

Faster Algorithms for Constructing a Galois Lattice, Enumerating All Maximal Bipartite Cliques and Closed Frequent Sets

Vicky Choi ^{*} Yang Huang [†]

February 17, 2006

Abstract

In this paper, we give a fast algorithm for constructing a Galois lattice of a binary relation. When the binary relation is represented as a bipartite graph, each vertex of the lattice (called a *concept*) corresponds to a maximal bipartite clique of the bipartite graph. Thus, our algorithm also enumerates all maximal bipartite cliques. Further, our algorithm can be naturally modified to compute only large concepts that are known as closed frequent sets in data mining. The running time of our algorithm depends on the lattice structure and is faster than all other existing algorithms for these problems. Let \mathcal{B} denote the set of all concepts, and $\mathcal{L} = \langle \mathcal{B}, \prec \rangle$ be the corresponding lattice. For a concept $C \in \mathcal{B}$, a descendant $D = (\text{ext}(D), \text{int}(D))$ of C is called an *upper descendant* of C if there exists $i \in \text{int}(D)$ such that for any descendant $E \prec C$ with $i \in \text{int}(E)$, $\text{ext}(E) \subseteq \text{ext}(D)$. Denote the set of upper descendants of C by U_C . For most of concepts, U_C consists of all successors of C only. The running time of our algorithm is $O(\sum_{C \in \mathcal{B}} \sum_{a \in \text{ext}(C)} |\{D \in U_{G(C)} : a \in \text{ext}(D)\}|)$ where $G(C)$ is a predecessor of C .

1 Introduction

Given a bipartite graph $G = (\mathcal{O} \cup \mathcal{M}, E)$, a pair (A, B) with $A \subseteq \mathcal{O}$ and $B \subseteq \mathcal{M}$ of G is called a *bipartite clique* if for all $a \in A$ and all $b \in B$, $(a, b) \in E$. A bipartite clique (A, B) is *maximal* if there does not exist a bipartite clique (C, D) such that (1) $A \subseteq C, B \subset D$ or (2) $A \subset C, B \subseteq D$. In Formal Concept Analysis (FCA) [12], the bipartite graph $G = (\mathcal{O} \cup \mathcal{M}, E)$ is represented by a binary relation $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{M}$. The triple $(\mathcal{O}, \mathcal{M}, \mathcal{I})$ is called a *context*. Each maximal bipartite clique of the bipartite graph $G = (\mathcal{O} \cup \mathcal{M}, E)$ is called a *concept* of the context $(\mathcal{O}, \mathcal{M}, \mathcal{I})$. Concepts (ordered by set-inclusion) constitute a nice structure, known as a *lattice*. The lattice of all concepts is called *concept* or *Galois lattice*. A concept is also called a *closed frequent set* [32] in data mining.

The equivalence of all these terminologies – maximal bipartite cliques in theoretical computer science (TCS), concepts for applied mathematicians in FCA, and closed frequent sets for researchers in data mining (DM) – was known, e.g. [2, 32]. There is extensive work of the related problems in these three communities, e.g. [1]–[7] in TCS, [9]–[21] in FCA, and [23]–[34] in DM. In general, in TCS, the research focuses on efficiently enumerating all maximal bipartite cliques (of a bipartite graph); in FCA, one is interested in the lattice structure of all concepts; in DM, one is often interested in computing only large closed frequent sets.

Time complexity. Given a bipartite graph, it is not difficult to see that there can be exponentially many maximal bipartite cliques. For problems with potential exponential (in the size of the input) size output, in their seminal paper [5], Johnson et al introduced several notions of *polynomial time* for algorithms for these problems: *polynomial total time*, *incremental polynomial time*, *polynomial delay time*. An algorithm runs in polynomial total time if the time is bounded by a polynomial in the size of the input and the size of the output. An algorithm runs in incremental polynomial time if the time required to generate a successive output is bounded by the size of input and the size of output generated thus far. An algorithm runs in

^{*}Department of Computer Science, Virginia Tech, USA. vchoi@cs.vt.edu.

[†]Department of Computer Science, Rutgers University, USA. yahuang@cs.rutgers.edu

polynomial delay time if the generation of each output is only polynomial in the size of input. It is not difficult to see that polynomial delay is stronger than incremental polynomial (namely an algorithm with polynomial delay time is also running in incremental polynomial), which is stronger than polynomial total time. One says such a problem is NP-hard if it has no polynomial total time algorithm unless $P=NP$. For polynomial delay algorithm, we can further distinguish if the space used is polynomial or exponential in the input size.

Previous work. Observe that the maximal bipartite clique (MBC) problem is a special case of the maximal clique problem in a general graph. Namely, given a bipartite graph $G = (V_1, V_2, E)$, a maximal bipartite clique corresponds to a maximal clique in $\tilde{G} = (V_1 \cup V_2, \tilde{E})$ where $\tilde{E} = E \cup (V_1 \times V_1) \cup (V_2 \times V_2)$. Consequently, any algorithm for enumerating all maximal cliques in a general graph, e.g., [7, 5], also solves the MBC problem. In fact, the best known algorithm in enumerating all maximal bipartite cliques, which was proposed by Makino and Uno [6] that takes $O(\Delta^2)$ polynomial delay time where Δ is the maximum degree of G , was based on this approach. The fact that the set of maximal bipartite cliques constitutes a lattice was not observed in the paper and thus the property was not utilized for the enumeration algorithm.

In FCA, much of research has been devoted to study the properties of the lattice structure. There are several algorithms, e.g. [17, 21, 16], that construct the lattice, i.e. computing all concepts together with its lattice order. There are also some algorithms that compute only concepts, e.g. [19, 12]. See [14] for a comparison studies of these algorithms. The best polynomial total time algorithm was by Nourine and Raynaud [17] with $O(nm|\mathcal{B}|)$ time and $O(n|\mathcal{B}|)$ space, where $n = |\mathcal{O}|$ and $m = |\mathcal{M}|$ and \mathcal{B} denote the set of all concepts. This algorithm can be easily modified to run in $O(mn)$ incremental time [18]. Observe that the space of total size of all concepts is needed if one is to keep the entire structure explicitly. There were several other algorithms, e.g. [12, 16], all run in $O(n^2m)$ polynomial delay. There is another algorithm [21] that is based on divide-and-conquer approach, but the analytical running time of the algorithm is unknown as it is difficult to analyze.

There are several heuristics in data mining for computing closed frequent sets, such as CHARM [33, 34], and CLOSET [27, 30]. To our best knowledge, the algorithm with theoretical analysis running time was given in [2] with $O(m^2n)$ incremental polynomial running time, where $n = |\mathcal{O}|$ and $m = |\mathcal{M}|$.

Our Results. In this paper, making use of the lattice structure of concepts, we present a simple and fast algorithm for computing all concepts together with its lattice order. The main idea of the algorithm is that given a concept, when all of its successors are considered together (i.e. in a batch manner), they can be efficiently computed. We compute concepts in the Breadth First Search (BFS) order – the ordering given by BFS traversal of the lattice. When computing the concepts in this way, not only do we compute all concepts but also we identify all successors of each concept. The efficiency is further achieved by using the same implementation technique that one employs to implement the Lexicographic Breadth First Search (LBFS) traversal of a graph in linear time of number of vertices and edges. Another idea of the algorithm is that we make use of the concepts generated to dynamically update the adjacency relations. The running time of our algorithm depends on the lattice structure. Let \mathcal{B} denote the set of all concepts, and $\mathcal{L} = \langle \mathcal{B}, \prec \rangle$ the corresponding lattice. Each concept $C \in \mathcal{B}$ consists of two components and is written as $(\text{ext}(C), \text{int}(C))$, where $\text{ext}(C)$ and $\text{int}(C)$ are the extent and intent of C respectively (see Section 2 for related background and terminology). For a concept $C \in \mathcal{B}$, a descendant $D = (\text{ext}(D), \text{int}(D))$ of C is called an *upper descendant* of C if there exists $i \in \text{int}(D)$ such that for any descendant $E \prec C$ with $i \in \text{int}(E)$, $\text{ext}(E) \subseteq \text{ext}(D)$. Denote the set of upper descendants of C by U_C . In particular, the set of upper descendants of C consists of all successors of C . For most of concepts, U_C consists of only successors of C . The running time of our algorithm is $O(\sum_{C \in \mathcal{B}} \sum_{a \in \text{ext}(C)} |\{D \in U_{G(C)} : a \in \text{ext}(D)\}|)$, where $G(C)$ is a predecessor of C . Our algorithm is faster than the best known algorithms for computing all concepts or constructing a lattice because (1) the algorithm is improved upon an easy algorithm (described in Section 4.1) and (2) the easy algorithm is as fast as the current best algorithms for the problem.

Outline. The paper is organized as follows. In Section 2, we review some background and notation on FCA. In Section 3, we describe some basic properties of concepts that we use in our lattice-construction algorithm. In Section 4, we first describe an easy algorithm that is as fast as the best known algorithms for computing all concepts or constructing a lattice. Then we describe how to improve the algorithm based on the idea of compressing attributes. Finally, we further improve the algorithm by eliminating explicit successor-checking. We conclude with discussion in Section 5.

2 Background and Terminology on FCA

In FCA, a triple $(\mathcal{O}, \mathcal{M}, \mathcal{I})$ is called a *context* where \mathcal{O} and \mathcal{M} are sets and $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{M}$. The elements of \mathcal{O} and \mathcal{M} are called *objects* and *attributes* respectively. The context is often represented by a *cross-table* as shown in Figure 1.

For $a \in \mathcal{M}$, define $\text{nbr}(a) = \{g \in \mathcal{O} : (g, a) \in \mathcal{I}\}$. Similarly, for $g \in \mathcal{O}$, define $\text{nbr}(g) = \{a \in \mathcal{M} : (g, a) \in \mathcal{I}\}$. For a subset A of \mathcal{O} (or \mathcal{M}), $A' = \bigcap_{a \in A} \text{nbr}(a)$. By definition, it is easy to check that $A \subseteq A''$, and if $A \subset B$, then $B' \subset A'$. The function $'$ induces a *Galois connection* between \mathcal{O} and \mathcal{M} (the underlying sets are ordered by set-inclusion \subseteq relation). Readers are referred to [12] for properties of the Galois connection. A pair $C = (A, B)$, with $A \subseteq \mathcal{O}$ and $B \subseteq \mathcal{M}$, is a *concept* if and only if $A' = B$ and $B' = A$. A and B are called the *extent* and *intent* of C respectively, written as $A = \text{ext}(C)$. and $B = \text{int}(C)$. The set of all concepts of the context $(\mathcal{O}, \mathcal{M}, \mathcal{I})$ is denoted by $\mathcal{B}(\mathcal{O}, \mathcal{M}, \mathcal{I})$ or simply \mathcal{B} when the context is understood.

Let (A_1, B_1) and (A_2, B_2) be two concepts in \mathcal{B} . Observe that if $A_1 \subseteq A_2$, then $B_2 \subseteq B_1$. We order the concepts in \mathcal{B} by the following relation \prec :

$$(A_1, B_1) \prec (A_2, B_2) \iff A_1 \subseteq A_2 (B_2 \subseteq B_1).$$

It is not difficult to see that the relation \prec is a partial order on \mathcal{B} . In fact, $\mathcal{L} = \langle \mathcal{B}, \prec \rangle$ is a complete lattice and it is known as the *concept* or *Galois* lattice of the context $(\mathcal{O}, \mathcal{M}, \mathcal{I})$. For $C, D \in \mathcal{B}$ with $C \prec D$, if for all $E \in \mathcal{B}$ such that $C \prec E \prec D$ implies that $E = C$ or $E = D$, then C is called the *successor* ^{*}(or *lower neighbor*) of D , and D is called the *predecessor* (or *upper neighbor*) of C . The diagram representing an ordered set (where only successors/predecessors are connected by edges) is called a *Hasse diagram* (or a line diagram). See Figure 1 for an example of the line diagram of a Galois lattice.

Note that each concept (A, B) is uniquely determined by either its extent, A , or by its intent, B . We denote the concepts restricted to the objects \mathcal{O} by $\mathcal{B}_{\mathcal{O}} = \{A \subseteq \mathcal{O} : A'' = A\}$. When there is no danger of confusion, for $A \in \mathcal{B}_{\mathcal{O}}$, we refer to its corresponding intent by $\text{int}(A)$ and the corresponding concept by $(A, \text{int}(A))$. Also, the order \prec is completely determined by the inclusion order on $2^{\mathcal{O}}$ or equivalently by the reverse inclusion order on $2^{\mathcal{M}}$. That is, $(A, \text{int}(A)) \prec (P, \text{int}(P))$ if and only if $A \subseteq P$.

Following the convention, we will simplify the standard set notation by dropping out all the separators (e.g., 123 will stand for the set of attributes $\{1, 2, 3\}$ and $abcd$ for the set of objects $\{a, b, c, d\}$).

3 Basic Properties

In this section, we describe some basic properties of the concepts on which our lattice construction algorithms are based.

Proposition 3.1 *Let $(C, \text{int}(C))$ be a concept in $\mathcal{B}(\mathcal{O}, \mathcal{M}, \mathcal{I})$. For $i \in \mathcal{M} \setminus \text{int}(C)$, if $C_i = C \cap \text{nbr}(i)$ is not empty, then $C_i \in \mathcal{B}_{\mathcal{O}}$, that is, C_i is an extent of some concept.*

Conversely, for any set $C \in \mathcal{B}_{\mathcal{O}}$, $C \neq \emptyset$, there exists a concept $(P, \text{int}(P))$ such that $C = P \cap \text{nbr}(i)$ for some $i \in \mathcal{M} \setminus \text{int}(P)$.

Proof. For $i \in \mathcal{M} \setminus \text{int}(C)$, suppose that $C_i = C \cap \text{nbr}(i)$ is not empty. We will show that $C_i'' = C_i$. Since $C_i \subseteq C_i''$, it remains to show that $C_i'' \subseteq C_i$. By definition, $(\text{int}(C) \cup \{i\})' = (\bigcap_{j \in \text{int}(C)} \text{nbr}(j)) \cap \text{nbr}(i) =$

^{*}Some authors called this as immediate successor.

$C \cap \text{nbr}(i) = C_i$. Since $(\text{int}(C) \cup \{i\}) \subseteq (\text{int}(C) \cup \{i\})''$, $(\text{int}(C) \cup \{i\}) \subseteq C_i'$. By the property of $'$, $C_i'' \subseteq (\text{int}(C) \cup \{i\})' = C_i$.

Conversely, for any set $C \in \mathcal{B}_{\mathcal{O}}$, $C \neq \mathcal{O}$, let $(P, \text{int}(P))$ be a predecessor of $(C, \text{int}(C))$, then $C = P \cap \text{nbr}(i)$ where $i \in \text{int}(C) \setminus \text{int}(P)$. ■

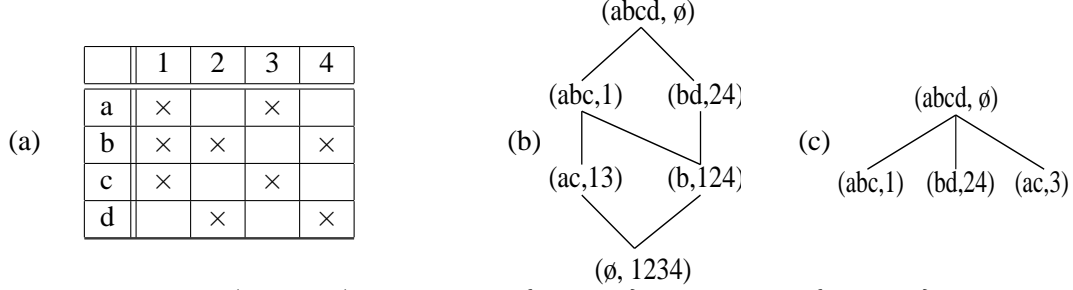


Figure 1: (a) a context $(\mathcal{O}, \mathcal{M}, \mathcal{I})$ with $\mathcal{O} = \{a, b, c, d\}$ and $\mathcal{M} = \{1, 2, 3, 4\}$. The cross \times indicates a pair in the relation \mathcal{I} . (b) the corresponding Galois/concept lattice. (c) $\text{Child}(abcd, \emptyset) = \{(abc, 1), (bd, 24), (ac, 3)\}$.

Example. Consider the concept $(C, \text{int}(C)) = (abcd, \emptyset)$ of context in Figure 1, we have $C_1 = abc, C_2 = bd, C_3 = ac, C_4 = bd$.

For a concept $(C, \text{int}(C))$ in $\mathcal{B}(\mathcal{O}, \mathcal{M}, \mathcal{I})$, denote the set of remaining attributes $\{i \in \mathcal{M} \setminus \text{int}(C) : C \cap \text{nbr}(i) \neq \emptyset\}$ by $R(C)$. Consider the following equivalence relation \sim on $R(C)$: $i \sim j \iff C \cap \text{nbr}(i) = C \cap \text{nbr}(j)$, for $i \neq j \in R(C)$. Let I_1, \dots, I_t be the equivalence classes induced by \sim , i.e. $R(C) = I_1 \cup \dots \cup I_t$, and, $C_i = C_j$ for any $i \neq j \in I_k$, $1 \leq k \leq t$. For $1 \leq k \leq t$, we define a new pair $(E_k, \text{pre_int}_C(E_k))$ where $E_k = C_j$ for $j \in I_k$ and $\text{pre_int}_C(E_k) = I_k$. Call the set of these pairs $\{(E_k, \text{pre_int}_C(E_k)) : 1 \leq k \leq t\}$ the children of $(C, \text{int}(C))$, and denote the set by $\text{Child}(C, \text{int}(C))$. For example, in Figure 1, $\text{Child}(abcd, \emptyset) = \{(abc, 1), (bd, 24), (ac, 3)\}$.

From Proposition 3.1, the object set E in each pair $(E, \text{pre_int}_C(E)) \in \text{Child}(C, \text{int}(C))$ is an extent of some concept. We denote the extent set $\{E_k : 1 \leq k \leq t\}$ by $\text{ExtChild}(C)$. Thus, $\text{Child}(C, \text{int}(C)) = \{(E, \text{pre_int}_C(E)) : E \in \text{ExtChild}(C)\}$.

Recall that $\mathcal{L} = \langle \mathcal{B}, \prec \rangle$ and $\mathcal{L}_{\mathcal{O}} = \langle \mathcal{B}_{\mathcal{O}}, \subseteq \rangle$ are order-isomorphic. We have the property that $(E, \text{int}(E))$ is a successor of $(C, \text{int}(C))$ in \mathcal{L} if and only if E is a successor of C in $\mathcal{L}_{\mathcal{O}}$. For $E \in \text{ExtChild}(C)$, we call E a *child* (or *upper descendant*) of C and C a *parent* of E . It is not difficult to see that if E is a successor of C , then E is a child of C . However, not every child of C is a successor of C . For the example in Figure 1, $\text{ExtChild}(abcd) = \{abc, bd, ac\}$, where abc and bd are successors of $abcd$ but ac is not. Similarly, if P is a predecessor of C , then P is a parent of C but it is not necessarily true that every parent of C is a predecessor of C .

Lemma 3.2 *If $E \in \text{ExtChild}(C)$ is a successor of C , then $\text{int}(E) = \text{int}(C) \cup \text{pre_int}_C(E)$. If $E \in \text{ExtChild}(C)$ is not a successor of C , then there exists $F \in \text{ExtChild}(C)$ such that $E \subset F$.*

Proof. Let $E \in \text{ExtChild}(C)$. Suppose that $\text{int}(C) \cup \text{pre_int}_C(E) \subset \text{int}(E)$. Let $s \in \text{int}(E) \setminus (\text{int}(C) \cup \text{pre_int}_C(E))$, then we have $E \subseteq (C \cap \text{nbr}(s))$. By Proposition 3.1, there exists $F \in \text{ExtChild}(C)$ such that $F = C \cap \text{nbr}(s)$. By definition of $\text{pre_int}_C(E)$, $E \neq F$, i.e. $E \subset F$. Thus, if E is a successor of C , it would imply $\text{int}(E) \setminus (\text{int}(C) \cup \text{pre_int}_C(E)) = \emptyset$, that is, $\text{int}(E) = \text{int}(C) \cup \text{pre_int}_C(E)$. ■

Corollary 3.3 *Let $E \in \text{ExtChild}(C)$. Suppose that E is not contained in any of its siblings in $\text{ExtChild}(C)$, i.e. $E \not\subseteq F$ for any $F \in \text{ExtChild}(C)$, then E is a successor of C .*

For $E \in \text{ExtChild}(C)$, we call $\text{pre_int}_C(E)$ the attribute set of E induced by C . When C is understood, we simply refer $\text{pre_int}_C(E)$ the *induced* attribute set of E . We call the set $\text{int}(C) \cup \text{pre_int}_C(E)$ the *combined* attribute set of E . We say that $\text{pre_int}_C(E)$ is *complete* if its combined attribute set equals to

its intent, i.e. $\text{int}(E) = \text{int}(C) \cup \text{pre_int}_C(E)$. If $E \in \text{ExtChild}(C)$ is a successor of C , we call the corresponding pair $(E, \text{pre_int}_C(E))$ a successor pair of $(C, \text{int}(C))$, otherwise, the non-successor pair. From Lemma 3.2, we know that all induced attribute sets in successor pairs are complete.

Recall that to compute the lattice structure of all concepts, it suffices to find all successor/predecessor relations in addition to computing all concepts. Denote the successors of C by $\text{Succ}(C)$. Then by Lemma 3.2, $\text{Succ}(C) = \text{ExtChild}(C) \setminus \{E : \exists F \in \text{ExtChild}(C) \text{ such that } E \subset F\}$.

In general, an extent can have several parents. For example, b is in $\text{ExtChild}(abc)$ and $\text{ExtChild}(bd)$, with $\text{pre_int}_{abc}(b) = 24$ and $\text{pre_int}_{bd}(b) = 1$ respectively. In the following, we give another characterization of $\text{Succ}(C)$ in terms of the size of the attribute sets.

Lemma 3.4 *Suppose that E is not a successor of C , then $E \in \text{ExtChild}(F)$, for all $F \in \text{ExtChild}(C)$ with $E \subset F$. Furthermore, $|\text{int}(C)| + |\text{pre_int}_C(E)| < |\text{int}(F)| + |\text{pre_int}_F(E)|$.*

Proof. Suppose that E is not a successor of C and $E \subset F$ with $F \in \text{ExtChild}(C)$. For $i \in \text{pre_int}_C(E)$, we have $E = C \cap \text{nbr}(i)$. Since $E \subseteq F$, intersecting both sides by $\text{nbr}(i)$, we have $E \cap \text{nbr}(i) \subseteq F \cap \text{nbr}(i)$. On the other hand, $F \subseteq C$, $F \cap \text{nbr}(i) \subseteq C \cap \text{nbr}(i) = E$. Thus, we have $E = E \cap \text{nbr}(i) \subseteq F \cap \text{nbr}(i) \subseteq E$, implying that $E = F \cap \text{nbr}(i)$. By Proposition 3.1, we have $E \in \text{ExtChild}(F)$. Also, we have $i \in \text{pre_int}_F(E)$ (as $i \notin \text{int}(F)$) and hence $\text{pre_int}_C(E) \subseteq \text{pre_int}_F(E)$. Since $F \in \text{ExtChild}(C)$, $|\text{int}(C)| < |\text{int}(F)|$. Consequently, $|\text{int}(C)| + |\text{pre_int}_C(E)| < |\text{int}(F)| + |\text{pre_int}_F(E)|$. ■

Data Structures. Throughout this paper, the binary relation (or the bipartite graph) is represented in adjacency lists of all objects and attributes. The adjacency list of each object (attribute respectively) contains a list of pointers to the adjacent attributes (objects respectively). WLOG, we assume that no object is adjacent to all attributes, and no attribute is adjacent to all objects. We use a trie over the object set to store the extents. Each extent in the trie is associated with a set of attribute set. Thus, each concept is represented by its extent in the trie and the corresponding intent is associated with the extent. To efficiently match the extents, we use hash keys over each branch in the trie. It will take time $O(|C|)$ to search or insert an object set C in the trie.

3.1 Computing $\text{Child}(C, \text{int}(C))$ by Procedure SPROUT

We will describe how to compute $\text{Child}(C, \text{int}(C))$ in $O(\sum_{a \in C} |\text{nbr}(a)|)$ time, using a procedure called SPROUT.

Lemma 3.5 *For $(C, \text{int}(C)) \in \mathcal{B}$, it takes $O(\sum_{a \in C} |\text{nbr}(a)|)$ to compute $\text{Child}(C, \text{int}(C))$.*

Proof. Let $R(C) = \cup_{a \in C} \text{nbr}(a) \setminus \text{int}(C)$. For each $i \in R(C)$, we associate it with a set C_i (which is initialized as an empty set). For each object $a \in C$, we scan through each attribute i in its neighbor list $\text{nbr}(a)$, append a to the set C_i that is associated with i . This step takes $O(\sum_{a \in C} |\text{nbr}(a)|)$. Next we collect all the sets $\{C_i : i \in R(C)\}$. We use a trie to group the same object set: search C_i in the trie; if not found, insert C_i into the trie with $\{i\}$ as its attribute set, otherwise we append i to C_i 's existing attribute set. This step takes $O(\sum_{i \in R(C)} |C_i|) = O(\sum_{a \in C} |\text{nbr}(a)|)$. Thus, this procedure, called SPROUT($C, \text{int}(C)$), takes $O(\sum_{a \in C} |\text{nbr}(a)|)$ time to compute $\text{Child}(C, \text{int}(C))$. ■

In order to compare the running time with the algorithm by Makino and Uno [6], we express the running time in terms of Δ the maximum degree of vertices (objects/attributes). The size of a extent set is $O(\Delta)$. Thus, $O(\sum_{a \in C} |\text{nbr}(a)|) = O(\Delta^2)$.

3.2 Efficient Intent Completion by Procedure COMPLETEINTENT

Recall that not all sets in $\text{ExtChild}(C)$ are the successors of C . From Lemma 3.2, if E is not a successor of C , then its induced attribute set is not complete, i.e. $\text{int}(C) \cup \text{pre_int}_C(E) \neq \text{int}(E)$. Thus, by comparing the combined attribute set, $\text{int}(C) \cup \text{pre_int}_C(E)$, with $\text{int}(E) = \cap_{a \in E} \text{nbr}(a)$, we can determine if E is a successor or not. This can be implemented in Lexicographic Breadth First Search-like manner in

$O(\sum_{a \in E} |\text{nbr}(a)|)$. We call this procedure COMPLETEINTENT. The procedure returns “Yes” if the induced attribute set is complete, otherwise ”No” and set the attribute set $\text{pre_int}_C(E)$ to $\text{int}(E) \setminus \text{int}(C)$. That is, upon return, we always have $\text{int}(C) \cup \text{pre_int}_C(E) = \text{int}(E)$. We summarize this in the following lemma.

Lemma 3.6 *For $(E, \text{pre_int}_C(E)) \in \text{Child}(C, \text{int}(C))$, we can determine whether $\text{pre_int}_C(E)$ is complete, i.e. $\text{pre_int}_C(E) = \text{int}(E) \setminus \text{int}(C)$, in $O(\sum_{a \in E} |\text{nbr}(a)|)$ time.*

4 Algorithms for Constructing A Galois Lattice

4.1 An Easy Algorithm

In this section, we describe an easy algorithm to construct the lattice structure, including generating all concepts and a list of its successors for each concept, in $O(\sum_{C \in \mathcal{B}} \sum_{a \in C} |\text{nbr}(a)|)$ time.

Our algorithm starts with processing the top concept $(C, \text{int}(C)) = (\mathcal{O}, \emptyset)$. First, we generate all the children $\text{Child}(C, \text{int}(C))$ by procedure SPROUT. Then we check each pair $(E, \text{pre_int}_C(E)) \in \text{Child}(C, \text{int}(C))$ by the procedure COMPLETEINTENT. If the induced attribute set $\text{pre_int}_C(E)$ is complete, we add E to the successor list of C . To systematically process concepts so that each concept is processed (and thus sprouted) once and only once, we process concepts in a Breadth First Search (BFS) order. Namely, we enqueue an extent into a queue when the extent is generated (or sprouted) by its first parent. Each extent can have several parents and thus can be generated several times. Recall that the procedure COMPLETEINTENT also completes the attribute set $\text{pre_int}_C(E)$ and an extent is a successor if its induced attribute set is complete. Thus, after the first time an extent was generated, we store E together with the intent set $\text{int}(E)$ in the trie (regardless if E is a successor of C or not). When an extent is generated again, we can avoid calling COMPLETEINTENT to determine if it is a successor of the current parent. Instead we compare its combined attribute set with the intent set in the trie. If the combined attribute set is the same as the intent set, then the extent is a successor of its (current) parent, otherwise it is not.

See Figure 2 for an illustration of the algorithm and Algorithm 2 in the appendix for the pseudo-code.

Time complexity analysis. Each concept calls once the procedure COMPLETEINTENT and once the procedure SPROUT. Thus, the total running time is $O(\sum_{C \in \mathcal{B}} \sum_{a \in C} |\text{nbr}(a)|)$. In terms of polynomial delay, it takes $O(\Delta^3)$ to output a successive concept together with its successor list. If we need only concepts, the algorithm can be modified to run in $O(\sum_{a \in C} |\text{nbr}(a)|) = O(\Delta^2)$ polynomial delay. Thus, this easy algorithm is as fast as the algorithm by Makino & Uno [6] that computes only concepts. It is also as fast as the best lattice-construction algorithm given by Nourine and Raynaud [17], in total polynomial time of $O(mn|\mathcal{B}|)$ (or $O(mn)$ incremental polynomial), and several other algorithms (e.g. see [14]) of $O(n^2m)$ polynomial delay. Remark: The order of the concepts being processed is not important as long as the pair that is being processed is a concept (i.e. the induced attribute set is complete). If we use stack instead of queue, then the concepts are generated in Depth First Search (DFS) order. Also, in the DFS version, if we need only concepts, we can avoid the intent completion procedure. However, we will need to compare the attribute sets for the lattice structure.

4.2 An Improved Algorithm

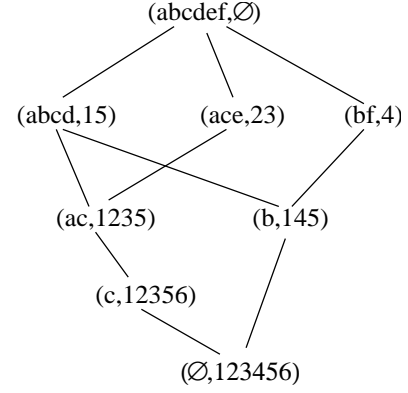
The running time of the above algorithm is dominated by procedures COMPLETEINTENT and SPROUT. Each procedure takes $O(\sum_{a \in C} |\text{nbr}(a)|)$ for processing a concept $(C, \text{int}(C))$. Thus, if we can reduce the size of the adjacency lists (i.e. $|\text{nbr}(a)|$), we can reduce the running time of the algorithm.

Observation. Consider a concept $(C, \text{int}(C)) \in \mathcal{L}$, the extents of all descendants of $(C, \text{int}(C))$ are all subsets of C . To compute the descendants of $(C, \text{int}(C))$, it suffices to consider the adjacency lists of attributes with restriction to C . That is, for $i \in R(C)$, we restrict the adjacent list to C , $\text{nbr}(i) \cap C = \{a \in C : (a, i) \in \mathcal{I}\}$, denoted by $\text{nbr}_C(i)$.

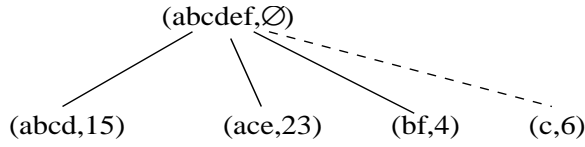
Compressing attributes. For $(E, \text{pre_int}_C(E)) \in \text{Child}(C, \text{int}(C))$, $\text{nbr}_C(i) = \text{nbr}_C(j)$, for any $i, j \in \text{pre_int}_C(E)$ with $i \neq j$. That is, all attributes in $\text{pre_int}_C(E)$ have the same adjacency list when the object

	1	2	3	4	5	6
a	×	×	×		×	
b	×			×	×	
c	×	×	×		×	×
d	×				×	
e		×	×			
f				×		

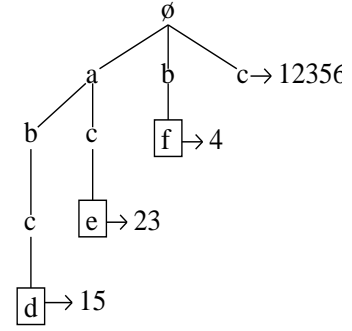
(a)



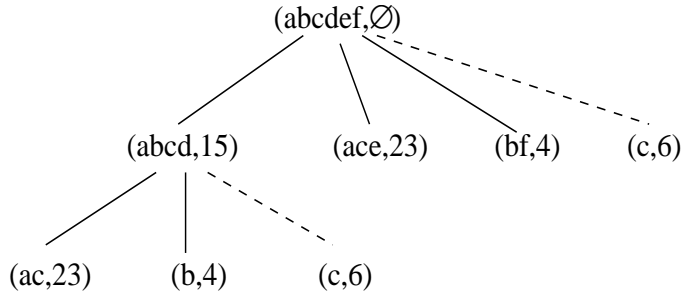
(b)



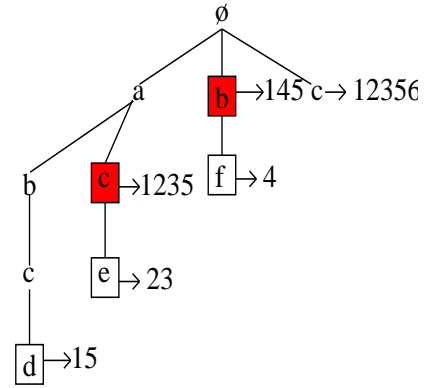
(c)



(d)



(e)



(f)

Figure 2: (a) A context; (b) The corresponding lattice; (c) $\text{Child}(abcdef, \emptyset)$, a solid line indicates a successor while a dash line indicates a non-successor; (d) The trie after inserting $\text{Child}(abcdef, \emptyset)$, each extent is associated with its corresponding intent. At this step, the queue $Q = (abcd, ace, bf)$. (e) Same as (c) for $\text{Child}(abcd, 15)$; (f) The corresponding trie after inserting $\text{Child}(abcd, 15)$. A filled box indicates the newly generated extents. At this step, $Q = (ace, bf, ac, b)$.

set is restricted to C . To reduce the sizes of adjacent lists, we thus can represent the attributes in $\text{pre_int}_C(E)$ by a single element, called a *compressed attribute*. For example, using a single element 15 to represent the two attributes 1 and 5. We call the reduced adjacency lists the compressed adjacency lists. The set of compressed adjacency lists corresponds to a reduced cross-table. For example, the reduced cross table of $\text{Child}(abcdef, \emptyset)$ of the above example is shown in Figure 3.

	15	23	4	6
a	×	×		
b	×		×	
c	×	×		×
d	×			
e		×		
f			×	

Figure 3: Reduced cross-table of $\text{Child}(abcdef, \emptyset)$ of the context in Figure 2.

To make use of the compressed adjacent lists, we proceed the algorithm in a two-level manner. For a concept $(C, \text{int}(C))$, we first generate all its children $\text{Child}(C, \text{int}(C))$ by the procedure **SPROUT**. Then we compress the attributes in each child of $(C, \text{int}(C))$ to get a set of compressed adjacency lists. We then use these compressed adjacency lists to process each child $(E, \text{pre_int}_C(E))$ of $(C, \text{int}(C))$, by procedures **SPROUT** and **COMPLETEINTENT**. That is, instead of using the global adjacency lists, when processing $(E, \text{pre_int}_C(E))$, we use the compressed adjacency lists with restriction to C .

Recall that for a re-generated extent E , we need to compare the new combined attribute set with its (already computed) intent. However, instead of expanding the compressed attributes to make the comparison, we can determine if an extent is a successor or not by the size of attribute sets. This is because an extent is a successor (of its parent) if and only if its combined attribute set is the same as the intent set. This is equivalent to the size of the combined attribute set is the same size as its intent. If an extent is not a successor, its combined attribute set will be a subset of its intent and thus the corresponding size will be smaller. We summarize this characterization in the following lemma.

Lemma 4.1 *If $E \in \text{ExtChild}(C)$ is a successor of C , then $|\text{int}(C)| + |\text{pre_int}_C(E)| = |\text{int}(E)|$. If $E \in \text{ExtChild}(C)$ is not a successor of C , then $|\text{int}(C)| + |\text{pre_int}_C(E)| < |\text{int}(E)|$.*

By associating each compressed attribute with the number of the original attributes that it contains, we can easily keep track of the size of attribute sets. After computation, we can expand the compressed attributes to restore the original attributes. It is not difficult to see that this will not take more asymptotic time. Readers are referred to the full version of this paper for the implementation details.

Time complexity analysis. Now the running time of each procedure (**SPROUT** or **COMPLETEINTENT**) will depend on the size of compressed adjacency lists it uses. It will take $O(\sum_{a \in C} |\text{cnbr}_{G(C)}(a)|)$ where $\text{cnbr}_{G(C)}(a)$ is the compressed adjacency lists with restriction to $G(C)$, when processing an extent C . For the procedure **SPROUT**, $G(C)$ is a predecessor of C ; however, for the **COMPLETEINTENT**, $G(C)$ might be an ancestor of C that is a few levels higher than C . Since the size of a adjacency list restricted to a predecessor is smaller than the size of a adjacency list restricted to a higher ancestor, the running time of this algorithm is dominated by **COMPLETEINTENT**.

4.3 A Further Improved Algorithm

In this section, we further improve the above algorithm by eliminating the **COMPLETEINTENT** procedure. That is, the running time will be dominated by the procedure **SPROUT** on each concept.

We make two modifications to the above algorithm: First, we process the children according to the size of their extents in decreasing order. Based on this processing order, we claim that we can avoid calling **COMPLETEINTENT** to compute the size of intent. Instead, we determine if an extent is a successor or not by

comparing the sizes of the extent's attribute sets generated so far. Namely, when we process an extent, the extent is a successor of its current parent if the size of its combined attribute set is not smaller than the size of attribute set generated up to this point. The correctness follows from the following lemma.

Lemma 4.2 *If $E \in \text{ExtChild}(C)$ is not a successor of C , then one of its siblings F in $\text{ExtChild}(C)$ will be processed earlier than E , and $|\text{int}(C)| + |\text{pre_int}_C(E)| < |\text{int}(F)| + |\text{pre_int}_F(E)|$.*

Proof. By Lemma 3.2, if E is not a successor of C , there exists $P \in \text{ExtChild}(C)$ such that $E \subset P$. Let F be the largest such set. That is, F is not contained by any other sibling in $\text{ExtChild}(C)$. By Corollary 3.3, F is a successor of C . Thus, by the ordering, F will be processed (not necessarily as a child of C) earlier than E . Then by Lemma 3.4, $E \in \text{ExtChild}(F)$ and thus $|\text{int}(C)| + |\text{pre_int}_C(E)| < |\text{int}(F)| + |\text{pre_int}_F(E)|$ as claimed. ■

See Algorithm 1 for the pseudo-code and Figure 4 for an illustration of the algorithm.

Algorithm 1 Constructing a Galois Lattice

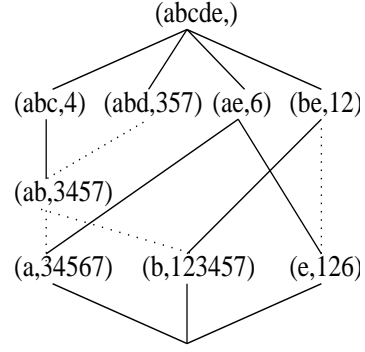
Input: $(\mathcal{O}, \mathcal{M})$ and the adjacency list of each element;
Output: All concepts $(C, \text{int}(C))$ and a successor list for each concept.

- 1: Initialize a trie T for the object set \mathcal{O} ;
- 2: Start with a queue Q containing \mathcal{O} ; \triangleright Each element in Q consists of a pointer pointing to an extent in T .
- 3: $\text{int}(\mathcal{O}) = \emptyset$; $\triangleright (\mathcal{O}, \emptyset)$ is the “top” concept.
- 4: Generate the children of \mathcal{O} by procedure SPROUT;
- 5: **while** Q is not empty **do**
- 6: $C = \text{dequeue}(Q)$;
- 7: Compress attributes in each induced attribute set in the children of C and generate the corresponding compressed adjacency lists accordingly;
- 8: Sort the children of C according to the extent size in decreasing order. Suppose the children are $(E_k, \text{pre_int}_C(E_k))$, $1 \leq k \leq t$ and $|E_i| \geq |E_j|$ for $i < j$.
- 9: **for** $k = 1 \dots t$ **do**
- 10: **if** the size of E_k 's combined attribute set is not smaller than the largest attribute set generated **then**
- 11: Add E_k to the successor list of C ;
- 12: **if** E_k is not sprouted **then**
- 13: Generate the children of E_k by procedure SPROUT;
- 14: For each child of E_k , if its extent exists in T , compare and keep the size of the larger combined attribute set for the extent, otherwise insert the extent into T with the size of its combined attribute set.
- 15: Enqueue E_k into Q ;
- 16: **end if**
- 17: **end if**
- 18: **end for**
- 19: Expand the compressed attributes in $\text{int}(C)$; output the concept $(C, \text{int}(C))$ and its successor list.
- 20: **end while**

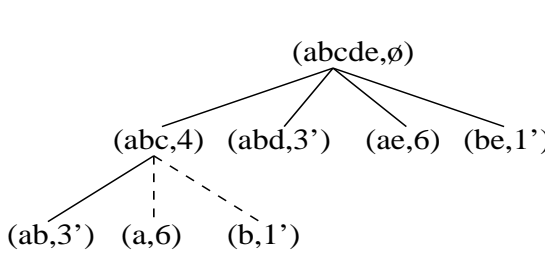
Time complexity analysis. The total running time of the algorithm is $O(\sum_{C \in \mathcal{B}} \sum_{a \in C} |\text{cnbr}_{G(C)}(a)|)$, where $O(\sum_{a \in C} |\text{cnbr}_{G(C)}(a)|)$ is the time for sprouting the extent C , and $G(C)$ is the leftmost predecessor of C and $\text{cnbr}_{G(C)}(a)$ is the size of the compressed adjacency list of a with restriction to $G(C)$. Observe that $|\text{cnbr}_{G(C)}(a)| = |\{E \in \text{ExtChild}(G(C)) : a \in E\}|$. Recall that U_C is the set of upper descendants of C defined by $\{D \in \mathcal{B} : D \prec C \text{ and there exists } i \in \text{int}(D) \text{ such that for all } E \prec C \text{ with } i \in \text{int}(E),$

	1	2	3	4	5	6	7
a			×	×	×	×	×
b	×	×	×	×	×		×
c				×			
d			×		×		×
e	×	×				×	

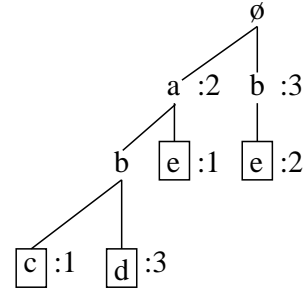
(a)



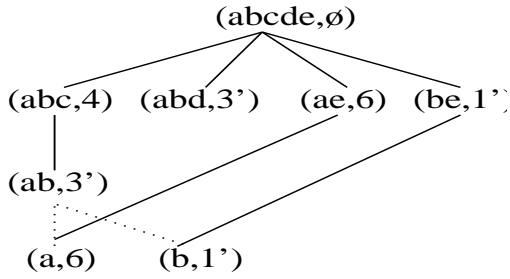
(b)



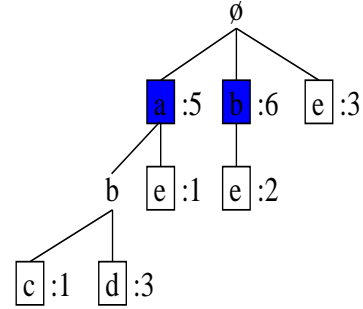
(c)



(d)



(e)



(f)

Figure 4: (a) A context; (b) The corresponding lattice. A solid line indicates the child is generated by the predecessor, while a dotted line indicates the child is generated by another predecessor. (c) Compress attributes in $\text{Child}(abcde, \emptyset)$: $3' = 357$, $1' = 12$ and generate $\text{Child}(abc, 4)$. (d) the trie T after inserting $\text{Child}(abc, 4)$. Each extent is associated with the **size** of its largest attribute set generated so far. For illustration purpose, we skip steps of sprouting $(abc, 3')$, $(ae, 6)$, $(be, 1')$. (e) Generate $\text{Child}(ab, 3')$. (f) Note that the attribute set size of $(a, 6)$ as a child of $(abc, 4)$ is only 2, less than its size of 5 (after a is sprouted by $(ae, 6)$) in the trie, therefore $(a, 6)$ is not a successor of $(abc, 4)$. Similarly, $(b, 1')$ is not a successor of $(abc, 4)$.

$\text{ext}(E) \subseteq \text{ext}(D)\}$. It is not difficult to see that $\text{ExtChild}(C) = \{E : (E, \text{int}(E)) \in U_C\}$. Hence the total running time of the algorithm is $O(\sum_{C \in \mathcal{B}} \sum_{a \in \text{ext}(C)} |\{D \in U_{G(C)} : a \in \text{ext}(D)\}|)$, and it takes $O(\sum_{a \in \text{ext}(C)} |\{D \in U_{G(C)} : a \in \text{ext}(D)\}|)$ polynomial delay for processing concept $(C, \text{int}(C))$.

5 Discussion

In this paper, by making use of the lattice structure of concepts, we present a simple and fast algorithm for constructing a Galois lattice of a context. We generate all concepts, together with a list of successors of each concept, in the BFS order. Our algorithm is faster than the existing algorithms for constructing lattices [17, 21, 16, 14], and algorithms for computing only all concepts [6]. The algorithm by Makino and Uno [6] generates the concepts according to the (reverse search) enumeration tree order. As it does not seem to have a clear correspondence between the lattice and the enumeration tree, it is not clear that one can modify their algorithm to output the concepts together with its lattice order. In FCA, much of research has been devoted to the study of the structure properties of concept lattices. However, the lattice-construction algorithms in the area do not seem to take advantage of these properties.

In data mining, one is in general interested in concepts with large extent size, and the lattice structure. Since we generate our concepts in decreasing order of extent size, we can naturally modify our algorithm to output these large concepts and its lattice order, namely by keeping and sprouting only concepts with large enough extents. Theoretically, our algorithm is much faster than the current state-of-art algorithm CHARM [34] in data mining. We are testing and comparing our algorithm with CHARM on the benchmark dataset. The results will be reported elsewhere.

As FCA finds its wide applications in areas ranging from data mining to bioinformatics, efficient algorithms for constructing Galois lattices are much needed. Our algorithm is faster than previous algorithms for this problem, nevertheless, it seems to have much room to improve.

Acknowledgment

We would like to thank Reinhard Laubenbacher for introducing us FCA.

References

- [1] J. Abello, A. Pogel, L. Miller. Breadth first search graph partitions and concept lattices. *J. of Universal Computer Science*, 10(8), p934–954, 2004.
- [2] E. Boros, V. Gurvich, L. Khachiyan, K. Makino. On maximal frequent and minimal infrequent sets in binary matrices. *Annals of Mathematics and Artificial Intelligence*, 39, p211–221, 2003. (STACS 2002)
- [3] D. Eppstein. Arboricity and bipartite subgraph listing algorithm. *Information Processing Letters*, 51(4), p207–211, 1994.
- [4] T. Kashiwabara, S. Masuda, K. Nakajima, T. Fujisawa. Generation of maximal independent sets of a bipartite graph and maximum cliques of a circular-arc graph. *J. Algorithms*, 13, p161–174, 1992.
- [5] D.S. Johnson, M. Yannakakis, C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27, p119–123, 1988.
- [6] K. Makino, T. Uno. New algorithms for enumerating all maximal cliques. *Proc. 9th Scand. Workshop on Algorithm Theory (SWAT 2004)*, p260–272. Springer Verlag, Lecture Notes in Computer Science 3111, 2004.
- [7] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets *SIAM Journal on Computing*, 6(3), p505–517, 1977.
- [8] M. Barbut and B. Montjardet. *Ordre et Classifications: Algebre et combinatoire*. Hachette, 1970.
- [9] F. Baklouti, R.E. Grarvy. A fast and general algorithm for Galois lattices building. *J. of Symbolic Data Analysis*, 3(1), p19–31, 2005. www.icons.rodan.pl/publications/
- [10] A Formal Concept Analysis Homepage. <http://www.upriss.org.uk/fca/fca.html>
- [11] B. Ganter, G. Stumme, R. Wille (eds.). Formal Concept Analysis: Foundations and Applications. *Lecture Notes in Computer Science*, vol 3626, Springer, 2005.
- [12] B. Ganter, R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer Verlag, 1996 (Germany)

version), 1999 (English version).

- [13] S.O. Kuznetsov. Complexity of learning in context lattices from positive and negative examples. *Discrete Applied Mathematics*, 142, p111–125, 2004.
- [14] S.O. Kuznetsov and S.A. Obedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(23), p189–216, 2002.
- [15] S.O. Kuznetsov. On computing the size of a lattice and related decision problems. *Order*, 18, p313–321, 2001.
- [16] C. Lindig. Fast concept analysis. Gerhard Stumme ed. *Working with Conceptual Structures - Contributions to ICCS 2000*, Shaker Verlag, Aachen, Germany, 2000.
- [17] L. Nourine, O. Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71, p199–204, 1999.
- [18] L. Nourine, O. Raynaud. A fast incremental algorithm for building lattices. *J. of Experimental and Theoretical Artificial Intelligence*, 14, p217–227, 2002.
- [19] E.M. Norris. An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de mathematiques Pures et Appliquees*, 23(2), p243–250, 1978.
- [20] P. Valtchev, R. Missaoui, R. Godin, M. Meridji. Generating frequent itemsets incrementally: two novel approaches based on Galois lattice theory. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2-3), p115–142, 2002.
- [21] P. Valtchev, R. Missaoui, P. Lebrun. A partition-based approach towards constructing Galois (concept) lattices. *Discrete Mathematics*, 256(3), p801–829, 2002.
- [22] R. Wille. Restructuring the lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered sets*, pages 445–470, Dordrecht-Boston, 1982, Reidel.
- [23] F. Afrati, A. Gionis, H. Mannila. Approximating a collection of frequent sets. *Proc. 10th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, p12–19, 2004.
- [24] G. Cong, K.L. Tan, A.K.H. Tung, F. Pan. Mining frequent closed patterns in microarray data. *Proc. 4th IEEE International Conference on Data Mining*, p363–366, 2004.
- [25] T. Calders, C. Rigotti, J.F. Boulicaut. A survey on condensed representations for frequent sets. *Constrained-based Mining*, Springer, 3848, 2005.
- [26] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1), p25–46, 1999.
- [27] J. Pei, J. Han, R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, p21–30, 2000.
- [28] F. Rioult, J.F. Boulicaut, B. Cremilleux, J. Besson. Using transposition for pattern discovery from microarray data. *Proc. 8th ACM SIGMOD workshop on Research issues in Data Mining and Knowledge Discovery*, p73–79, 2003.
- [29] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, L. Lakhal. Fast Computation of Concept Lattices using data mining techniques. *Proc. 7th International Workshop on Knowledge Representation meets Databases*, p129–139, 2000.
- [30] J. Wang, J. Han, J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. *Proc. 9th ACM SIGKDD international conference on Knowledge Discovery and Data mining*, p236–245, 2003.
- [31] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. *Proc. 10th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, p344–353, 2004.
- [32] M.J. Zaki, M. Ogihara. Theoretical foundations of association rules. *Proc. 3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, p1–7, 1998.
- [33] M.J. Zaki, C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. *Proc. 2nd SIAM International Conference on Data Mining*, 2002.
- [34] M.J. Zaki, C.-J. Hsiao. Efficient Algorithms for mining closed itemsets and their lattice structure. *IEEE Trans. Knowledge and Data Engineering*, 17(4), p462–478, 2005.

Algorithm 2 Computing a Galois Lattice

Input: $(\mathcal{O}, \mathcal{M})$ and the adjacency list of each element;
Output: All concepts $(C, \text{int}(C))$ and a successor list for each concept.

- 1: Initialize a trie T for the object set \mathcal{O} ;
- 2: Start with a queue Q containing \mathcal{O} ; \triangleright Each element in Q consists of a pointer pointing to an extent in T .
- 3: $\text{int}(\mathcal{O}) = \emptyset$; $\triangleright (\mathcal{O}, \emptyset)$ is the “top” concept.
- 4: **while** Q is not empty **do**
- 5: $C = \text{dequeue}(Q)$;
- 6: $\text{Child}(C, \text{int}(C)) = \text{SPROUT}(C, \text{int}(C))$;
- 7: **for** each pair $(A, \text{pre_int}_C(A)) \in \text{Child}(C, \text{int}(C))$ **do**
- 8: **if** A does not exists in T **then**
- 9: **if** $\text{COMPLETEINTENT}(A, \text{pre_int}_C(A)) = \text{Yes}$ **then**
- 10: Add A to the successor list of C ;
- 11: Enqueue A into Q ;
- 12: **end if**
- 13: $\text{int}(A) = \text{int}(C) \cup \text{pre_int}_C(A)$.
- 14: Insert A into T ;
- 15: **else if** $(\text{int}(C) \cup \text{pre_int}_C(A) = \text{int}(A))$ **then**
- 16: Add A to the successor list of C ;
- 17: **if** A is not in Q **then**
- 18: Enqueue A into Q ;
- 19: **end if**
- 20: **end if**
- 21: **end for**
- 22: Output the concept $(C, \text{int}(C))$ and the successor list of C ;
- 23: **end while**
