated by several examples.

1 Introduction

In this paper we present a new algorithm for computing formal concepts. Formal concepts are conceptual clusters hidden in object-attribute data tables. We are dealing with data tables which take form of a table with rows corresponding to objects, columns corresponding to attributes, and table entries being ×s and blanks indicating whether objects have/do not have the corresponding attributes. Tables of this form represent a fundamental form of data which is the subject of study of several data mining disciplines. One of them is formal concept analysis [Carpineto and Romano, 2004; Ganter and Wille, 1999] invented in the early 80s by Rudolf Wille [Wille, 1982]. Formal concept analysis produces two basic outputs out of the object-attribute data table: (1) concept lattice—a hierarchical structure of conceptual clusters hidden in the data and (2) attribute implications—particular if-then dependencies among attributes. In this paper we focus on computing the conceptual clusters.

Since the inception of formal concept analysis, there have been proposed several algorithms for computing formal concepts, see [Kuznetsov and Obiedkov, 2002] for an overview and comparison of the algorithms. Among the best known algorithms are Ganter's algorithm [Ganter, 1984] and Lindig's algorithm [Lindig, 2000]. Almost all algorithms proposed to date are sufficient for data tables of small sizes. However, with growing data tables, the performance of algorithms is not satisfactory. Needless to say, fast algorithms for formal concept analysis are a necessary prerequisite for our ability to apply the formal concept analysis in a wider area of problems where large collections of data appear.

We present a light-weight algorithm which is fast and memory efficient and can be implemented so that it uses just static linear data structures. Moreover, it can utilize low-level operations present in CPUs and FPUs of contemporary computers, which significantly improves the performance of the algorithm. This enables us to have efficient implementations of the algorithm utilizing well the power of the computer hardware. In this paper, we describe the algorithm and compare its performance with the other algorithms.

This paper is organized as follows. Section 2 presents preliminaries from formal concept analysis, Section 3 describes data structures we use during the computation and introduces the algorithm for computing closures, Section 4 shows the main algorithm for computing all formal concepts. Finally, in Section 5 we comment on the implementation of the algorithm and its performance.

2 Formal Concept Analysis: Overview

Formal concept analysis deals with binary data tables describing relationship between objects and attributes, respectively. The input for FCA is a data table with rows corresponding to objects, columns corresponding to attributes (or features), and table entries being ×'s and blanks, indicating whether an object given by row has or does not have an attribute given by column. An example of such data table is depicted in Fig. 1. In this table, we have objects $a, \ldots, e$, attributes $s, \ldots, z$. According to the table, object a does not have attribute s; a has t and u; b does not have w, etc. More formally, denote the set of all objects by X and the set of all attributes Y, respectively. The data table from Fig. 1 can be seen as a binary relation $I \subseteq X \times Y$ such that $\langle x, y \rangle \in I$ iff object x has attribute y. That is,

*The author is also with Dept. Computer Science, Palacky University, Czech Republic; Support of grant #1ET101370417 of GA AV ČR, by grant #201/05/0079 of the Czech Science Foundation, and by institutional support, research plan MSM 6198959214, is gratefully acknowpedged.

	s	t	u	v	w	x	y	z
a		×	×		×	×		×
b	×			×			×	×
c	×	×	×	×		×	×	
d		×			×	×		
e	×	×			×	×	×	×

Figure 1: Object-attribute data table

$I = \{\langle a,t\rangle, \langle a,u\rangle, \langle a,w\rangle, \ldots, \langle e,y\rangle, \langle e,z\rangle\}$. In FCA, I is usually called a *formal context* [Ganter and Wille, 1999]. Each formal context $I \subseteq X \times Y$ induces a couple of operators $^{\uparrow}$ and $^{\downarrow}$ defined, for each $A \subseteq X$ and $B \subseteq Y$, as follows:

$$A^{\uparrow} = \{y \in Y \mid \text{for each } x \in A\colon \langle x,y\rangle \in I\}, \quad (1)$$
$$B^{\downarrow} = \{x \in X \mid \text{for each } y \in B\colon \langle x,y\rangle \in I\}. \quad (2)$$

Operators $^{\uparrow}\colon 2^X \to 2^Y$ and $^{\downarrow}\colon 2^Y \to 2^X$ defined by (1) and (2) form so-called Galois connection [Ganter and Wille, 1999]. By definition (1), $A^{\uparrow}$ is a set of all attributes shared by all objects from A and, by (2), $B^{\downarrow}$ is a set of all objects sharing all attributes from B. Following our sample data table, if we put $A = \{a,c\}$ then $A^{\uparrow}$ is the set of attributes common to both a and c, i.e. $A^{\uparrow} = \{t,u,x\}$. In a similar way, for $B = \{s,y\}$, we have $B^{\downarrow} = \{b,c,e\}$.

A pair $\langle A,B\rangle$ where $A \subseteq X$, $B \subseteq Y$, $A^{\uparrow} = B$, and $B^{\downarrow} = A$, is called a *formal concept* (*in* $I \subseteq X \times Y$). Formal concepts can be seen as particular clusters hidden in the data. Namely, if $\langle A,B\rangle$ is a formal concept, A (called an *extent of* $\langle A,B\rangle$) is the set all objects sharing all attributes from B and, conversely, B (called an *intent of* $\langle A,B\rangle$) is the set of all attributes shared by all objects from A. Note that this approach to "concepts" as entities given by their extent and intent goes back to classical Port Royal logic. From the technical point of view, formal concepts are fixed points of the Galois connection $\langle^{\uparrow},^{\downarrow}\rangle$ induced by the formal context.

The set $\mathcal{B}(X,Y,I)$ of all formal concepts in I equipped with the subconcept-superconcept ordering $\leq$ given, for any $\langle A_1,B_1\rangle, \langle A_2,B_2\rangle \in \mathcal{B}(X,Y,I)$, by

$$\langle A_1,B_1\rangle \leq \langle A_2,B_2\rangle \text{ iff } A_1 \subseteq A_2 \text{ (or, iff } B_2 \subseteq B_1),$$

is called a *concept lattice*. The main theorem of FCA [Ganter and Wille, 1999] shows that concept lattice is indeed a complete lattice and gives additional criteria describing its structure.

Another way of looking at formal concept is that they represent maximal rectangles in the data table which are full of ×'s, see [Ganter and Wille, 1999] for details.

3 Computing the Closures

In this section we clarify representation of the input data and introduce data structures that will be used during the computation. Then we describe first part of the algorithm. Namely, the part which computes intents generated by sets of attributes.

3.1 Representation of the Input Data

The input for our analysis is a formal context. For simplicity, let $X = \{0,1,\ldots,m\}$ and $Y = \{0,1,\ldots,n\}$ be our sets of objects and attributes, respectively. That is, we denote the objects and attributes by nonnegative integers. There is no danger of confusing objects with attributes because we do not mix elements from the sets X and Y in any way.

We are going to represent each formal context $\langle X,Y,I\rangle$ two ways. First, by a two-dimensional array, denoted *context*, which corresponds with I. The array *context* contains 1s and 0s so that

$$context[i,j] = \begin{cases} 1, & \text{if } \langle i,j\rangle \in I, \\ 0, & \text{otherwise.} \end{cases}$$

Considering the formal context from Fig. 1, the two-dimensional array corresponding to the context will be the following:

	0	1	2	3	4	5	6	7
0	0	1	1	0	1	1	0	1
1	1	0	0	1	0	0	1	1
2	1	1	1	1	0	1	1	0
3	0	1	0	0	1	1	0	0
4	1	1	0	0	1	1	1	1

The second representation of the data, which will be used together with the first one, is an array of lists of objects. For each attribute $y \in Y$, we let $rows[y]$ be a list of all objects having the attribute y. Thus, $rows[y]$ contains $x \in X$ iff $\langle x,y\rangle \in I$. In other words, $rows[y]$ is a list of all the objects from $\{y\}^{\downarrow}$. In case of our example, for $y = 0,\ldots,7$, we get:

$$rows[0] = (1,2,4), \qquad rows[4] = (0,3,4),$$
$$rows[1] = (0,2,3,4), \qquad rows[5] = (0,2,3,4),$$
$$rows[2] = (0,2), \qquad rows[6] = (1,2,4),$$
$$rows[3] = (1,2), \qquad rows[7] = (0,1,4).$$

Altogether, *rows* can be seen as an array of lists. The two-dimensional array *context* and the array of lists *rows* will be used by the subsequent algorithm.

The algorithms we are going to describe will use sets of attributes represented by their characteristic arrays. That is, a subset $B \subseteq Y = \{0,1,\ldots,n\}$ will be represented by an $(n+1)$-element linear array b of 1s and 0s such $b[k] = 1$ iff $k \in B$ (and $b[k] = 0$ iff $k \notin B$). By a slight abuse of notation, we will identify B with b and write $B[k] = 1$ to denote $k \in B$, etc.

3.2 Algorithm for Computing Closures

Our algorithm for computing all formal concepts of a formal context $I \subseteq X \times Y$ is based on several well-known observations. First, in order to compute all the formal concepts, it is sufficient to compute all the intents because each formal concept is uniquely determined by its intent. Namely, if B is intent, the corresponding formal concept is $\langle B^{\downarrow},B\rangle$.

Second, intents can be characterized by their closure properties. It is a well-known fact that B is an intent of a concept iff $B = B^{\downarrow\uparrow}$, i.e. iff B is a fixed point of the closure operator $^{\downarrow\uparrow}\colon 2^Y \to 2^Y$, see [Ganter and

Procedure `compute_closure`(B, y)

```
1  for j from 0 upto n do
2  |  set D[j] to 1;
3  end
4  foreach i in rows[y] do
5  |  set match to true;
6  |  for j from 0 upto n do
7  |  |  if B[j] = 1 and context[i, j] = 0 then
8  |  |  |  set match to false;
9  |  |  |  break for loop
10 |  |  end
11 |  end
12 |  if match = true then
13 |  |  for j from 0 upto n do
14 |  |  |  if context[i, j] = 0 then
15 |  |  |  |  set D[j] to 0;
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 return D
```

Wille, 1999]. Therefore, the set of all intents we wish to compute is

$$\{B \subseteq Y \mid B = B^{\downarrow\uparrow}\} = \{B^{\downarrow\uparrow} \mid B \subseteq Y\}.$$

It is therefore important to have a quick procedure to compute closures of sets of attributes. In a more detail, for each set $B \subseteq Y$ of attributes, we wish to compute $B^{\downarrow\uparrow}$ where $^{\downarrow\uparrow}$ is a consecutive application of the operators $^{\downarrow}$ and $^{\uparrow}$ induced from I by (1) and (2), respectively.

Another important observation is that if B is an intent then due to the monotony of $^{\downarrow\uparrow}$, all the intents which are strictly greater than B can be expressed as $(B \cup C)^{\downarrow\uparrow}$ where C is a set of attributes such that the set difference $C - B$ is nonempty. In particular, if B is an intent then, for each $y \in Y - B$, $(B \cup \{y\})^{\downarrow\uparrow}$ is an intent which is strictly greater than B.

In what follows we introduce an algorithm which recursively computes all the intents so that for each new intent B, we compute its "descendant intents" by adding attributes from $Y - B$ to B and computing the closure. The recursive procedure will be described in the next section. We now focus on the problem of computing $(B \cup \{y\})^{\downarrow\uparrow}$.

Procedure `compute_closure`(B, y) accepts two arguments: a characteristic array B of a set of attributes and an attribute y which does not belong to the set of attributes represented by B. Invoked with these arguments, the algorithm proceeds as follows:

- It initializes a local variable D which is the characteristic array of the closure being computed. Before the computation begins, D is filled with 1s, i.e. it represents the whole Y (lines 1–3).
- Then the algorithm goes through all the rows in which $y \in Y$ appears (lines 4–19) and finishes after each such row has been processed.
- First part of the loop (lines 5–11) checks whether the current row contains all attributes from B. This is checked by comparing the corresponding values of the i-th row of the two-dimensional array representing the formal context and the characteristic array of B. After leaving this nested loop, local variable *match* is set to `true` iff the row matches the pattern. Technically speaking, *match* is `true` iff $\{i\}^{\uparrow} \subseteq B$.
- The rest of the main loop (lines 12–18) is executed iff *match* is `true` in which case we compute the intersection of the attributes in D and the current row of the two-dimensional table representing the formal context. This intersection is computed by removing from D all the attributes which are not in the i-th row of *context*.
- Finally, the procedure returns array D (line 20).

We can show that `compute_closure` is sound:

Theorem 1. Let $\langle X, Y, I\rangle$ be a formal context represented by the two-dimensional array *context* and array of lists *rows*. Then, for each $B \subseteq Y$ and $y \in Y$, `compute_closure`(B, y) halts and returns $(B \cup \{y\})^{\downarrow\uparrow}$.

Hint of the proof. The claim is proved by induction over the number of attributes in *rows*$[y]$. In a more detail, we can show that $(B \cup \{y\})^{\downarrow\uparrow}$ equals

$$(B \cup \{y\})^{\downarrow\uparrow} = \bigcap\big\{\{x\}^{\uparrow} \mid \langle x, y\rangle \in I \text{ and } \{x\}^{\uparrow} \subseteq B\big\},$$

i.e. $(B \cup \{y\})^{\downarrow\uparrow}$ is the intersection of rows in the data table in which y appears and which are contained in B. Procedure `compute_closure` goes through all such rows one by one and removes from D the attributes which are not present in the rows. □

Remark 1. (a) Conventional methods for computing the closure $B^{\downarrow\uparrow}$ are based on the definitions (1) and (2) of the arrow operators. These methods are implemented by two-way algorithms which first compute the extent $B^{\downarrow}$ that needs to be stored somewhere, and then it is used to compute the closure $B^{\downarrow\uparrow}$. Contrary to that, procedure `compute_closure` accumulates the closure in its local variable D and it operates in a constant space.

(b) We have used two representations of the input data to improve efficiency of computing closures. The two-dimensional array representation is used when matching rows with the linear array representing a set $B \cup \{y\}$ of attributes. The second representation is used to skip rows in which y does not appear. Such rows do not contribute to the closure $(B \cup \{y\})^{\downarrow\uparrow}$, i.e. they can be disregarded.

Example 1. Consider the data table from Fig. 1 and its representation by the two-dimensional array *context* and array of lists *rows*. Take $B = \{4, 5\}$ and let $y = 7$. For B and y, `compute_closure` proceeds as follows. Since *rows*$[7] = (0, 1, 4)$, the procedure goes through objects 0, 1, and 4, only. Before the iteration begins, D is set to Y (i.e., $D[y] = 1$ for each $y = 0, \ldots, 7$). In case of $i = 0$, we have a match because *context*$[0, 4] = 1$ and *context*$[0, 5] = 1$.

Thus, $D[0]$, $D[3]$, and $D[6]$ are set to 0. As a consequence, D now represents set $\{1,2,4,5,7\}$ of attributes. For $i = 1$, we do not have a match because $context[1,4] = 0$. Hence, we proceed with $i = 4$. In this case, we have a match and, consequently, $D[2]$ and $D[3]$ will be set to 0. After that, D represents $\{1,4,5,7\} = (B \cup \{7\})^{\downarrow\uparrow}$ which is returned as the result of the computation.

4 The Algorithm and its Properties

The previous section described how we can efficiently compute the closure $(B \cup \{y\})^{\downarrow\uparrow}$ of $B \cup \{y\}$. In this section we introduce an algorithm which lists all fixed points of the closure operator $^{\downarrow\uparrow}\colon 2^Y \to 2^Y$.

The core of our algorithm is a recursive procedure `generate_from` which computes the intents using a depth-first search. During the search, we compute a new intent and check whether the intent has already been found. If not, we process the intent (e.g., we print it on the screen) and proceed with generating further intents which result from the new intent by adding attributes and computing corresponding closures. A similar idea has been used in the Lindig's algorithm [Kuznetsov and Obiedkov, 2002; Lindig, 2000]. The key difference between our algorithm and that from [Lindig, 2000] and its modifications is that we test whether the new intent has already been found. Lindig's algorithm and its modifications use an additional data structure to store found intents. Thus, after a new intent is computed, Lindig's algorithm looks up for the intent in a data structure, typically a search tree of a hashing table. Needless to say, the structure of found intents must be well organized in order to achieve quick responses when looking up for intents.

Our algorithm uses another idea. We generate the intents in a unique order which ensures that one intent is processed exactly once. Before we introduce the procedure for computing all fixed points of $^{\downarrow\uparrow}$ (i.e., all intents), let us comment some more on the order in which the intents are processed. New intents are generated as closures of $B \cup \{y\}$, where B is an intent and $y \in Y$ is not present in B. Let D be the closure of $B \cup \{y\}$, i.e. $D = (B \cup \{y\})^{\downarrow\uparrow}$. After computing D using procedure `compute_closure`, we check whether

$$D \cap \{0,1,\ldots,y-1\} = B \cap \{0,1,\ldots,y-1\} \qquad (3)$$

is true. Note that "$\supseteq$"-part of (3) is trivial. Moreover, (3) is true iff D agrees with B on the attributes $0,1,\ldots,y-1$. In other words, (3) is true iff, for each $i \in \{0,1,\ldots,y-1\}$: $i \in D$ iff $i \in B$. Thus, condition (3) expresses the fact that the closure D of $B \cup \{y\}$ does not contain any new attributes which are "before y". Condition (3) will be used to check whether we should process D. If (3) will be false, we will not process D because due to the depth-first search method, D has already been processed. We will comment on this issue more in the description of the algorithm and in the proof of its soundness.

The algorithm for computing all intents is represented by a procedure `generate_from`(B, y) that accepts two arguments. First, a characteristic array B of an initial intent from which we start generating other intents (the descendants of B), and an attribute y which is the first attribute to be added to B. After its invocation, `generate_from` proceeds as follows:

Procedure `generate_from`(B, y)

```
1  process B (e.g., print B on screen);
2  if B = Y or y > n then
3  |  return
4  end
5  for j from y upto n do
6  |  if B[j] = 0 then
7  |  |  set B[j] to 1;
8  |  |  set D to compute_closure(B, j);
9  |  |  set skip to false;
10 |  |  for k from 0 upto j − 1 do
11 |  |  |  if D[k] ≠ B[k] then
12 |  |  |  |  set skip to true;
13 |  |  |  |  break for loop ;
14 |  |  |  end
15 |  |  end
16 |  |  if skip = false then
17 |  |  |  generate_from(D, j + 1);
18 |  |  end
19 |  |  set B[j] to 0;
20 |  end
21 end
22 return
```

- It processes the attributes in the intent B (e.g., it prints the attributes in B on screen, it adds B to a data structure, etc).
- Then, the procedure checks whether B contains all the attributes from Y, i.e. whether B represents the greatest intent, in which case we exit current branch of recursion (lines 2–4). If y (current attribute) is greater than n (the number of the last attribute), we also exit this branch of recursion because $y > n$ indicates that all attributes have been processed.
- The main loop (lines 5–21) iterates over all remaining attributes, starting with the attribute y.
- In the body of the main loop (lines 6–20), j denotes the current attribute which we are about to add to B. The if-condition at line 6 checks whether j is already present in B. If so, we proceed with another attribute. If j is not present in B, we try to generate new intent from $B \cup \{j\}$ (lines 7–19).
- At lines 7 and 8, we add j to B and compute the closure of $B^{\downarrow\uparrow}$ which is further denoted by D.
- The loop between lines 9–15 checks whether B and D satisfy condition (3) for y being j. A flag *skip* is initially set to false. The flag is reset to true iff there is an attribute $k < j$ such that B and D disagree on k.
- At this point we test whether D can be generated another way and whether we should proceed with generating the descendant intents of D. The main idea here is that if any of the attributes in D, which

are prior to j, are not present in B then there is another branch of recursion which can generate D. Namely, it is the branch processing the left-most attribute which is present in D but not in B. Therefore, we proceed only iff D and B agree on all attributes up to $j-1$ which is iff *skip* is false.

– If *skip* is false, we make a recursive call of the procedure `generate_from` to compute descendant intents of D, starting with the next attribute $j+1$ (line 17).
– After returning from the recursion or in case of *skip* being true, we remove the attribute j from B (line 19) and continue with processing next attribute.

In order to compute all the intents with given support, we first compute the closure $\emptyset^{\downarrow\uparrow}$ of the empty set of attributes and then we invoke `generate_from` with the *intent* set to the array representation of $\emptyset^{\downarrow\uparrow}$ and with y being 0. The following assertion says that the algorithm is sound:

Theorem 2. Let $\langle X, Y, I\rangle$ be a formal context represented by the two-dimensional array *context* and array of lists *rows*. Then, `generate_from`$(\emptyset^{\downarrow\uparrow}, 0)$ processes all intents of $\langle X, Y, I\rangle$.

Hint of the proof. Obviously, the sets of attributes which are processed by the algorithm (line 1 of the algorithm) are intents. It suffices to prove that the algorithms processes each intent exactly once. The fact that each intent is processed at least once can be proved by the analysis of the depth-recursive search for concepts. At the same time, we can prove that each intent is processed at most once because of (3). The full proof of the claim is postponed to a full version of this paper. □

Example 2. Let us return to the formal context depicted in Fig. 1 and the corresponding two-dimensional array *context* and array of lists *rows*. The data table induces 15 formal concepts which are given by the following intents:

$B_1 = \emptyset$, $B_9 = \{0, 6, 7\}$,
$B_2 = \{0, 6\}$, $B_{10} = \{1, 5\}$,
$B_3 = \{0, 1, 5, 6\}$, $B_{11} = \{1, 2, 5\}$,
$B_4 = \{0, 1, 2, 3, 5, 6\}$, $B_{12} = \{1, 2, 4, 5, 7\}$,
$B_5 = \{0, 1, 2, 3, 4, 5, 6, 7\} = Y$, $B_{13} = \{1, 4, 5\}$,
$B_6 = \{0, 1, 4, 5, 6, 7\}$, $B_{14} = \{1, 4, 5, 7\}$,
$B_7 = \{0, 3, 6\}$, $B_{15} = \{7\}$,
$B_8 = \{0, 3, 6, 7\}$,

which are listed in the order in which they are computed by procedure `generate_from` when it is invoked with $\emptyset^{\downarrow\uparrow} = \emptyset$ and 0. Therefore, we have the following collection of formal concepts:

$C_1 = \langle X, \emptyset\rangle$, $C_9 = \langle\{1, 4\}, \{0, 6, 7\}\rangle$,
$C_2 = \langle\{1, 2, 4\}, \{0, 6\}\rangle$, $C_{10} = \langle\{0, 2, 3, 4\}, \{1, 5\}\rangle$,
$C_3 = \langle\{2, 4\}, \{0, 1, 5, 6\}\rangle$, $C_{11} = \langle\{0, 2\}, \{1, 2, 5\}\rangle$,
$C_4 = \langle\{2\}, \{0, 1, 2, 3, 5, 6\}\rangle$, $C_{12} = \langle\{0\}, \{1, 2, 4, 5, 7\}\rangle$,
$C_5 = \langle\emptyset, Y\rangle$, $C_{13} = \langle\{0, 3, 4\}, \{1, 4, 5\}\rangle$,
$C_6 = \langle\{4\}, \{0, 1, 4, 5, 6, 7\}\rangle$, $C_{14} = \langle\{0, 4\}, \{1, 4, 5, 7\}\rangle$,
$C_7 = \langle\{1, 2\}, \{0, 3, 6\}\rangle$, $C_{15} = \langle\{0, 1, 4\}, \{7\}\rangle$,
$C_8 = \langle\{1\}, \{0, 3, 6, 7\}\rangle$,

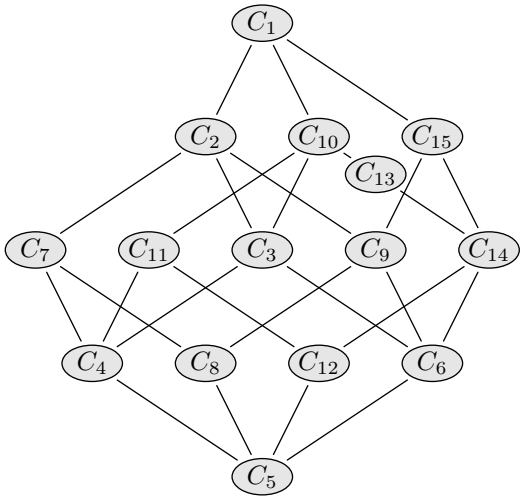

Figure 2: Concept lattice

listed in the same order as the intents. If we order $C_1, \ldots, C_{15}$ by the subconcept-superconcept hierarchy (see Section 2), we get a concept lattice which is depicted in Fig. 2. If we focus on the order in which the intents are computed, we can clearly see the role of condition (3). The first intent which is computed by the procedure is $B_1 = \emptyset^{\downarrow\uparrow} = \emptyset$. In a further step, we obtain B_2 as a closure of $(B_1 \cup \{0\})^{\downarrow\uparrow} = \{0, 6\}$. Moreover, $B_3 = (B_2 \cup \{1\})^{\downarrow\uparrow} = \{0, 1, 5, 6\}$ and in the next step, $B_4 = (B_3 \cup \{2\})^{\downarrow\uparrow} = \{0, 1, 2, 3, 5, 6\}$. Then, 3 is already present in B_4, i.e. we proceed with attribute 4 and compute $B_5 = (B_4 \cup \{4\})^{\downarrow\uparrow} = Y$. Since B_5 contains all attributes, we exit this branch of recursion and return to B_4 and try to generate a new intent by adding next attribute. The last attribute used was 4. Since 5 and 6 are present in B_4, the only option is to add 7. But $(B_4 \cup \{7\})^{\downarrow\uparrow} = Y$, i.e. $4 \in (B_4 \cup \{7\})^{\downarrow\uparrow}$ and $4 \notin B_4$. That is, we do not proceed with intent $(B_4 \cup \{7\})^{\downarrow\uparrow}$ because (3) is violated. This is right because this intent has already been processed as B_5. An analogous situation appears several times during the computation. For instance, in case of $B_2 = \{0, 6\}$ and adding attribute 5, we obtain $(B_2 \cup \{5\})^{\downarrow\uparrow} = \{0, 1, 5, 6\}$, i.e. $1 \in (B_2 \cup \{5\})^{\downarrow\uparrow}$ but $1 \notin B_2$. Hence, we do not proceed with $(B_2 \cup \{5\})^{\downarrow\uparrow}$ at this point (again, the intent $B_3 = \{0, 1, 5, 6\}$ has already been processed).

Remark 2. The algorithm described by procedure `generate_from` computes all intents, i.e. all fixed points of a particular closure operator. As we can see from the code of procedure `generate_from`, it is not dependant on a definition of the closure operator. The procedure can be used to compute fixed points of any closure operator $cl\colon 2^X \to 2^X$ (defined on a finite universe set X) provided that `compute_closure`(B, y) returns the closure $cl(B \cup \{y\})$ of $B \cup \{y\}$.

5 Implementation and its Efficiency

The algorithm has been implemented in ANSI Common LISP and ANSI C and tested on 32-bit and 64-bit

hardware architectures (i386, x86_64, and sparc64). In both the implementations we have used linear data structures to represent the two-dimensional array (context) and array of lists (rows). After the data structures are initialized, our implementation does not use any additional dynamic data structures which significantly improves the performance. Another factor that improves the speed of the algorithm is that most of the set operations with attributes can be implemented by bitarray operations: bitwise "and", "not", and "or" which are implemented in virtually all computer processors.

We have run several experiments to compare the algorithm with other algorithms. During the experiments, we have used implementations of Ganter's and Lindig's algorithms and their improved variants, see [Ganter, 1984; Lindig, 2000]. Our algorithm outperforms all known algorithms, especially if we deal with sparse data sets. Although the comparison is a bit problematic, because not all implementations are widely available, we have a strong indication that our algorithm is considerably faster. Namely, we have compared our algorithm to Sklenar's modification of the Bordat's algorithm which is believed to be the fastest algorithm for computing intents to date. Our algorithm outperforms this modified version of Bordat's algorithm significantly. For instance, we have tested the computing time with several data sets from the UCI Machine Learning Repository [Asuncion and Newman, 2007]. For the well-known MUSHROOM data set, which consists of $8,124$ objects and 119 attributes, there are $238,710$ formal concepts. The analysis of the MUSHROOM dataset has been run on the same computer (Intel Xeon 4, 3.20 GHz, 1 GB RAM), our algorithm finishes the computation in 27 seconds ($>$ 8840 intents per second) whereas the Bordat's algorithm needed more than 80 seconds. In addition to that, our algorithm runs in a constant space (it does not allocate additional dynamic data structures during the operation except for the stack which handles recursive procedure calls) and has consumed just 2 MB of memory during the computation which is much less than, e.g., Lindig's algorithm which uses search trees or hash tables to store found intents.

6 Conclusions

We have introduced an algorithm for computing intents and, in general, fixed points of closure operators. The algorithm has been designed to give satisfactory performance. Our algorithm uses a unique representation of the input data which allows us to compute the closure of a set of attributes efficiently. Namely, we use an efficient bitarray representation for each row of the data table and an array of lists of rows which contain given attributes. The core of our algorithm is a recursive procedure computing fixed points of closure operators. The implementation of our algorithm avoids any dynamic data structures (like trees) which are usually used to store the found closed sets of attributes. Instead, we generate the closed sets in a unique order which ensures that one closure is not generated multiple times. The algorithm outperforms known algorithms for computing intents. Our future research will focus on:

- Comparison with further algorithms.
- Further refinements of the algorithm.
- Multi-threaded and distributed versions of the algorithm.
- Generalization of the algorithm (e.g., processing of graded attributes).
- Focusing on generating constrained intents. In [Belohlavek and Vychodil, 2008], we have introduced background knowledge to formal concept analysis which can significantly reduce the number of intents which are extracted from data, focusing on the intents which are compatible with the background knowledge, only. It may be appealing to extend the present algorithm so that it will incorporate the background knowledge.

References

[Asuncion and Newman, 2007] Asuncion A., Newman D. J., *UCI Machine Learning Repository.* University of California, Irvine, School of Information and Computer Sciences, 2007. http://www.ics.uci.edu/~mlearn/MLRepository.html

[Belohlavek and Vychodil, 2008] Belohlavek R., Vychodil V.: Adding background knowledge to formal concept analysis via attribute dependency formulas. *Proc. ACM SAC 2008* (to appear).

[Carpineto and Romano, 2004] Carpineto C., Romano G.: *Concept Data Analysis. Theory and Applications.* J. Wiley, 2004.

[Ganter, 1984] Ganter B.: Two basic algorithms in concept analysis. FB4-Preprint No. 831, TH Darmstadt, 1984.

[Ganter and Wille, 1999] Ganter B., Wille R.: *Formal Concept Analysis. Mathematical Foundations.* Springer, Berlin, 1999.

[Kuznetsov and Obiedkov, 2002] Kuznetsov S., Obiedkov S.: Comparing performance of algorithms for generating concept lattices. *J. Exp. Theor. Artif. Int.* **14**(2–3)(2002), 189–216.

[Lindig, 2000] Lindig C.: Fast concept analysis. In: Stumme G.: *Working with Conceptual Structures– –Contributions to ICCS 2000.* Shaker Verlag, Aachen, 2000, 152–161.

[Wille, 1982] Wille R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In: I. Rival (Ed.): Ordered Sets, 445–470, Reidel, Dordrecht-Boston, 1982.