Yet a Faster Algorithm for Building the Hasse Diagram of a Concept Lattice

Jaume Baixeries, Laszlo Szathmary, Petko Valtchev, and Robert Godin

Dépt. d'Informatique UQAM, C.P. 8888,
Succ. Centre-Ville, Montréal H3C 3P8, Canada
baixeries.jaume@uqam.ca, Szathmary.L@gmail.com, valtchev.petko@uqam.ca,
godin.robert@uqam.ca

Abstract. Formal concept analysis (FCA) is increasingly applied to data mining problems, essentially as a formal framework for mining reduced representations (bases) of target pattern families. Yet most of the FCA-based miners, closed pattern miners, would only extract the patterns themselves out of a dataset, whereas the generality order among patterns would be required for many bases. As a contribution to the topic of the (precedence) order computation on top of the set of closed patterns, we present a novel method that borrows its overall incremental approach from two algorithms in the literature. The claimed innovation consists of splitting the update of the precedence links into a large number of lower-cover list computations (as opposed to a single upper-cover list computation) that unfold simultaneously. The resulting method shows a good improvement with respect to its counterpart both on its theoretical complexity and on its practical performance. It is therefore a good starting point for the design of efficient and scalable precedence miners.

1 Introduction

Formal concept analysis (FCA) extracts knowledge from datasets represented as objects × attributes tables [4]. Concepts are key knowledge chunks that represent meaningful abstractions in the underlying domain. They are particularly useful when hierarchically ordered into the concept lattice as the structure can support various types of reasoning such as classification, clustering, implication discovery, etc. Yet the construction of the concept lattice is not a trivial problem, especially with large datasets. Indeed, the size of the lattice could grow exponentially with the number of data items, hence the need to design efficient algorithms for the task (see [3]). The existing ones may roughly be split into three categories with respect to the structure that is effectively output. Historically, the first algorithms looked at the set of all concepts [1] which was not provided with any particular structure. Later on, methods constructing the Hasse diagram of the lattice, i.e., the concept set plus the precedence relation of the lattice, have been designed [2,5]. Quite recently, the problem of ordering the set of concepts, i.e., extracting the set of precedence links out of them, has been tackled [17].

S. Ferré and S. Rudolph (Eds.): ICFCA 2009, LNAI 5548, pp. 162–177, 2009.

The question is of both theoretical and practical significance. Indeed, on the one hand, efficient algorithms for the computation of the concept set exist which, if appropriately completed, could yield a good overall method for lattice construction. In particular, such methods can be easily crafted from the various frequent closure miners abounding in the data mining literature [9,10,12]. Yet some miners, while very efficient, are particularly hard to adapt to the simultaneous computation of the Hasse diagram. We tend to see this fact as a sufficient motivation for the design of algorithms dedicated to the latter task. On the other hand, FCA is increasingly used within the data mining community as a formal framework for the numerous reduced representations of patterns and associations. Hence the interest for the construction of the iceberg concept lattice, i.e., the ordered structure of all frequent intents. Once again, the existing closure miners only output the intents and could therefore benefit from a complementary procedure deriving precedence out of the intent set. Yet to be efficient, the target procedure should not perform operations that depend on the size of the object set.

Our current study is motivated by the need for efficient computation of the precedence links among intents without looking on the respective extents. It is a follow-up to a previous work of the third author [17] and a response to [7]. Both algorithms basically work in the same way, i.e., they use the same intuition and very similar supporting data structures. The initial approach, which boils down to an incremental top-down construction of the lattice/iceberg diagram, has been revisited and substantially improved, both on its theoretical and practical aspects. In particular, given a concept, the basic test is on its being a lower cover of specific concepts already integrated in the diagram (instead of looking for its upper covers among previously processed concepts). Moreover, a deeper insight into the concept neighborhoods within the lattice (which builds upon a property from [18]) is exploited to speedup the computation of precedence links. The overall precedence link discovery process thus unfolds gradually: at each iteration the lower cover lists of concepts already in the diagram are tentatively updated.

Our method is a clear improvement of the state of the art as it outperforms the reference algorithms. Indeed, the new approach of precedence calculations results in a lower worst-case complexity (a multiplicative factor drops out). This is empirically confirmed by the results of an experimental study involving a straightforward implementation of the method and a large set of typical datasets used in the pattern mining literature.

All in all, the contributions of the paper are three-fold. First, a handy property of neighborhoods in the Hasse diagram is proven which can be used in the design of further algorithms targeting the precedence links among concepts. Next, a concrete method exploiting that property is devised whose complexity shows a substantial improvement with respect to the reference method. Finally, an extensive empirical study on the practical performances of the new method as opposed to its counterpart is reported.

The paper is organized as follows. After a short overview of the relevant FCA notions and notations, we recall the reference method from [17] (Section 2). The structural results behind our approach are provided next (Section 3), followed by the presentation of the new method called **iPred** (Section 4). Experimental evaluations are also provided (Section 5). Finally, conclusion and future work directions are given (Section 6).

2 Previous Work

2.1 Notation

We depart from the usual FCA notation, in which we have a formal context $\mathbb{K}(G, M, I)$ plus the associated concept lattice. In this paper we focus on the lattice formed by the set of intents of a formal context, let this set be $\mathcal{C} \subseteq \wp(M)$. The lattice is this set plus an order on that set, this is: $\mathcal{L} = \langle \mathcal{C}, \leq_{\mathcal{L}} \rangle$, where $\leq_{\mathcal{L}} \subseteq \mathcal{C} \times \mathcal{C}$.

Since we are just focusing in the intents of the concept lattice, we have that the bottom of the lattice is M whereas the top is $\emptyset$, if we assume that the formal context is reduced. The **precedence** relation $\preceq$ between $c, \tilde{c} \in \mathcal{C}$ is such that $c \preceq \tilde{c}$ if and only if $\tilde{c} \subseteq c$. This relation works inverted w.r.t. the inclusion relation because of the orientation of the intents in the concept lattice.

Given two elements $c, \tilde{c} \subseteq \mathcal{C}$, the relation $\prec$ is that of being the immediate predecessor, this is, if $c \prec \tilde{c}$ we say that c is the immediate predecessor of $\tilde{c}$.

Property 1. If $c \prec \tilde{c}$, then $|c| > |\tilde{c}|$.

We also define the following metrics on the lattice $\mathcal{L}$: $\omega(\mathcal{L})$ is the **width** of $\mathcal{L}$, whereas $d(\mathcal{L})$ is the **maximal degree** of all the elements in $\mathcal{L}$.

For our convenience, we flatten the set notation when dealing with sets. Therefore, $\{ a, b, c \}$ becomes abc and $\{ \{ a, b \}, \{ b, c \} \}$ becomes $\{ ab, bc \}$.

2.2 Methods Computing the Precedence Order

The historically first algorithm to compute the set of concepts of a context, **Next-Closure** [1], does not provide the precedence of the elements in that context. Later algorithms, such as those due to Godin [5] and to Bordat [2], compute both concepts and precedence yet the two tasks are interleaved hence difficult to separate. This does not make good candidates to complete a frequent closed itemset (FCI) miner out of them.

The first algorithm dedicated to the computing of the precedence was published in [8] as a distinct and separable part of a complete algorithm for the construction of a family of open sets and its semi-lattice. Yet the corresponding method does not qualify for an efficient precedence miner either. Indeed, the core operation which, in FCA terms, corresponds to the computing of the upper covers of a concept, boils down to intersecting the concept intent with all intents of object that are not in the concept extent. Performing an operation a

number of times that depends on the size of the object set in a context clearly hurts the scalability of a method and hence limits its data mining potential as in realistic settings the number of the objects is orders of magnitude higher than the number of attributes. Nothewortily, transposing the context matrix would not help here as the cost of the algoritm in [8] depends on both dimensions of the context. Thus, whenever one of these is huge, its computation performances will invariably suffer.

The method in [17] is, to the best of our knowledge, the first attempt to address the precedence computation problem with data mining concerns in mind. In fact, the method only considers the set of all (frequent) intents and organizes them into a graph representing the Hasse diagram of the (iceberg) lattice. To that end, the intents are processed sizewise while at each step, the current intent is integrated into the already constructed part of the lattice graph (an upper set thereof for that matters) by recognizing its upper covers among the vertices of that partial graph. More precisely, the target concepts are pinpointed among a larger set of candidates, themselves generated by intersecting the current concept intent with the intents of all the current minimal concepts of the partial graph (see next subsection). The method has been recently rediscovered (see [7]) yet this new version shows greatly improved practical performance.

The idea of computing the order among frequent closed itemsets (FCI, alias frequent concept intents) has made its way into the data mining literature. For instance, the **Charm-L** algorithm [13] tackles that composite problem and its performance is very satisfactory (see [16]). Yet **Charm-L**, like the aforementioned lattice algorithms mixes concept computing with precedence detection, hence it is not a good choice for a mere precedence miner to adjoin to an existing FCI miner.

Recently, an approach for computing the precedence link out of FCIs and frequent generators has been proposed (see [14]). The corresponding method, **Snow**, fully qualifies for the task of completing existing FCI miners in a generic way. This has been extensively argued on in [16]. Yet **Snow** comes with a price: the minimal generators of frequent intents must be known as well as their respective closures (among the frequent intents). Although this is often the case that FCI miners output frequent minimal generators as byproduct, such practice is not a must in the field, so an overhead for computing the generators and for associating them to their closures must be provided for.

In our current study, we push further the ideas from [17], i.e., computing the precedence incrementally by incorporating the current intent into the already constructed subgraph. The novelty is a completely reshuffled procedure for establishing the links among concepts. In what follows, we first present the original algorithm and then the new one. Their respective worst-case complexity functions are established to theoretically ground the claimed improvement: Here a whole factor from the original formula vanishes. This is experimentally confirmed by the results of our comparative study on the practical performances of both methods.

2.3 The BorderAlg Algorithm

We depart from the algorithms in [17] and in [7], which we generically call **BorderAlg**. In general terms, the approach of those algorithms is to find the upper cover of the elements that are in the lattice. In order to achieve that, the algorithm sorts all the elements in the lattice sizewise and then, it proceeds to process each element one by one. At a given point of the algorithm, the element that is to be processed is intersected with all the elements in the border. The **border** set is the set of maximal elements of the set of already processed elements. Those intersections form the **candidate** set of upper covers for the current element. It is clear that those elements in the candidate set exist in the lattice, because it is closed under intersection. However, not all of them may necessarily be immediate predecessors. In order to find those predecessors, the maximal elements of this candidate set must be found, which we call the **cover** set. This is the set of upper covers for the current element. The algorithm proceeds to add the connections between the current element and all the elements in the cover set, updates the border set and proceeds with the next element.

Intuitively, we can see that for each element, we are sure that all those elements in the lattice that are of size strictly smaller (Property 1), have been processed. Since an element can only be in the upper cover if the size is strictly minor, we know that all the elements that potentially are immediate predecessors have already been processed. Some of them are in the border set, and the rest will result from the intersection with all the elements in cover.

Input: $\mathcal{C} = \{ c_1, c_2, \ldots, c_l \}$
Output: $\mathcal{L} = \langle \mathcal{C}, \leq_{\mathcal{L}} \rangle$
1 Sort($\mathcal{C}$);
2 Border $\leftarrow \{ c_1 \}$;
3 **foreach** $i \in \{ 2, l \}$ **do**
4 $\quad Candidate \leftarrow \{ c_i \cap \tilde{c} \mid \tilde{c} \in Border \}$;
5 $\quad Cover \leftarrow Maxima(Candidate)$;
6 $\quad \leq_{\mathcal{L}} \leftarrow \leq_{\mathcal{L}} \cup \{ (c_i, \tilde{c}) \mid \tilde{c} \in Cover \}$;
7 $\quad Border \leftarrow (Border - Cover) \cup c_i$;
8 **end**

Algorithm 1. The **BorderAlg** algorithm

To illustrate the **BorderAlg** algorithm, we have a formal context with its associate concept lattice (where only the intents are shown) in Figure 1. Let us assume that this algorithm has sorted the sets in the concept lattice (line 1) in the following way:

$$\{ \emptyset, c, d, a, bc, cd, de, abc, bcd, ade, cde \}$$

and that, at a certain point, all the elements up to de have been processed. Therefore, we have that the lattice has been constructed up to the iceberg in Figure 2.

	a	b	c	d	e
1	X	X	X		
2		X	X	X	
3	X			X	X
4			X	X	X
5	X				
6			X		

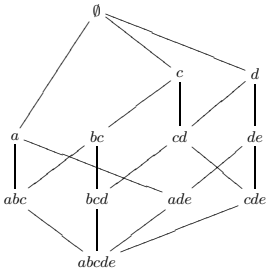

Fig. 1. Formal context and its concept lattice, in which only the intents are present

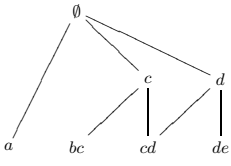

Fig. 2. Iceberg of the already processed elements

And that the next element to be processed is abc. The border set will therefore contain the elements a, bc, cd, de, and the candidate set computed in line 4 will yield the following set: $\{ a, bc, c, \emptyset \}$, which is the intersection of abc with the border set. As it can be seen, the candidate set contains the elements of the upper cover of abc (i.e. $\{ a, bc \}$) that are the immediate predecessors of abc, as well as other sets which are not in that upper cover: (i.e $\{ c, \emptyset \}$). Precisely, the function $Maxima$ in line 5 computes the $Cover$ set from the $Candidate$ set. The resulting set is $\{ a, bc \}$, and then, the connections (a, abc) and (bc, abc) are added to $\leq_{\mathcal{L}}$ in line 6, the sets a and bc are removed from the $Border$ set, and abc is added to that same set, so that the final $Border$ set after this iteration of the algorithm is cd, de, abc.

The complexity of **BorderAlg** is ([17]):

$$| \mathcal{C} | \times \omega(\mathcal{L}) \times |M|^2$$

More detailed information on the complexity analysis of **BorderAlg** can be found in Appendix A.

3 Theoretical Background of iPred

We have seen that the dominant factors in the complexity of Algorithm **BorderAlg** are the size of the input set $| \mathcal{C} |$ and the computation of the maximal elements of $Candidate$ (the set $Cover$) which is $\omega(\mathcal{L}) \times |M| \times d(\mathcal{L})$. Since the first

factor can't be avoided, the efforts to reduce the complexity of this algorithm should be directed towards the reduction of the computation of the *Cover* set. In this paper we present our new algorithm, called **iPred**, which improves the second factor by assuming a new strategy.

If we focus on the second factor of complexity in Algorithm 1, we see that for any given element c_i we test which elements of the *Candidate* set also belong to the *Cover* set. This computation is performed by choosing the maximal elements of the *Candidate* set, since the following condition is met:

Property 2. An element $\tilde{c}$ is in the *Candidate* set of an element $c_i \in \mathcal{C}$ if and only if $c_i \preceq \tilde{c}$.

That means that all potential elements that can be immediate predecessors of c_i are all in the *Candidate* set. Therefore, the algorithm only needs to choose among the *Candidate* set all those elements that are **immediate** predecessors of c_i, i.e:

$$\{\, \tilde{c} \in Candidate(c_i) \mid c_i \prec \tilde{c} \,\}$$

This is what the function $Maxima$ in line 5 of Algorithm 1 does, since it chooses the maximal elements of the *Candidate* set.

The algorithm **iPred** that we propose in this paper takes a different strategy: instead of checking if an element $\tilde{c}$ of the *Candidate* set is in the upper cover of the element c_i which is currently being processed, we check if the element c_i belongs to the lower cover of $\tilde{c}$. Although in principle, it may seem that both strategies should yield a similar (not to say the same) complexity, we prove in this paper that **iPred** improves in a factor of $d(\mathcal{L})$ the complexity of previous algorithms. Since $|M|$ is an upper bound of $d(\mathcal{L})$, the improvement, in the best of the cases, is that of $|M|$.

In this section we present the mathematical background on which **iPred** is based. We define an enumeration of $\mathcal{C}$:

Definition 1. *An **enumeration** of $\mathcal{C}$ is the set:*

$$\text{enum}(\mathcal{C}) = \{\, c_1, c_2, \ldots, c_n \,\}$$

such that $\forall i, j \leq n : i \leq j \implies |c_i| \leq |c_j|$.

An enumeration is simply a sizewise sorting of the elements of a set. We now define the **face** ([11]) and the **set of faces** of an element:

Definition 2. *The* face *of an element $c \in \mathcal{C}$ w.r.t. an immediate successor $\tilde{c}$ is the difference between those two sets. The **set of faces** is:*

$$faces(c) = \{\, \tilde{c} - c \mid \tilde{c} \prec c \,\}$$

For instance, according to the concept lattice in Figure 1, the face of a w.r.t. abc is bc, and $faces(a) = \{\, bc, de \,\}$. We now define a partial union of the faces of an element of the lattice:

Definition 3. *We define the* ***accumulation of faces*** *of an element* $c \in \mathcal{C}$ *w.r.t. an enumeration* $\text{enum}(\mathcal{C})$ *as:*

$$\Delta_c^i = \bigcup \{\, c_j - c \mid c_j \in \text{enum}(\mathcal{C}) \text{ and } c_j \prec c \text{ and } j < i \,\}$$

The accumulation of faces of $c \subseteq M$ up to i is simply the faces of c in the iceberg lattice formed by the elements of the enumeration $\text{enum}(\mathcal{C})$ up to i. Following Figure 1, we have that if the enumeration of the lattice is

$$\{\, \emptyset, c, d, a, bc, cd, de, abc, bcd, ade, cde \,\}$$

then, $\Delta_a^9 = bc$ and $\Delta_a^{11} = bcde$. This accumulation of faces is a handy way to test whether an element of the lattice is in the lower cover of another element of the lattice, as the next proposition shows:

Proposition 1. $c_i \prec c$ *if and only if* $c_i \cap \Delta_c^i = \emptyset$ *and* $c_i \preceq c$.

Proof. $\Rightarrow$ $c_i \cap \Delta_c^i = \emptyset$ and $c_i \preceq c$ implies that $c_i \prec c$. By the way of contradiction, let us assume that $c_i \not\prec c$. Since $c_i \preceq c$, there is a $\tilde{c}$ such that $c_i \preceq \tilde{c} \prec c$. Therefore, by Definition 1 of sequence, and by Definition 3 of accumulation of faces, $\tilde{c} - c \subseteq \Delta_c^i$, and since $c_i \preceq \tilde{c}$, then, $c_i \cap \Delta_c^i \neq \emptyset$, which is a contradiction.

$\Leftarrow$ $c_i \prec c$ implies that $c_i \cap \Delta_c^i = \emptyset$ and $c_i \preceq c$. If $c_i \prec c$, by Definition 1 of sequence, all the immediate predecessors of c of size smaller or equal than $|c_i|$ are incomparable with c_i, meaning that $c_i \cap \Delta_c^i = \emptyset$. □

This proposition basically states that, given an enumeration of the elements of a lattice, in order to know if between two elements $c, \tilde{c}$ of the lattice we have that $c \prec \tilde{c}$, we only need to test if the accumulation of faces of $\tilde{c}$ has an empty intersection with c.

In order to compute the connections in a lattice according to Proposition 1, the following must be performed:

1. Sort the elements of the lattice into an enumeration.
2. For each element in the lattice, the candidate set must be computed.
3. It must be checked if the element currently being processed belongs to the lower set of all the elements of the candidate set.

The first two steps are as in **BorderAlg**, and the difference appears in the third step. In order to test if the current element is in the lower cover of an element of the candidate set, we must compute the accumulation of faces of the latter. This step will reduce the complexity of **BorderAlg** in a factor of $|M|$ as we will see in the next section.

4 The iPred Algorithm

In this section we present our new algorithm called **iPred**, we examine its correctness and complexity, and we also provide a running example.

4.1 The Algorithm

The algorithm is based on Proposition 1, and it computes the sets *Border*, *Candidate* as in Algorithm 1, but the set *Cover* is not needed any more. There is an extra structure, Δ, in which we store the accumulation of faces for all the elements of the lattice, and we choose the notation $\Delta[c]$ to show the access to the accumulated faces of the set of attributes c. In terms of complexity, if this structure is implemented with a trie, the access to an element would be linear on the number of attributes: $|M|$. The **iPred** algorithm is as follows:

```
   Input: C = { c_1, c_2, ..., c_l }
   Output: L = ⟨C, ≤_L⟩
 1 Sort(C);
 2 foreach i ∈ { 2, l } do
 3 |   Δ[c_i] ← ∅;
 4 end
 5 Border ← { c_1 };
 6 foreach i ∈ { 2, l } do
 7 |   Candidate ← { c_i ∩ c̃ | c̃ ∈ Border };
 8 |   foreach c̃ ∈ Candidate do
 9 |   |   if Δ[c̃] ∩ c_i = ∅ then
10 |   |   |   ≤_L ← ≤_L ∪(c_i, c̃);
11 |   |   |   Δ[c̃] = Δ[c̃] ∪ (c_i − c̃);
12 |   |   |   Border ← Border − c̃;
13 |   |   end
14 |   end
15 |   Border ← Border ∪ c_i;
16 end
```

Algorithm 2. The **iPred** algorithm

The algorithm works as follows:

1. It sorts the elements of the lattice by size (line 1). This sequence is now an enumeration as in Definition 1.
2. All the $\Delta[c_i]$ in each element of the input set is initialized to the empty set. This $\Delta[c_i]$ will contain the accumulation of faces for each element (lines 2–4).
3. The first element in the border is the first element in the sequence (line 5).
4. All remaining elements in the input sequence are processed in the order in which they appear in the enumeration (lines 6–16).
5. The candidate set is computed by intersecting the current element c_i with all the elements in the border (line 7).
6. We check if the current element belongs to the upper set of the elements that are in the candidate set (lines 8–14). This is done by checking Proposition 1 (line 9).

7. If the test result is positive, by Proposition 1 we know that $c_i \prec \tilde{c}$, so we can add this connection to the output set (line 10), then we add that face to the set of accumulated faces of $\tilde{c}$ (line 11) and finally, we remove $\tilde{c}$ from the Border (line 12).
8. Before the next element is processed, we make sure that c_i is added to the border (line 15).

The correctness of this algorithm follows from the following facts:

1. $\mathcal{C}$ is a valid enumeration, according to Definition 1.
2. $\Delta[\tilde{c}]$ has at each step of the algorithm the accumulation of faces $\Delta^i_{\tilde{c}}$. At the beginning of the algorithm, for any element $\tilde{c}$ of the lattice, its accumulation is the empty set, this is $\Delta^0_{\tilde{c}}$. Every time a new element $\tilde{\tilde{c}}$ such that $\tilde{\tilde{c}} \prec \tilde{c}$ is found, $\Delta^i_{\tilde{c}}$ is updated conveniently in line 11, which guarantees that at the loop i of the algorithm, $\Delta[\tilde{c}] = \Delta^i_{\tilde{c}}$.
3. A connection is added if and only if the test in Proposition 1 is positive. It should be noted that since all the elements of the *Candidate* set are the intersection of c_i with the elements of the *Border* set, we are sure that, for each element $c \in Candidate$, $c \subseteq c_i$ (i.e. $c_i \preceq c$) holds, which is one of the conditions in Proposition 1.
4. *Border* always contains the maximal elements of the set of processed elements. This border is updated in line 12, where the element $\tilde{c}$ is removed. This is valid since we are sure that this can be done because c_i will be added to the border (line 15) and, at the same time, we know that $c_i \prec \tilde{c}$ since we are in the positive case of the test in line 9. If the test is negative, c_i is also added to the border, but no elements are removed.

Therefore, we conclude that Algorithm 2 finishes (since we assume that the set $|\mathcal{C}|$ is finite), and correctly computes the connections in the lattice set, since it correctly tests the condition in Proposition 1.

4.2 Complexity Analysis

The complexity of the previous algorithm is based on the following factors:

1. The sort in line 1 can be performed in linear time w.r.t. the size of the set, this is, $|\mathcal{C}| \times |M|$ (as in Algorithm 1). The cost of line 2 is exactly the same.
2. The loop in lines 6–16 is done $|\mathcal{C}|$ times (as in Algorithm 1).
3. The complexity of the computation of the candidate set in line 7 is $\omega(\mathcal{L}) \times |M|$ (as in Algorithm 1).
4. The loop in lines 8–14 is performed $\omega(\mathcal{L})$ times (as in Algorithm 1).
5. The cost of checking the condition in line 9 is $|M|$ if Δ is a trie.
6. The cost of line 10 is $|M|$ since it consists in adding a pair of size M to $\leq_{\mathcal{L}}$.
7. The cost of line 11 is $|M|$ because it consists in updating an element in a trie.
8. As for line 12, it consists in removing an element from a set. If this set is also implemented with a trie, then the cost is also $|M|$.

We condense the costs of lines 9, 10, 11 and 12 into $|M|$ and, therefore, the total cost of this algorithm is:

$$\underbrace{|\mathcal{C}| \times |M|}_{\text{line: } 1\text{–}2} + \underbrace{|\mathcal{C}|}_{6\text{–}16} \times \Big(\underbrace{\omega(\mathcal{L}) \times |M|}_{7} + \underbrace{\omega(\mathcal{L})}_{8\text{–}14} \times \underbrace{|M|}_{9\text{–}12} \Big)$$

Since the cost of line 7 ($\omega(\mathcal{L}) \times |M|$) subsumes the cost of lines 8–14 and 9–12 (it is just the same), and since the factor in lines 1–2 is subsumed by the rest of the formula, the complexity of Algorithm 2 is finally of order:

$$|\mathcal{C}| \times \omega(\mathcal{L}) \times |M|$$

Compared with the complexity of **BorderAlg**, we can see that the factor $|M|^2$ is now $|M|$, which means that we should expect an improvement of the performance of the algorithm by a factor linear on the size of the attribute set.

4.3 Running Example

Let us see how the algorithm would perform according to Figure 1. We list the following variables:

1. The element currently being processed (line 6).
2. The candidate set (line 7).
3. The output $\leq_{\mathcal{L}}$ (line 10).
4. The accumulation of faces, only for those that are changed (line 11).
5. The border (line 12).

At the beginning of the algorithm, the elements of the input set are sorted sizewise, let us assume that one of the possible orderings is:

$$\{\, \emptyset, c, d, a, bc, cd, de, abc, bcd, ade, cde \,\}$$

All the accumulation of faces are set to the empty set (lines 2 - 4) and Border has the first element of the sorted input sequence, this is $\emptyset$. We now list how the precedent sets change according to each loop in the algorithm.

1 Current element := c	7 Current element := abc
Candidate set := $\{\, \emptyset \,\}$	Candidate set := $\{\, \emptyset, a, c, bc \,\}$
$\leq_{\mathcal{L}}$: added $(\emptyset, c)$	$\leq_{\mathcal{L}}$: added $(a, abc), (bc, abc)$
$\Delta[\emptyset] = c$	$\Delta[a] := bc$, $\Delta[bc] = a$
Border := $\{\, c \,\}$	Border := $\{\, cd, de, abc \,\}$
2 Current element := d	8 Current element := bcd
Candidate set := $\{\, \emptyset \,\}$	Candidate set := $\{\, \emptyset, d, bc, cd \,\}$
$\leq_{\mathcal{L}}$: added $(\emptyset, d)$	$\leq_{\mathcal{L}}$: added $(bc, bcd), (cd, bcd)$
$\Delta[\emptyset] = cd$	$\Delta[bc] = ad$,$\Delta[cd] = b$
Border := $\{\, c, d \,\}$	Border := $\{\, de, abc, bcd \,\}$

3 Current element := a
Candidate set := $\{ \emptyset \}$
$\leq_{\mathcal{L}}$: added $(\emptyset, a)$
$\Delta[\emptyset] = acd$
Border := $\{ a, c, d \}$

4 Current element := bc
Candidate set := $\{ \emptyset, c \}$
$\leq_{\mathcal{L}}$: added (c, bc),
$\Delta[c] = b$
Border := $\{ a, bc, d \}$

5 Current element := cd
Candidate set := $\{ \emptyset, c, d \}$
$\leq_{\mathcal{L}}$: added $(c, cd), (d, cd)$
$\Delta[c] := bd$, $\Delta[d] := c$
Border := $\{ a, bc, cd \}$

6 Current element := $\{d, e\}$
Candidate set := $\{ \emptyset, d \}$
$\leq_{\mathcal{L}}$: added (d, de)
$\Delta[d] := ce$
Border := $\{ a, bc, cd, de \}$

9 Current element := ade
Candidate set := $\{ \emptyset, a, d, de \}$
$\leq_{\mathcal{L}}$: added $(a, ade), (de, ade)$
$\Delta[a] = bcde$,$\Delta[de] = a$
Border := $\{ abc, bcd, ade \}$

9 Current element := cde
Candidate set := $\{ \emptyset, c, cd, de \}$
$\leq_{\mathcal{L}}$: added $(cd, cde), (de, cde)$
$\Delta[cd] = be$,$\Delta[de] = ac$
Border := $\{ abc, bcd, ade, cde \}$

10 Current element := $abcde$
Candidate set := $\{ abc, bcd, ade, cde \}$
$\leq_{\mathcal{L}}$: added $(abc, abcde)$,$(bcd, abcde)$
$(ade, abcde)$,$(cde, abcde)$
$\Delta[abc] = de$,$\Delta[bcd] = ae$
$\Delta[ade] = bc$,$\Delta[cde] = ab$
Border := $\{ abcde \}$

As an example, let us see what happens in step 7. We add the element abc. The $Candidate$ set is $\{ \emptyset, a, c, bc \}$, which results from the intersection with the $Border$ set, which is $\{ a, bc, cd, de \}$. We point out the fact that the $Candidate$ set contains the upper set of abc in the lattice $\mathcal{L}$. The accumulated faces of the elements of the $Candidate$ set are then tested one by one with abc (line 9). The first intersection is with the accumulated face of $\emptyset$, which is acd according to its last update in loop 3. Since the intersection is not void, then, $\emptyset$ is not considered as an immediate predecessor of abc. As it has been previously explained, the reason is that the pairs $(\emptyset, a)$ and $(\emptyset, bc)$ have already been added to $\leq_{\mathcal{L}}$ and, therefore, the differences between those sets and $\emptyset$ is in $\Delta^7_\emptyset$. The next element to be checked is the accumulated faces of a, which is void and, hence, the intersection is also void. It means that $abc \prec a$, and the following operations are performed in lines 10–12: the pair (a, abc) is added to $\leq_{\mathcal{L}}$, and a is removed from the $Border$ set. The accumulated faces of a is updated accordingly, and we have now that $\Delta[a] := bc$. We now check the intersection of accumulated faces of the next element, this is element$\Delta[c]$, which is bd, and abc. The intersection is not void, and therefore, c is not a predecessor of abc. The reason is that the element bc has already been processed and $\Delta[c]$ contains, at least, the attribute b, meaning that c is the predecessor of an element that is contained by abc. The final checking is between $\Delta[bc]$, which is empty, and abc. The intersection is obviously void, and the algorithm adds (bc, abc) to $\leq_{\mathcal{L}}$, deletes bc from the border and updates $\Delta[bc]$ which now is a.

Table 1. Top: database characteristics. **Bottom:** response times of **iPred**.

database name	# records	# non-empty attributes	# attributes (in average)	largest attribute
T20I6D100K	100,000	893	20	1,000
T25I10D10K	10,000	929	25	1,000
chess	3,196	75	37	75
connect	67,557	129	43	129
pumsb	49,046	2,113	74	7,116
MUSHROOMS	8,416	119	23	128
C20D10K	10,000	192	20	385
C73D10K	10,000	1,592	73	2,177

min_supp	# concepts (including top)	**BorderAlg**	**iPred**
T20I6D100K			
0.75%	4,711	2.29	2.07
0.50%	26,209	74.92	51.88
0.25%	149,218	2,930.29	1,941.07
T25I10D10K			
0.40%	83,063	978.58	707.75
0.30%	122,582	2,207.86	1,763.42
0.20%	184,301	5,155.20	4,740.87
chess			
65%	49,241	974.23	87.60
60%	98,393	4,320.81	374.26
55%	192,864	21,550.23	1,905.60
connect			
65%	49,707	1,331.96	78.15
60%	68,350	2,634.35	140.50
55%	94,917	5,349.14	262.15

min_supp	# concepts (including top)	**BorderAlg**	**iPred**
pumsb			
84%	11,443	614.85	57.80
82%	19,942	2,043.31	173.31
80%	33,296	6,270.42	471.66
MUSHROOMS			
20%	1,169	0.71	0.17
10%	4,850	7.31	1.31
5%	12,789	53.35	7.87
C20D10K			
0.60%	119,734	10,847.59	993.29
0.50%	132,952	13,784.71	1,328.33
0.40%	151,394	19,013.53	1,858.34
C73D10K			
70%	19,501	414.94	37.29
65%	47,491	2,864.02	226.80
60%	108,428	18,323.78	1,296.40

5 Experimental Results

The original **BorderAlg** and the improved **iPred** algorithms were implemented in Java in the CORON data mining platform [15].[1] The experiments were carried out on a bi-processor Intel Quad Core Xeon 2.33 GHz machine with 4 GB RAM running under Ubuntu GNU/Linux. All times reported are real, wall clock times.

For the experiments, we used several real and synthetic dataset benchmarks. Database characteristics are shown in Table 1 (top). The chess and connect datasets are derived from their respective game steps. The MUSHROOMS database describes mushrooms characteristics. These three datasets can be found in the UC Irvine Machine Learning Database Repository. The pumsb, C20D10K, and C73D10K datasets contain census data from the PUMS sample file. The synthetic datasets T20I6D100K and T25I10D10K, using the IBM Almaden generator, are constructed according to the properties of market basket data.

Table 1 (bottom left and right) provides a summary of the experimental results. The first column specifies the various minimum support values for each of the datasets (low for the sparse dataset, higher for dense ones), while the second column comprises the number of FCIs. The third and fourth columns compare the execution times of **BorderAlg** and **iPred** (given in seconds). The CPU time does not include the cost of computing FCIs since it is assumed as given.

[1] `http://coron.loria.fr`

As can be seen, the improved algorithm outperforms the original **BorderAlg** algorithm in all cases. In the case of sparse datasets (T20 and T25), the difference is not that spectacular. However, in the case of dense datasets, there is a significant difference between the two algorithms, especially at lower minimum support thresholds. This is due to the fact that **iPred** reduces the complexity of **BorderAlg** by a factor linear on the size of the attribute set (see Section 4.2). The experimental results prove that the practical performance of **iPred** reflects its better theoretical complexity.

6 Conclusion

We presented a novel method for computing the precedence order among concepts that only manipulates their intents and is therefore suitable for data mining applications. The method explores the basic fact that the faces of all lower covers of a given concept in the Hasse diagram are pair-wise disjoint. Hence, the incremental incorporation of concepts into the current diagram could be organized as a set of gradually unfolding lower-cover list completions. A completion step is executed upon the incorporation of a new intent into the diagram and boils down to testing its disjointness with the already recognized faces.

The new method has been shown to outperform the reference one both on its worst-case complexity (smaller by a multiplicative factor) and practical performances (speedup from 30 to 3 000 %, depending on dataset profile). The advantages of the new method are only starting to unravel as no particular speedup techniques have been employed in the current implementation. Thus, the next step would be to study the benefits of indexing on the border set to avoid unnecessary intersections with the current intent.

Another promising track seems to reside in the batch processing of the levels in the diagram, i.e., the set of intents of identical size. Another intriguing question is the performance of a modern FCI miner, such as **Charm** or **Closet**, completed with our method.

References

1. Ganter, B.: Two basic algorithms in concept analysis (preprint). Technical Report 831, Technische Hochschule, Darmstadt (1984)
2. Bordat, J.-P.: Calcul pratique du treillis de Galois d'une correspondance. Mathématiques et Sciences Humaines 96, 31–47 (1986)
3. Carpineto, C., Romano, G.: Concept Data Analysis: Theory and Applications. John Wiley & Sons, Ltd., Chichester (2004)
4. Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer, Heidelberg (1999)
5. Godin, R., Missaoui, R.: An Incremental Concept Formation Approach for Learning from Databases. Theoretical Computer Science 133, 378–419 (1994)
6. Kryszkiewicz, M.: Concise Representation of Frequent Patterns Based on Disjunction-Free Generators. In: Proc. of the 2001 IEEE Intl. Conf. on Data Mining (ICDM 2001), Washington, DC, pp. 305–312. IEEE Computer Society Press, Los Alamitos (2001)

7. Martin, B., Eklund, P.W.: From Concepts to Concept Lattice: A Border Algorithm for Making Covers Explicit. In: Medina, R., Obiedkov, S. (eds.) ICFCA 2008. LNCS (LNAI), vol. 4933, pp. 78–89. Springer, Heidelberg (2008)
8. Nourine, L., Raynaud, O.: A fast algorithm for building lattices. Inf. Process. Lett. 71(5–6), 199–204 (1999)
9. Pei, J., Han, J., Mao, R.: CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 21–30 (2000)
10. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering Frequent Closed Itemsets for Association Rules. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 398–416. Springer, Heidelberg (1998)
11. Pfaltz, J.L.: Incremental Transformation of Lattices: A Key to Effective Knowledge Discovery. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 351–362. Springer, Heidelberg (2002)
12. Zaki, M.J., Hsiao, C.-J.: ChARM: An Efficient Algorithm for Closed Itemset Mining. In: SIAM Intl. Conf. on Data Mining (SDM 2002), pp. 33–43 (April 2002)
13. Zaki, M.J., Hsiao, C.-J.: Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure. IEEE Trans. on Knowl. and Data Eng. 17(4), 462–478 (2005)
14. Szathmary, L., Valtchev, P., Napoli, A., Godin, R.: Constructing Iceberg Lattices from Frequent Closures Using Generators. In: Boulicaut, J.-F., Berthold, M.R., Horváth, T. (eds.) DS 2008. LNCS (LNAI), vol. 5255, pp. 136–147. Springer, Heidelberg (2008)
15. Szathmary, L.: Symbolic Data Mining Methods with the Coron Platform. PhD Thesis in Computer Science, Univ. Henri Poincaré – Nancy 1, France (November 2006)
16. Szathmary, L., Valtchev, P., Napoli, A., Godin, R.: A Modular Approach for Mining Iceberg Lattices with Generators. In: Proc. of the 9th SIAM Intl. Conf. on Data Mining (SDM 2009) (submitted) (2009)
17. Valtchev, P., Missaoui, R., Lebrun, P.: A Fast Algorithm for Building the Hasse Diagram of a Galois Lattice. In: Proc. of Colloque LaCIM 2000, Montreal, Canada, pp. 293–306 (2000)
18. Valtchev, P., Hacene, M.R., Missaoui, R.: A Generic Scheme for the Design of Efficient On-Line Algorithms for Lattices. In: Ganter, B., de Moor, A., Lex, W. (eds.) ICCS 2003. LNCS, vol. 2746, pp. 282–295. Springer, Heidelberg (2003)

A Complexity of the Border Algorithm

We analyze now the complexity of Algorithm 1, which is based on the following costs:

1. The sizewise sorting of the input set (line 1). This can be done in linear time w.r.t. the size of the set: $|\mathcal{C}| \times |M|$, since we can first scan the list and compute the amount of elements for each size, allocate the corresponding slot memory, and in a second scan, we can store each element according to its size.
2. The number of loops in lines 3-8 depends on the amount of elements to be processed, which is the size of the input set: $|\mathcal{C}|$.

3. The cost of computing the *Candidate* set depends on the size of the border, since each element of $\mathcal{L}$ will be intersected with that set (line 4). This size is, in the worst of the cases, the width of the lattice, and each intersection, in the worst case, can be done in time linear on $|M|$. The total cost for computing the *Candidate* set is $\omega(\mathcal{L}) \times |M|$.
4. The computation of the maxima of the set resulting from the intersection, this is, the *Cover* set in line 5,$\Delta[bc] = a$ is performed in $|M| \times \omega(\mathcal{L}) \times d(\mathcal{L})$. Details of this function and its cost can be found in [17].
5. The update of $\leq_{\mathcal{L}}$ with the new connections in line 6 depends on the maximal number of connections that any element may have, which is the maximal degree: $d(\mathcal{L})$.
6. The update of the border in line 7 can be neglected since it only consists in adding a pair in a set.

The total complexity of this algorithm is, therefore:

$$\text{line: } \underbrace{|\mathcal{C}| \times |M|}_{1} + \underbrace{|\mathcal{C}|}_{3\text{-}8} \times \Big(\underbrace{\omega(\mathcal{L}) \times |M|}_{4} + \underbrace{|M| \times \omega(\mathcal{L}) \times d(\mathcal{L})}_{5} + \underbrace{d(\mathcal{L})}_{6} \Big)$$

Some factors are subsumed by others: the cost of computing the *Candidate* set $\omega(\mathcal{L}) \times |M|$ in line 4 and the cost $d(\mathcal{L})$ of updating $\leq_{\mathcal{L}}$ in line 6 are both subsumed by the factor $|M| \times \omega(\mathcal{L}) \times d(\mathcal{L})$ in line 5, and the additive factor $|\mathcal{C}| \times |M|$ in line 1 is also subsumed by the rest of the formula. Therefore, we have that the final complexity is:

$$|\mathcal{C}| \times \omega(\mathcal{L}) \times |M| \times d(\mathcal{L})$$

Since in the worst of the cases we have that $d(\mathcal{L}) = |M|$, then, we have:

$$|\mathcal{C}| \times \omega(\mathcal{L}) \times |M|^2.$$