


# Border Algorithms for Computing Hasse Diagrams of Arbitrary Lattices\*

View metadata, citation and similar papers at [core.ac.uk](http://core.ac.uk)

brought to you by  CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Departamento de Matemáticas, Estadística y Computación  
Universidad de Cantabria  
Santander, Spain  
{jose-luis.balcazar,cristina.tirnauca}@unican.es

**Abstract.** The Border algorithm and the iPred algorithm find the Hasse diagrams of FCA lattices. We show that they can be generalized to arbitrary lattices. In the case of iPred, this requires the identification of a join-semilattice homomorphism into a distributive lattice.

**Keywords:** Lattices, Hasse diagrams, border algorithms.

## 1 Introduction

Lattices are mathematical structures with many applications in computer science; among these, we are interested in fields like data mining, machine learning, or knowledge discovery in databases. One well-established use of lattice theory is in formal concept analysis (FCA) [8], where the concept lattice with its diagram graph allows the visualization and summarization of data in a more concise representation. In the Data Mining community, the same mathematical notions (often under additional “frequency” constraints that bound from below the size of the support set) are studied under the banner of Closed-Set Mining (see e.g. [21]).

In these applications, each dataset consists of *transactions*, also called *objects*, each of which, besides having received a unique identifier, consists of a set of *items* or *attributes* taken from a previously agreed finite set. A *concept* is a pair formed by a set of transactions —the *extent* set or *support set* of the concept— and a set of attributes —the *intent* set of the concept— defined as the set of all those attributes that are shared by all the transactions present in the extent. Some data analysis processes are based on the family of all intents (the “closures” stemming from the dataset); but others require to determine also their order relation, which is a finite lattice, in the form of a line graph (the *Hasse diagram*).

Existing algorithms can be divided into three main types: the ones that only generate the set of concepts, the ones that first generate the set of concepts

---

\* This work has been partially supported by project FORMALISM (TIN2007-66523) of Programa Nacional de Investigación, Ministerio de Ciencia e Innovación (MICINN), Spain, by the Juan de la Cierva contract JCI-2009-04626 of the same ministry, and by the Pascal-2 Network of the European Union.

and then construct the Hasse diagram, and the ones that construct the diagram while computing the lattice elements (see [21], and also [9,12] and the references therein). The goal is to obtain the concept lattice in linear time in the number of concepts because this number is, most of the times, already exponential in the number of attributes, making the task of getting polynomial algorithms in the size of the input rather impossible.

One widespread use of concepts or closures is the generation of implications or of partial implications (also called association rules). Several data mining algorithms aim at processing large datasets in time linear in the size of the closure space, and explore closed sets individually; these solutions tend to drown the user under a deluge of partial implications. More sophisticated works attempt at providing selected “bases” of partial implications; the early proposal in [13] requires to compute immediate predecessors, that is, the Hasse diagram. Alternative proposals such as the Essential Rules of [1] or the equivalent Representative Rules of [11] (of which a detailed discussion with new characterizations and an alternative basis proposal appears in [6]) require to process predecessors of closed sets obeying tightly certain support inequalities; these algorithms also benefit from the Hasse diagram, as the slow alternatives are blind repeated traversal of the closed sets in time quadratic in the size of the closure space, or storage of all predecessors of each closed set, which soon becomes large enough to impose a considerable penalty on the running times.

The problem of constructing the Hasse diagram of an arbitrary finite lattice is less studied. One algorithm that has a better worst case complexity than various previous works is described in [16]. From our “arbitrary lattices” perspective, its main drawback is that it requires the availability of a *basis* from which each element of the lattice can be derived. In the absence of such a subset, one may still use this algorithm (at a greater computational cost) to output the Dedekind-MacNeille completion [7] of the given lattice, which in our case is isomorphic to the lattice itself. The algorithm is also easily adaptable to concept lattices, where indeed a basis is available immediately from the dataset transactions.

We consider of interest to have available further, faster algorithms for arbitrary finite lattices; we have two reasons for this aim. First, many (although not all) algorithms constructing Hasse diagrams traverse concepts in layers defined by the size of the intents; our explorations about association rules sometimes require to follow different orderings, so that a more abstract approach is helpful; second, we keep in mind the application area corresponding to certain variants of implications and database dependencies that are characterized by lattices of equivalence relations, so that we are interested in laying a strong foundation that gives us a clear picture of the applicability requirements for each algorithm constructing Hasse diagrams in lattices other than powerset sublattices.

Of course, we expect that FCA-oriented algorithms could be a good source of inspiration for the design of algorithms applicable in the general case. An example that such an extension can be done is the algorithm in [20] (see Section 3 for more details), whose highest-level description matches the general case of arbitrary lattices; nevertheless, the actual implementation described in [20]

works strictly for formal concept lattices, so that further implementations and complexity analyses are not readily available for arbitrary finite lattices.

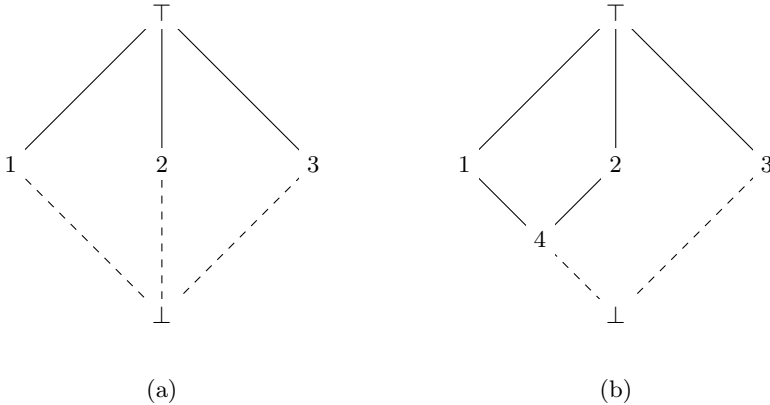
The contribution of the present paper supports the same idea: we show how two existing algorithms that build the Hasse diagrams of a concept lattice can be adapted to work for arbitrary lattices. Both algorithms have in common the notion of *border*, which we (re-)define and formalize in Section 3, after presenting some preliminary notions about lattice theory in Section 2; our approach has the specific interest that the notion of border is given just in terms of the ordering relation, and not in terms of a set of elements already processed as in previous references ([5,14,20]); yet, the notions are equivalent. We state and prove properties of borders and describe the *Generalized Border Algorithm*; whereas the algorithm reads, in high level, exactly as in previous references, its validation is new, as previous ones depended on the lattice being an FCA lattice. In Section 4 we introduce the *Generalized iPred Algorithm*, exporting the iPred algorithm of FCA lattices [5] to arbitrary lattices, after arguing its correctness. This task is far from trivial and is our major contribution, since the existing rendering and validation of the iPred algorithm relies again extensively on the fact that it is being applied to an FCA lattice, and even performs operations on difference sets that may not belong to the closure space. Concluding remarks and future work ideas are presented in Section 5.

## 2 Preliminaries

We develop all our work in terms of lattices and semilattices; see [7] as main source. All our structures are *finite*. A *lattice* is a partially ordered set in which every nonempty subset has a meet (greatest lower bound) and a join (lowest upper bound). If only one of these two operations is guaranteed to be available a priori, we speak of a *join-semilattice* or a *meet-semilattice* as convenient. Top and bottom elements are denoted  $\top$  and  $\perp$ , respectively. Lower case letters, possibly with primes, and taken usually from the end of the latin alphabet denote lattice elements:  $x, y'$ . Note that Galois connections are not explicitly present in this paper, so that the “prime” notation does not refer to the operations of Galois connections.

Finite semilattices can be extended into lattices by addition of at most one further element [7]; for instance, if  $(\mathcal{L}, \leq, \vee)$  is a join-semilattice with bottom element  $\perp$ , one can define a meet operation as follows:  $\bigwedge X = \bigvee \{y \mid \forall x \in X, y \leq x\}$ ; the element  $\perp$  ensures that this set is nonempty. Thus, if the join-semilattice lacks a bottom element, it suffices to add an “artificial” one to obtain a lattice. A dual process is obviously possible in meet-semilattices.

Given two join-semilattices  $(S, \vee)$  and  $(T, \vee)$ , a *homomorphism* is a function  $f : S \rightarrow T$  such that  $f(x \vee y) = f(x) \vee f(y)$ . Hence  $f$  is just a homomorphism of the two semigroups associated with the two semilattices. If  $S$  and  $T$  both include a bottom element  $\perp$ , then  $f$  should also be a monoid homomorphism, i.e. we additionally require that  $f(\perp) = \perp$ . Homomorphisms of meet-semilattices and of lattices are defined similarly. It is easy to check that  $x \leq y \Rightarrow f(x) \leq f(y)$  for



**Fig. 1.** Two join-semilattices converted into lattices

any homomorphism  $f$ ; the converse implication, thus the equivalence  $x \leq y \Leftrightarrow f(x) \leq f(y)$ , is also true for injective  $f$  but not guaranteed in general.

We must point out here a simple but crucial fact that plays a role in our later developments: given a homomorphism  $f$  between two join-semilattices  $S$  and  $T$ , if we extend both into lattices as just indicated, then  $f$  is *not* necessarily a lattice homomorphism; for instance, there could be elements of  $T$  that do not belong to the image set of  $f$ , and they may become meets of subsets of  $T$  in a way that prevents them to be the image of the corresponding meet of  $S$ . For one specific example, see Figure 1: consider the two join-semilattices defined by the solid lines, where the numbering defines an injective homomorphism from the join-semilattice in (a) to the join-semilattice in (b). Both lack a bottom element. Upon adding it, as indicated by the broken lines, in lattice (a) the meets of 1 and 2 and of 1 and 3 coincide, but the meets of their corresponding images in (b) do not; for this reason, the homomorphism cannot be extended to the whole lattices.

However, the following does hold:

**Lemma 1.** *Consider two join-semilattices  $S$  and  $T$ , and let  $f : S \rightarrow T$  be a homomorphism. After extending both semilattices into lattices,  $f(\bigwedge Y) \leq \bigwedge f(Y)$  for all  $Y \subseteq S$ .*

This is immediate to see by considering that  $\bigwedge Y \leq y$  for all  $y \in Y$ , hence  $f(\bigwedge Y) \leq f(y)$  for all such  $y$ , and the claimed inequality follows.

We employ  $x < y$  as the usual shorthand:  $x \leq y$  and  $x \neq y$ . We denote as  $x \prec y$  the fact that  $x$  is an immediate predecessor of  $y$  in  $\mathcal{L}$ , that is,  $x < y$  and, for all  $z$ ,  $x < z \leq y$  implies  $z = y$  (equivalently,  $x \leq z < y$  implies  $x = z$ ).

We focus on algorithms that have access to an underlying finite lattice  $\mathcal{L}$  of size  $|\mathcal{L}| = n$ , with ordering denoted  $\leq$ ; abusing language slightly, we denote by  $\mathcal{L}$  as well its carrier set. The *width*  $w(\mathcal{L})$  of the lattice  $\mathcal{L}$  is the maximum size of an antichain (a subset of  $\mathcal{L}$  formed by pairwise incomparable elements). The lattice is assumed to be available for our algorithms in the form of an abstract

data type offering an iterator that traverses all the elements of the carrier set, together with the operations of testing for the ordering (given  $x, y \in \mathcal{L}$ , find out whether  $x \leq y$ ) and computing the meet  $x \wedge y$  and join  $x \vee y$  of  $x, y \in \mathcal{L}$ ; also the constants  $\top \in \mathcal{L}$  and  $\perp \in \mathcal{L}$  are assumed available.

The algorithms we consider are to perform the task of constructing explicitly the Hasse diagram (also known as the reflexive and transitive reduction) of the given lattice:  $H(\mathcal{L}) = \{(x, y) \mid x \prec y\}$ . By projecting the Hasse diagram along the first or the second component we find our crucial ingredients: the well-known upper and lower covers.

**Definition 1.** *The upper cover of  $x \in \mathcal{L}$  is  $\text{uc}(x) = \{y \mid x \prec y\}$ . The lower cover of  $y \in \mathcal{L}$  is  $\text{lc}(y) = \{x \mid x \prec y\}$ .*

The following immediate fact is stated separately just for purposes of easy later reference:

**Proposition 1.** *If  $x < y$  then there is  $z \in \text{uc}(x)$  such that  $x \prec z \leq y$ ; and there is  $z' \in \text{lc}(y)$  such that  $x \leq z' \prec y$ .*

We will use as well yet another easy technicality:

**Lemma 2.** *If  $x_1 \prec y$  and  $x_2 \prec y$ , with  $x_1 \neq x_2$  then  $x_1 \vee x_2 = y$ .*

*Proof.* Since  $y \geq x_1$  and  $y \geq x_2$  we have  $y \geq x_1 \vee x_2$ . Then,  $x_1 \neq x_2$  implies that they are mutually incomparable, since otherwise the smallest is not an immediate predecessor of  $y$ ; this implies that  $y \geq x_1 \vee x_2 > x_1$ , whence  $y = x_1 \vee x_2$  as  $x_1 \prec y$ .  $\square$

### 3 The Border Algorithm in Lattices

The algorithms we are considering here have in common the fact that they traverse the lattice and explicitly maintain a subset of the elements seen so far: those that still might be used to identify new Hasse edges. This subset is known as the “border” and, as it evolves during the traversal, actually each element  $x \in \mathcal{L}$  “gets its own border” associated as the algorithm reaches it. The border associated to an element may be potentially used to construct new edges touching it (although these edges may not touch the border elements themselves): more precisely, operations on the border for  $x$  will result in  $\text{uc}(x)$ , hence in the Hasse edges of the form  $(x, z)$ .

In previous references the border is defined in terms of the elements already processed, and its properties are mixed with those of the algorithm that uses it. Instead, we study axiomatically the properties of the notion of “border” on itself, always as a function of the element for which the border will be considered as a source of Hasse edges, in a manner that is independent of the fact that one is traversing the lattice. This allows us to clarify which abstract properties are necessary for border-based algorithms, so that we can generalize them to arbitrary lattices, traversed in flexible ways. Our key definition is, therefore:

**Definition 2.** Given  $x \in \mathcal{L}$  and  $B \subseteq \mathcal{L}$ ,  $B$  is a border for  $x$  if the following properties hold:

1.  $\forall y \in B (y \not\leq x)$ ;
2.  $\forall z (x \prec z \Rightarrow \exists y \in B (y \leq z))$ .

That is,  $x$  is never above an element of a border, but each upper cover of  $x$  is; this last condition is equivalent to: all elements strictly above  $x$  are greater than or equal to some element of the border. Since  $x \leq (x \vee y)$  always holds and  $x = (x \vee y)$  if and only if  $y \leq x$ , we get:

**Lemma 3.** Let  $B$  be a border for  $x$ . Then  $\forall y \in B (x < x \vee y)$ .

All our borders will fulfill an extra “antichain” condition; the only use to be made of this fact is to bound the size of every border by the width of the lattice.

**Definition 3.** A border  $B$  is proper if every two different elements of  $B$  are mutually incomparable.

The key property of borders, that shows how to extract Hasse edges from them, is the following:

**Theorem 1.** Let  $B$  be a border for  $x_0$ . For all  $x_1$  with  $x_0 < x_1$ , the following are equivalent:

1.  $x_1 \in \text{uc}(x_0)$  (that is,  $x_0 \prec x_1$ );
2. there is  $y \in B$  such that  $x_1 = (x_0 \vee y)$  and, for all  $z \in B$ , if  $(x_0 \vee z) \leq (x_0 \vee y)$  then  $(x_0 \vee z) = (x_0 \vee y)$ .

*Proof.* Given  $x_0 \prec x_1$ , we can apply the second condition in the definition of border for  $x_0$ :  $\exists y \in B (y \leq x_1)$ . Using Lemma 3,  $x_0 < (x_0 \vee y) \leq x_1$ , implying  $(x_0 \vee y) = x_1$  since  $x_0 \prec x_1$ . Additionally, assuming  $(x_0 \vee z) \leq (x_0 \vee y)$  for some  $z \in B$  leads likewise to  $x_0 < (x_0 \vee z) \leq (x_0 \vee y) = x_1$  and the same property applies to obtain  $(x_0 \vee z) = (x_0 \vee y) = x_1$ .

Conversely, again Lemma 3 gives  $x_0 < (x_0 \vee y) = x_1$ . By Proposition 1, there is  $z_0 \in \text{uc}(x_0)$  with  $x_0 \prec z_0 \leq (x_0 \vee y) = x_1$ . We apply the second condition of borders to  $x_0 \prec z_0$  to obtain  $z_1 \in B$  with  $z_1 \leq z_0$ , whence  $(x_0 \vee z_1) \leq z_0 \leq (x_0 \vee y) = x_1$ , allowing us to apply the hypothesis of this direction:  $(x_0 \vee z_1) \leq (x_0 \vee y)$  with  $z_1 \in B$  implies  $(x_0 \vee z_1) = (x_0 \vee y)$  and, therefore,  $(x_0 \vee z_1) = z_0 = (x_0 \vee y) = x_1$ . That is,  $x_1 = z_0 \in \text{uc}(x_0)$ .  $\square$

Therefore, given an arbitrary element  $x_0$  of the lattice, any candidate for being an element of its upper cover has to be obtainable as a join between  $x_0$  and a border element ( $x_1 = x_0 \vee y$  for some  $y \in B$ ). Moreover, among these candidates, only those that are minimals represent immediate successors: they come from those  $y$  where  $(x_0 \vee z) \leq (x_0 \vee y)$  implies  $(x_0 \vee z) = (x_0 \vee y)$ , for all  $z \in B$ .

### 3.1 Advancing Borders

There is a naturally intuitive operation on borders; if we have a border  $B$  for  $x$ , and we use it to compute the upper cover of  $x$ , then we do not need  $B$  as such anymore; to update it, seeing that we no longer need to forbid the membership of  $x$ , it is natural to consider adding  $x$  to the border. If we had a proper border, and we wished to preserve the antichain property, the elements to be removed would be exactly the upper cover just computed, as these are, as we argue below, the only elements comparable to  $x$  that could be in a proper border. (All elements other than  $x$  are mutually incomparable, as the border was proper to start with.)

**Definition 4.** *Given  $x \in \mathcal{L}$  and a border  $B$  for  $x$ , the standard step for  $B$  and  $x$  is  $B \cup \{x\} - \text{uc}(x)$ .*

Note that this is *not* to say that  $\text{uc}(x) \subseteq B$ ; elements of  $\text{uc}(x)$  may or may not appear in  $B$ . We will apply the standard step always when  $B$  is a border for  $x$ , but let us point out that the definition would be also valid without this constraint, as it consists of just some set-theoretic operations.

**Proposition 2.** *Let  $B$  be a proper border for  $x$ . Then the standard step for  $B$  and  $x$  is also an antichain.*

*Proof.* Elements of the standard step different from  $x$  and from all elements of  $\text{uc}(x)$  were already in the previous proper border and are, therefore, mutually incomparable. None of them is below  $x$ , by the first border property. If  $y > x$  for some  $y \in B$ , then  $y \geq z \succ x$  for some  $z \in B$ , and the antichain property of  $B$  tells us that  $y = z$  so that it gets removed with  $\text{uc}(x)$ .  $\square$

However, we are left with the problem that we have now a candidate border but we lack the lattice element for which it is intended to be a border. In [14] and [5], the algorithm moves on to an intent set of the same cardinality as  $x$ , whenever possible, and to as small as possible a larger intent set if all intents of the same cardinality are exhausted. In [20] it is shown that, for their variant of the Border algorithm, it suffices to follow a (reversed) linear embedding of the lattice. Here we follow this more flexible approach, which is easier now that we have stated the necessary properties of borders with no reference to the order of traversal: there is no need of considering intent sets and their cardinalities.

Both lattices and their Hasse diagrams can be seen as directed acyclic graphs, by orienting the inequalities in either direction; here we choose to visualize edges  $(x, y)$  as corresponding to  $x \leq y$ . A linear embedding corresponds to the well-known operation of topological sort of directed acyclic graphs, which we will employ for lattices in a “reversed” way:

**Definition 5.** *A reverse topological sort of  $\mathcal{L}$  is a total ordering  $x_1, \dots, x_n$  of  $\mathcal{L}$  such that  $x_i \leq x_j$  always implies  $j \leq i$ .*

All our development could be performed with a standard topological sort, not reversed, that is, a linear embedding of the lattice’s partial order. However,

as it is customary in FCA to guide the visualization through the comparison of extents, the algorithms we build on were developed with a sort of “built-in reversal” that we inherit through reversing the topological sort (see the similar discussion in Section 2.1 of [5]). A reversed topological sort must start with  $\top$ , hence the initialization is easy:

**Proposition 3.**  $B = \emptyset$  is a border for  $\top \in \mathcal{L}$ .

*Proof.* Both conditions in the definition of border become vacuously true: the first one as  $B = \emptyset$  and the second one as the top element has no upper covers.  $\square$

**Theorem 2.** Let  $x_1, \dots, x_n$  be a reverse topological sort of  $\mathcal{L}$ . Starting with  $B_1 = \emptyset$ , define inductively  $B_{k+1}$  as the standard step for  $B_k$  and  $x_k$ . Then, for each  $k$ ,  $B_k$  is a border for  $x_k$ .

For clarity, we factor off the proof of the following inductive technical fact, where we use the same notation as in the previous statement.

**Lemma 4.**  $B_k \subseteq \{x_1, \dots, x_{k-1}\}$  and, for all  $x_j$  with  $j < k$ , there is  $y \in B_k$  with  $y \leq x_j$ .

*Proof.* For  $k = 1$ , the statements are vacuously true. Assume it true for  $k$ , and consider  $B_{k+1} = B_k \cup \{x_k\} - \text{uc}(x_k)$ , the standard step for  $B_k$  and  $x_k$ . The first statement is clearly true. For the second,  $x_k$  is itself in  $B_{k+1}$  and, for the rest, inductively, there is  $y \in B_k$  with  $y \leq x_j$ . We consider two cases; if  $y \notin \text{uc}(x_k)$ , then the same  $y$  remains in  $B_{k+1}$ ; otherwise,  $x_k \prec y \leq x_j$ , and  $x_k$  is the corresponding new  $y$  in  $B_{k+1}$ .  $\square$

*Proof (of Theorem 2).* Again by induction on  $k$ ; we see that the basis is Proposition 3. Assuming that  $B_k$  is a border for  $x_k$ , we consider  $B_{k+1} = B_k \cup \{x_k\} - \text{uc}(x_k)$ . Applying the lemma,  $B_{k+1} \subseteq \{x_1, \dots, x_k\}$ , which ensures immediately that  $\forall y \in B_{k+1} (y \not\leq x_{k+1})$  by the property of the reverse topological sort, and the first condition of borders follows. For the second, pick any  $z \in \text{uc}(x_{k+1})$ ; by the condition of reverse topological sort,  $z$ , being a strictly larger element than  $x_{k+1}$ , must appear earlier than it, so that  $z = x_j$  with  $j < k + 1$ . Then, again the lemma tells us immediately that there is  $y \in B_{k+1}$  with  $y \leq x_j = z$ , as we need to complete the proof.  $\square$

### 3.2 The Generalized Border Algorithm

The algorithm we end up validating through our theorems has almost the same high-level description as the rendering in [5]; the most conspicuous differences are: first, that a reverse topological sort is used to initialize the traversal of the lattice; and, second, that the “reversed lattice” model in [5] has the consequence that their set-theoretic intersection in computing candidates becomes a lattice join in our generalization. Another minor difference is that Proposition 3 spares us the separate handling of the first element of the lattice.



```

RevTopSort( $\mathcal{L}$ );
 $B = \emptyset$ ;
 $H = \emptyset$ ;
for  $x$  in  $\mathcal{L}$ , according to the sort do
  | candidates =  $\{x \vee y \mid y \in B\}$ ;
  | cover = minimals(candidates);
  | for  $z$  in cover do add  $(x, z)$  to  $H$ ;
  |  $B = B \cup \{x\} - \text{cover}$ ;
end

```

**Algorithm 1.** The Generalized Border Algorithm

Theorem 2 and Proposition 3 tell us that the following invariant is maintained:  $B$  is a border for  $x$ . Then, the Hasse edges are computed and added to  $H$  according to Theorem 1, in two steps: first, we prepare the list of joins  $x \vee y$  and, then, we keep only the minimal elements in it. In essence, this process is the same as described (in somewhat different renderings) in [5], [14] or [20]; however, while the definition of border given in [20] (and recalled in [5]) leads, eventually, to the same notion employed in this paper, further development of a general algorithm that works outside the formal concept analysis framework is dropped off from [20] on efficiency considerations. Moreover, the border algorithm described in [14] works exclusively on the set of intents and assumes the elements are sorted sizewise. The validations of the algorithms in these references rely very much, at some points, on the fact that the lattice is a sublattice of a powerset and contains formal concepts, explicitly operating set-theoretically on their intents. Theorem 1 captures the essence of the notion of border and lifts the algorithm to arbitrary lattices.

One additional difference comes from the fact that the cost of computing the meet and join operations plays a role in the complexity analysis, but is not available in the general case. If we assume that meet and join operations take constant time, then the total running time of the algorithm (except for the sort initialization, which takes  $\mathcal{O}(|\mathcal{L}| \log |\mathcal{L}|)$ ) is bounded by  $\mathcal{O}(|\mathcal{L}|w(\mathcal{L})^2)$ . By comparison with [20], one can see that one factor of the formula given in [20] gets dropped under the constant time assumption for computing meet and join. However, this assumption may be unreasonable in certain applications; the same reference indicates that their FCA target case requires a considerable amount of graph search for the same operations. Nevertheless, in absence of further information about the specific lattice at hand, it is not possible to provide a finer analysis.

We must point out that, in our implementation, we have employed a heapsort-based version that keeps providing us the next element to handle by means of an iterator, instead of completing the sorting step for the initialization.

## 4 Distributivity and the iPred Algorithm

In [5], an extra sophistication is introduced that, as demonstrated both formally in the complexity analysis of the algorithm and also practically, leads to a faster

algorithm; namely, if some further information is maintained along, once the candidates are available there is a constant-time test to pick those that are in the cover, by employing the duality  $y \in \text{uc}(x) \Leftrightarrow x \in \text{lc}(y) \Leftrightarrow x \prec y$ . Constant time also suffices to maintain the additional information. This gives the iPred algorithm. However, it seems that the unavoidable price is to work on formal concepts, as the extra information is heavily set-theoretic (namely, a union of set differences of previously found cover sets for the candidate under study).

Again we show that a fully abstract, lattice-theoretic interpretation exists, and we show that the essential property that allows for the algorithm to work is distributivity: be it due to a distributive  $\mathcal{L}$ , or, as in fact happens in iPred, due to the embedding of the lattice into a distributive lattice, in the same way as concept lattices (possibly nondistributive) can be embedded in the distributive powerset lattice.

We start treating the simplest case, of very limited usefulness in itself but good as stepping stone towards the next theorem. The property where distributivity can be applied later, if available, is as follows:

**Proposition 4.** *Consider two comparable elements,  $x < z$ , from  $\mathcal{L}$ ; let  $Y \subseteq \text{lc}(z)$  be the set of lower covers of  $z$  that show up in the reverse topological sort before  $x$  (it could be empty). Then,  $x \in \text{lc}(z)$  if and only if  $\bigwedge_{y \in Y} (x \vee y) \geq z$ .*

*Proof.* Applying Proposition 1, we know that there is some  $y \in \text{lc}(z)$  such that  $x \leq y \prec z$ . Any such  $y$ , if different from  $x$ , must appear before  $x$  in the reverse topological sort.

Suppose first that no lower covers of  $z$  appear before  $x$ , that is,  $Y = \emptyset$ . Then, no such  $y$  different from  $x$  can exist; we have that both  $x = y \prec z$  and  $\bigwedge_{y \in Y} (x \vee y) = \top \geq z$  trivially hold.

In case  $Y$  is nonempty, assume first  $x \prec z$ ; we can apply Lemma 2:  $x \vee y = z$  for every  $y \in Y$ , hence  $\bigwedge_{y \in Y} (x \vee y) = z$ . To argue the converse, assume  $x \notin \text{lc}(z)$  and let  $x \leq y' \prec z$  as before, where we know further that  $x \neq y'$ : then  $y' \in Y$ , so that  $\bigwedge_{y \in Y} (x \vee y) \leq (x \vee y') = y' < z$ .  $\square$

This means that the test for minimality of Algorithm 1 can be replaced by checking the indicated inequality; but it is unclear that we really save time, as a number of joins have to be performed (between the current element  $x$  and all the elements in the lower cover of the candidate  $z$  that appeared before  $x$  in the reverse topological sort) and the meet of their results computed. However, clearly, in distributive lattices the test can be rephrased in the following, more convenient form:

**Proposition 5.** *Assume  $\mathcal{L}$  distributive. In the same conditions as in the previous proposition,  $x$  is in the lower cover of  $z$  if and only if  $x \vee (\bigwedge_{y \in Y} y) \geq z$ .*

This last version of the test is algorithmically useful: as we keep identifying elements  $Y = \{y_1, \dots, y_m\}$  of  $\text{lc}(z)$ , we can maintain the value of  $y = \bigwedge_{i \in \{1, \dots, m\}} y_i$ ; then, we can test a candidate  $z$  by computing  $x \vee y$  and comparing this value to  $z$ . Afterwards, we update  $y$  to  $y \wedge x$  if  $x = y_{m+1}$  is indeed in the cover. This may save the loop that tests for minimality at a small price.

However, unfortunately, if the lattice is not distributive, this faster test may fail: given  $Y \subseteq \text{lc}(z)$ , the cover elements found so far along the reverse topological sort, it is always true that  $x$  is in the lower cover of  $z$  if  $x \vee (\bigwedge_{y \in Y} y) \geq z$ , because  $z \leq x \vee (\bigwedge_{y \in Y} y) \leq \bigwedge_{y \in Y} (x \vee y)$  and, then, one of the directions of Proposition 4 applies; but the converse does not hold in general. Again an example is furnished by Figure 1(a), one of the basic, standard examples of a small nondistributive lattice; assume that the traversal follows the natural ordering of the labels, and consider what happens after seeing that 1 and 2 are indeed lower covers of  $z = \top$ . Upon considering  $x = 3$ , we have  $Y = \{1, 2\}$ , so that  $x \vee (\bigwedge Y) = x \vee \perp = x < z$ , yet  $x$  is a lower cover of  $z$  and, in fact,  $\bigwedge_{y \in Y} (x \vee y) = (3 \vee 1) \wedge (3 \vee 2) = \top$ . Hence, the distributivity condition is necessary for the correctness of the faster test.

#### 4.1 The Generalized iPred Algorithm

The aim of this subsection is to show the main contribution of this paper: we can spare the loop that tests candidates for minimality in an indirect way, whenever a distributive lattice is available where we can embed  $\mathcal{L}$ . However, we must be careful in how the embedding is performed: the right tool is an injective homomorphism of join-semilattices. Recall that, often, this will *not* be a lattice morphism. Such an example is the identity morphism having as domain the carrier set of a concept lattice  $\mathcal{L}$  over the set of attributes  $X$ , and as range,  $\mathcal{P}(X)$  (see Section 5 for more details on this particular case).

**Theorem 3.** *Let  $(\mathcal{L}', \leq, \vee)$  be a distributive join-semilattice and  $f : \mathcal{L} \rightarrow \mathcal{L}'$  an injective homomorphism. Consider two comparable elements,  $x < z$ , from  $\mathcal{L}$ ; let  $Y \subseteq \text{lc}(z)$  be the set of lower covers of  $z$  that show up in the reverse topological sort before  $x$ . Then,  $x \prec z$  if and only if  $f(x) \vee (\bigwedge_{y \in Y} f(y)) \geq f(z)$ .*

*Proof.* If  $Y = \emptyset$  we have  $x \prec z$  as in Proposition 4; for this case,  $\bigwedge_{y \in Y} f(y) = \top$  (of  $\mathcal{L}'$ ) and  $f(x) \vee (\bigwedge_{y \in Y} f(y)) = f(x) \vee \top = \top \geq f(z)$ .

For the case where  $Y \neq \emptyset$ , assume first  $x \prec z$  and apply Proposition 4: we have that  $\bigwedge_{y \in Y} (x \vee y) \geq z$  whence  $f(\bigwedge_{y \in Y} (x \vee y)) \geq f(z)$ . By Lemma 1, we obtain  $f(z) \leq f(\bigwedge_{y \in Y} (x \vee y)) \leq \bigwedge_{y \in Y} f(x \vee y) = \bigwedge_{y \in Y} (f(x) \vee f(y)) = f(x) \vee \bigwedge_{y \in Y} f(y)$ , where we have applied that  $f$  commutes with join and that  $\mathcal{L}'$  is distributive.

For the converse, arguing along the same lines as in Proposition 4, assume  $x \notin \text{lc}(z)$  and let  $x \leq y' \prec z$  with  $x \neq y'$  so that  $y' \in Y$ : necessarily  $\bigwedge_{y \in Y} f(y) \leq f(y')$ , so that  $f(x) \vee (\bigwedge_{y \in Y} f(y)) \leq f(x) \vee f(y') = f(x \vee y') = f(y') < f(z)$ , where the last step makes use of injectiveness.  $\square$

The generalized iPred algorithm is based on this theorem, which proves it correct. In it, the homomorphism  $f$  is assumed available, and table LC keeps, for each  $z$ , the meet of the  $f(x)$ 's for all the lower covers  $x$  of  $z$  seen so far.

```

RevTopSort( $\mathcal{L}$ );
 $B = \emptyset$ ;
 $H = \emptyset$ ;
for  $x$  in  $\mathcal{L}$ , according to the sort do
   $LC[x] = \top$ ;
  candidates =  $\{x \vee y \mid y \in B\}$ ;
  for  $z$  in candidates do
    if  $f(x) \vee LC[z] \geq f(z)$  then
      add  $(x, z)$  to  $H$ ;
       $LC[z] = LC[z] \wedge f(x)$ ;
       $B = B - \{z\}$ ;
    end
  end
   $B = B \cup \{x\}$ ;
end

```

**Algorithm 2.** The Generalized iPred Algorithm

In the Appendix below, we provide some example runs for further clarification. Regarding the time complexity, again we lack information about the cost of meets, joins, and comparisons in both lattices, and also about the cost of computing the homomorphism. Assuming constant time for these operations, the running time of the generalized iPred algorithm is  $\mathcal{O}(|\mathcal{L}|w(\mathcal{L}))$  (plus sorting): the main loop (line 4-15) is repeated  $|\mathcal{L}|$  times, and then for each of the at most  $w(\mathcal{L})$  candidates, the algorithm checks if a certain condition is met (in constant time) and updates the diagram and the border in the positive case.

If meets and joins do not take constant time, there is little to say at this level of generality; however, for the particular case of the original iPred, which only works for lattices of formal concepts, see [5]: in the running time analysis there, one extra factor appears since the meet operation (corresponding to a set union plus a closure operation) is not guaranteed to work in constant time.

## 5 Conclusions and Future Work

We have provided a formal framework for the task of computing Hasse diagrams of arbitrary lattices through the notion of “border associated with a lattice element”. Although the concept of *border* itself is not new, our approach provides a different, more “axiomatic” point of view that facilitates considerably the application of this notion to algorithms that construct Hasse diagrams outside the formal concept analysis world.

While Algorithm 1 is a clear, straightforward generalization of the Border algorithm of [20,5] (although the correctness proof is far less straightforward), we consider that we should explain further in what sense the iPred algorithm comes out as a particular case of Algorithm 2. In fact, the iPred algorithm uses set-theoretic operations and, therefore, is operating with sets that do not belong to the closure space: effectively, it has moved out of the concept lattice into the

(distributive) powerset lattice. Starting from a concept lattice  $(\mathcal{L}, \leq, \vee, \wedge)$  on a set  $X$  of attributes, we can define:

- $x \leq y \Leftrightarrow x \supseteq y$
- $x \vee y := x \cap y$
- $x \wedge y := \bigvee\{z \in \mathcal{L} \mid z \leq x, z \leq y\} = \bigcap\{z \in \mathcal{L} \mid z \supseteq x, z \supseteq y\}$
- $\top := \emptyset, \perp := X$

Thus,  $\mathcal{L}$  is a join-subsemilattice of the (reversed) powerset on  $X$ , and we can define  $f : \mathcal{L} \rightarrow \mathcal{P}(X)$  as the identity function: it is injective, and it is a join-homomorphism since  $\mathcal{L}$ , being a concept lattice, is closed under set-theoretic intersection. Therefore, Theorem 3 can be translated to:  $x \in \text{lc}(z)$  if and only if  $x \cap (\bigcup_{y \in Y} y) \subseteq z$ , where  $Y$  is the set of lower covers of  $z$  already found; this is fully equivalent to the condition behind algorithm iPred of [5] (see Proposition 1 on page 169 in [5]). Additionally, iPred works on one specific topological sort, where all intents of the same cardinality appear together; our generalization shows that this is not necessary: any linear embedding suffices.

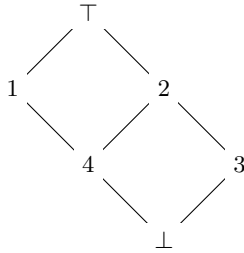
A further application we have in mind refers to various forms of implication known as multivalued dependency clauses [17,18]; in [2,3,4], these clauses are shown to be related to partition lattices in a similar way as implications are related to concept lattices through the Guigues-Duquenne basis ([8,10]); further, certain database dependencies (the degenerate multivalued dependencies of [17,18]) are related to these clauses in the same way as functional dependencies correspond to implications. Data Mining algorithms that extract multivalued dependencies do exist [19] but we believe that alternative ones can be designed using Hasse diagrams of the corresponding partition lattices or related structures like split set lattices [2]. The task is not immediate, as functional and degenerate multivalued dependencies are of the so-called “equality-generating” sort but full-fledged multivalued dependencies are of the so-called “tuple-generating” sort, and their connection to lattices is more sophisticated (see [2]); but we still hope that further work along this lattice-theoretic approach to Hasse diagrams would allow us to create a novel application to multivalued dependency mining.

## Appendix: Examples

We exemplify here some runs of iPred, for the sake of clarity. First we see how it operates on the lattice in Figure 1(a), denoted  $\mathcal{L}$  here, using as  $f$  the injective homomorphism into the distributive lattice of Figure 1(b) provided by the labels. The run is reported in Table 1, where we can see that we identify the respective upper covers of each of the lattice elements in turn. The linear order is assumed to be  $(\top, 1, 2, 3, \perp)$ . Only the last loop has more than one candidate, in fact three. The snapshots of the values of  $B$ ,  $H$ , and LC reported in each row (except the initialization) are taken at the end of the corresponding loop, so that each reported value of  $B$  is a border for the next row. In the Hasse edges  $H$ , thin lines represent edges that are yet to be found, and thick lines represent the edges

**Table 1.** Example run of the iPred algorithm using the lattices in Figure 1

$\mathcal{L}$	$B$	$H$	cand	LC[ $\top$ ]	LC[1]	LC[2]	LC[3]	LC[ $\perp$ ]
init	$\emptyset$	$\diamond$						
$\top$	$\{\top\}$	$\diamond$	$\emptyset$	$\top$				
1	$\{1\}$	$\diamond$	$\{\top\}$	1	$\top$			
2	$\{1, 2\}$	$\diamond$	$\{\top\}$	4	$\top$	$\top$		
3	$\{1, 2, 3\}$	$\diamond$	$\{\top\}$	$\perp$	$\top$	$\top$	$\top$	
$\perp$	$\emptyset$	$\blacklozenge$	$\{1, 2, 3\}$	$\perp$	$\perp$	$\perp$	$\perp$	$\top$



**Fig. 2.** A distributive lattice

**Table 2.** Example run of the iPred algorithm on the lattice in Figure 2

$\mathcal{L}$	$B$	$H$	cand	LC[ $\top$ ]	LC[1]	LC[2]	LC[3]	LC[4]	LC[ $\perp$ ]
init	$\emptyset$	$\diamond$							
$\top$	$\{\top\}$	$\diamond$	$\emptyset$	$\top$					
1	$\{1\}$	$\diamond$	$\{\top\}$	1	$\top$				
2	$\{1, 2\}$	$\diamond$	$\{\top\}$	4	$\top$	$\top$			
3	$\{1, 3\}$	$\diamond$	$\{\top, 2\}$	4	$\top$	3	$\top$		
4	$\{3, 4\}$	$\diamond$	$\{1, 2\}$	4	4	$\perp$	$\top$	$\top$	
$\perp$	$\emptyset$	$\blacklozenge$	$\{3, 4\}$	4	$\perp$	$\perp$	$\perp$	$\perp$	$\top$

found so far. Recall that the values of LC are actually elements of the distributive lattice of Figure 1(b), and not from  $\mathcal{L}$ .

All along the run we can see that LC[z] indeed maintains the meet of the set of predecessors found so far for  $f(z)$  in the distributive embedding lattice; of course, this meet is  $\top$  whenever the set is empty.

Let us compare with the run on the distributive lattice in Figure 2, where the homomorphism  $f$  is now the identity. Observe that the only different Hasse edge

is the one above 3 which now goes to 2 instead of going to  $\top$ . Again the linear sort follows the order of the labels.

Due to the similarity among the Hasse diagrams, the run of generalized iPred on this lattice starts exactly like the one already given, up to the point where node 3 is being processed. At that point, 2 is candidate and will indeed create an edge, but 1 leads to candidate  $1 \vee 3 = \top$  for which the test fails, as  $LC[\top] = 4$  at that point, and  $3 \vee 4 = 2 < \top$ . Hence, this candidate has no effect. After this, the visits to 4 and  $\perp$  complete the Hasse diagram with their corresponding upper covers.

## References

1. Aggarwal, C.C., Yu, P.S.: A new approach to online generation of association rules. *IEEE Transactions on Knowledge and Data Engineering* 13(4), 527–540 (2001)
2. Baixeries, J.: Lattice Characterization of Armstrong and Symmetric Dependencies. Ph.D. thesis, Universitat Politècnica de Catalunya (2007)
3. Baixeries, J.: A formal context for symmetric dependencies. In: Medina and Obiedkov [15], pp. 90–105
4. Baixeries, J., Balcázar, J.L.: Unified characterization of symmetric dependencies with lattices. In: Ganter, B., Kwuida, L. (eds.) *Contributions to the 4th International Conference on Formal Concept Analysis (ICFCA)*. Verlag Allgemeine Wissensch (2006)
5. Baixeries, J., Szathmary, L., Valtchev, P., Godin, R.: Yet a faster algorithm for building the Hasse diagram of a concept lattice. In: Ferré, S., Rudolph, S. (eds.) *ICFCA 2009*. LNCS, vol. 5548, pp. 162–177. Springer, Heidelberg (2009)
6. Balcázar, J.L.: Redundancy, deduction schemes, and minimum-size bases for association rules. *Logical Methods in Computer Science* 6(2:3), 1–33 (2010)
7. Davey, B., Priestley, H.: *Introduction to Lattices and Orders*, 2nd edn. Cambridge University Press, Cambridge (1991)
8. Ganter, B., Wille, R.: *Formal Concept Analysis: Mathematical Foundations*. Springer, Heidelberg (1999)
9. Godin, R., Missaoui, R., Alaoui, H.: Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence* 11, 246–267 (1995)
10. Guigues, J., Duquenne, V.: Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Mathématiques et Sciences Humaines* 95, 5–18 (1986)
11. Kryszkiewicz, M.: Representative association rules. In: Wu, X., Ramamohanarao, K., Korb, K.B. (eds.) *PAKDD 1998*. LNCS (LNAI), vol. 1394, pp. 198–209. Springer, Heidelberg (1998)
12. Kuznetsov, S.O., Obiedkov, S.A.: Algorithms for the construction of concept lattices and their diagram graphs. In: Raedt, L.D., Siebes, A. (eds.) *PKDD 2001*. LNCS (LNAI), vol. 2168, pp. 289–300. Springer, Heidelberg (2001)
13. Luxenburger, M.: Implications partielles dans un contexte. *Mathématiques et Sciences Humaines* 29, 35–55 (1991)
14. Martin, B., Eklund, P.W.: From concepts to concept lattice: A border algorithm for making covers explicit. In: Medina and Obiedkov [15], pp. 78–89
15. Medina, R., Obiedkov, S. (eds.): *ICFCA 2008*. LNCS (LNAI), vol. 4933. Springer, Heidelberg (2008)

16. Nourine, L., Raynaud, O.: A fast algorithm for building lattices. *Information Processing Letters* 71(5-6), 199–204 (1999)
17. Sagiv, Y., Delobel, C., Parker Jr., D.S., Fagin, R.: An equivalence between relational database dependencies and a fragment of propositional logic. *Journal of the ACM* 28(3), 435–453 (1981)
18. Sagiv, Y., Delobel, C., Parker Jr., D.S., Fagin, R.: Correction to “An equivalence between relational database dependencies and a fragment of propositional logic”. *Journal of the ACM* 34(4), 1016–1018 (1987)
19. Savnik, I., Flach, P.A.: Discovery of multivalued dependencies from relations. *Intelligent Data Analysis* 4(3-4), 195–211 (2000)
20. Valtchev, P., Missaoui, R., Lebrun, P.: A fast algorithm for building the Hasse diagram of a Galois lattice. In: Leroux, P. (ed.) *Publications du LaCIM*, pp. 293–306 (2000)
21. Zaki, M.J., Hsiao, C.J.: Efficient algorithms for mining closed itemsets and their lattice structure. *IEEE Transactions on Knowledge and Data Engineering* 17(4), 462–478 (2005)