ELSEVIER

Discrete Applied Mathematics 110 (2001) 169–187

DISCRETE APPLIED MATHEMATICS

Efficient algorithms on distributive lattices

Michel Habib[a], Raoul Medina[a], Lhouari Nourine[a], George Steiner[b,*,1]

[a] *Département d'Informatique Fondamentale, LIRMM, Université Montpellier II — CNRS UMR C09928: 161 rue Ada, 34392 Montpellier Cedex 5, France*

[b] *School of Business, McMaster University, MGD-415, 1280 Main Street West, Hamilton, ONT, Canada L8S 4M4*

Received 10 May 1996; revised 12 April 2000; accepted 24 April 2000

Abstract

We present several efficient algorithms on distributive lattices. They are based on a compact representation of the lattice, called the *ideal tree*. This allows us to exploit regularities in the structure of distributive lattices. The algorithms include a linear-time algorithm to reconstruct the covering graph of a distributive lattice from its ideal tree, a linear-time incremental algorithm for building the ideal lattice of a poset and a new incremental algorithm for listing the ideals of a poset in a combinatorial Gray code manner (in an $\mathscr{H}(1,2)$ code.)

MSC: 05C45; 06A05; 06A06

Keywords: Poset; Ideal lattice; Combinatorial code

1. Introduction

Listing and counting combinatorial objects (linear extensions, ideals, permutations, etc.) for partially ordered sets (posets) has many applications. Cooper and Marzullo [4] and Jégou et al. [11] cite applications in distributed algorithms, Steiner [20] mentions numerous applications in discrete optimization and operations research. For example, the set of events in an execution of a distributed system induces a partial order in which an ideal corresponds to a global state of the execution, i.e., the current values of the variables in each processor, as pointed out by Mattern [13].

* Corresponding author. Tel.: +1-905-525-9140; fax:+1-905-521-8995.

E-mail addresses: habib@lirmm.fr (M. Habib), medina@lirmm.fr (R. Medina), nourine@lirmm.fr (L. Nourine), steiner@mcmaster.ca (G. Steiner).

[1] This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, under Grant No. OGP0001798.

0166-218X/01/$ - see front matter
PII: S0166-218X(00)00258-4

An algorithm for listing N combinatorial objects is said to run in *constant amortized time* (*CAT*) if its total computation time is $O(N)$. Naturally, no algorithm can be faster, up to a constant factor. A *CAT* algorithm is said to be *loopless* (or constant worst-case time) if the amount of computation between successive listed objects is $O(1)$. We say that the objects are listed in a *combinatorial Gray code manner* if the difference between successive objects is small, e.g. one element for subsets or one transposition for permutations of a set. If the list contains each object exactly once, then we have a *combinatorial Gray code* [18]. Listings with small prescribed differences between consecutive objects may allow their faster generation. They also have the added advantage that successive objects can be identified by writing out *only* the change between them, thus facilitating an output whose size may be *less* than the cumulative size of the objects listed. For further interesting properties of combinatorial Gray codes, we refer the reader to the recent survey paper by Savage [18].

The problem of generating exactly once a class of combinatorial objects, so that successive objects differ only in a prespecified way, can be formulated as a Hamilton path/cycle problem: the vertices of the graph correspond to the objects, and two vertices are connected by an edge if they differ from each other the prespecified way. This graph has a Hamilton path exactly if the required combinatorial Gray code exists. Thus, proving the existence of a combinatorial Gray code for a class of objects is equivalent to showing that the above-defined graph has a Hamilton path, which of course may be hard, since it is NP-complete in general. In certain situations, we may know that such a Hamilton path does not exist. In such cases, it may be necessary to allow that certain vertices of the graph are visited (listed) more than once, in order to derive a listing in a combinatorial Gray code manner. Following [16], we say that a graph is in class $\mathscr{H}(s,t)$ if it has a closed walk that visits every vertex at least s times and at most t times, and we refer to the resulting listing of the vertices as an $\mathscr{H}(s,t)$ *code*. Thus a graph is in $\mathscr{H}(1,1)$ exactly if it has a Hamilton path.

In analyzing the computational complexity of graph algorithms, we make the quite common assumption that a graph $G=(V,E)$ can be encoded in $O(|E|)$ space [8,14]. This is based on using a sequential RAM model, in which integers, reals, pointers (addresses) and instructions need $O(1)$ space of memory and all elementary operations (creation, copying, comparison, etc.) on these objects need $O(1)$ time. It is also customary to describe the complexity of a graph algorithm using both $|V|$ and $|E|$ as parameters. This convention is based on considering $|V|$ as the primary measure of the size of a graph and on the fact that $|E|$ may vary between 0 and $|V|\cdot(|V|-1)/2$.

There has been gradual progress over the last 10 years towards the ultimate goal of listing the ideals of a general poset in constant amortized time. Steiner [20] gave an $O(|P|\cdot i(P))$ algorithm for listing the ideals of a poset P, where $i(P)$ is the number of these ideals. Bordat [3] proposed an $O(w(P)\cdot i(P))$ algorithm, which uses $O(|P|^2)$ space and where $w(P)$ is the width of P. Recently, Squire [19] has found an $O(\log|P|\cdot i(P))$ implementation of the algorithm in [20]. All these algorithms are off-line (i.e., the whole poset must be known before the start of the algorithm). We call an algorithm *incremental* (or up-growing) if it does not require full knowledge of the whole poset,

instead it can process the poset one element at a time *provided* the elements are given in the order of some linear extension of the poset. Incremental algorithms are more adaptable for many applications, e.g. for distributed systems or scheduling. An incremental algorithm was reported by Jégou et al. [11]. It has $O(\Delta(P) \cdot i(P))$ time complexity and uses $O(i(P))$ space, where $\Delta(P)$ is the maximum indegree in the covering graph of P.

None of these earlier algorithms lists the ideals in a combinatorial Gray code manner, however. Easy examples demonstrate that a Gray code does not always exist on the ideals of a poset. Pruesse and Ruskey [15] were the first to show the existence of a sequence which lists each ideal twice and in which consecutive ideals differ only by the addition and/or deletion of a single element. They mention that the algorithm they developed from their proof would require $O(|P| \cdot i(P))$ time in the worst case, this can also be reduced to $O(\Delta(P) \cdot i(P))$, however. Because of its recursive nature, its space requirements would be exponential in $|P|$.

The ideals of a poset P form a distributive lattice, called the *ideal lattice*, denoted by $I(P)$. In many applications, it is not sufficient to list just the ideals of a poset, but we need the whole ideal lattice, i.e., we also need the covering relations between the ideals. There is an $O(w(P) \cdot |V|)$ time off-line algorithm to build the covering graph $G(I(P)) = (V, U)$ of the ideal lattice $I(P)$ from the poset P [3]. The best previously known algorithm to compute the covering graph, $G(I(P)) = (V, U)$, of the ideal lattice $I(P)$ requires $O(|V| + |U| + w(P) \cdot |P|^2)$ time and is due to Diehl et al. [5].

Habib and Nourine [9] have introduced a tree representation for distributive lattices. They have shown how to construct this tree from the covering graph of the lattice and how to perform basic lattice operations (e.g. join or meet) using the tree. In this companion paper, we show how to reconstruct the covering graph of a distributive lattice from its tree in time linear in the size of the covering graph. We also present an *incremental* algorithm for building the ideal lattice of a poset requiring again linear time in the size of its output. In other words, all these algorithms generate their outputs in CAT. We also give an algorithm which generates the ideal tree $T(P) = (V, F)$ for the ideal lattice of a poset P in $O(\Delta(P) \cdot |V|)$ time and $O(|V|)$ space. This implicitly gives a listing of the ideals of P in a combinatorial Gray code manner, but the space requirements are still exponential in $|P|$. By performing a simulated depth-first search of the ideal tree without storing the tree, however, we can list the ideals of a poset P in $O(\Delta(P) \cdot i(P))$ time and $O(w(P) \cdot |P|)$ space. Successive ideals differ only by the addition, deletion or replacement of a single element, i.e., the algorithm gives a new proof for the existence of an $\mathscr{H}(1,2)$ code on the ideals of a poset.

The paper is organized as follows. Section 2 contains basic definitions and notation used for partial orders. In Section 3, we present the ideal-tree representation for distributive lattices and the algorithm which reconstructs the lattice from its tree. Section 4 describes the algorithms which build the ideal tree and lattice for a partial order. Section 5 contains the algorithm which lists the $\mathscr{H}(1,2)$ code for the ideals of a poset P in $O(\Delta(P))$ time per ideal and space polynomial in $|P|$. We conclude by a summary and open problems.

2. Basic definitions and notation

Most of the notation we use is standard [8]. Let us review here only the most important concepts. A *partially ordered set* (poset) will be denoted by $P = (X, \leqslant_P)$, where X is the *ground set* of elements or vertices and $\leqslant_P$ is the order relation, i.e., an antisymmetric, reflexive and transitive binary relation whose elements $(a, b) \in \leqslant_P$ are written as $a \leqslant_P b$ $(a, b \in X)$ with the usual interpretation. If $a \leqslant_P b$ but $a \neq b$ then we write $a <_P b$. For $a, b \in X$ we say b *covers* a, denoted by $a \prec b$, if $a <_P b$ and there is no $c \in X$ with $a <_P c <_P b$. Two elements $a, b \in X$ are *comparable* in P (denoted by $a \sim_P b$) if $a \leqslant_P b$ or $b \leqslant_P a$. Otherwise they are said to be *incomparable* (denoted by $a \|_P b$).

The set of *predecessors* of an element x is denoted by $Pred(x) = \{y \mid y <_P x\}$ and the set of *successors* is $Succ(x) = \{y \mid x <_P y\}$. The *maximal* elements of P are defined by $\max P = \{x \mid Succ(x) = \emptyset\}$. We also define $ImPred(x) = \{y \mid y \prec_P x\}$, the set of immediate predecessors, and $ImSucc(x) = \{y \mid x \prec_P y\}$, the set of immediate successors. Furthermore, $\downarrow x = Pred(x) \cup \{x\}$.

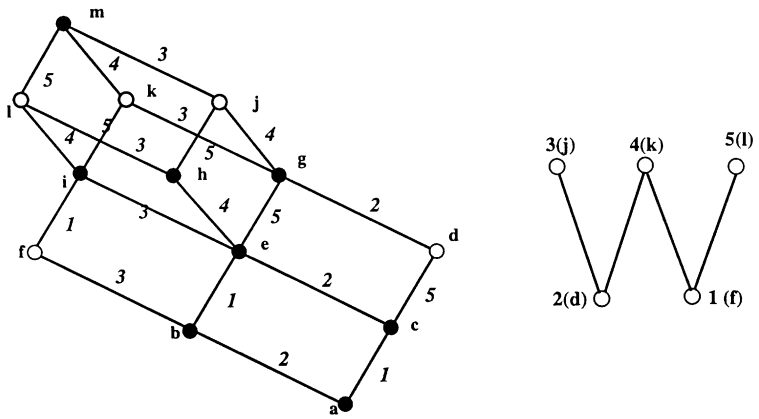
A linear order σ is a *linear extension* of a partial order P if $a \leqslant_P b$ implies $a \leqslant_\sigma b$ for all $a, b \in X$. A subset $I \subseteq X$ is an (*order*) *ideal* if $a \leqslant_P b$ and $b \in I$ imply $a \in I$. The complement of an ideal is called a *filter*. An *antichain* is a subset of X in which no two elements are comparable, a *chain* is a subset of X in which all elements are comparable. The width $w(P)$ of P is the maximum size of an antichain of P. We denote by $\Delta(P)$ the maximum indegree in the covering graph (transitive reduction) of P, i.e., $\Delta(P) = \max_{x \in P} |ImPred(x)|$.

In the following we assume, without the loss of generality, that the poset elements are $X = \{1, \ldots, n\}$ and $\sigma = 1, \ldots, n$ is a fixed linear extension of P. We define $B(i) = \{j \in X \mid i \|_P j$ and $j < i\}$, the suborder of the set of incomparable elements which are before i in the linear extension σ. Denote by $\sigma|B(i)$ the restriction of σ to $B(i)$.

For the rest of the paper, we will refer to order ideals as ideals, in short. It is a well-known fact that for every finite, distributive lattice L the set of join-irreducible (meet-irreducible) elements forms an ordered set whose ideal lattice is isomorphic to L [1]. Therefore, we treat the concepts of ideal lattice and distributive lattice as interchangeable in the remainder of the paper.

3. Tree representation for distributive lattices

Let us denote by $M(L) = (\{1, \ldots, n\}, \leqslant_L)$ the partial order induced by the set of meet-irreducible elements in a distributive lattice L. The unique maximal (resp. minimal) element of L is denoted by $\top$ (resp. $\bot$). (Note that $\top$ corresponds to ideal $\{1, \ldots, n\}$ and $\bot$ to $\emptyset$.) If $I <_L J$, we use the notation $[I, J]$ for the sublattice whose minimal element is I and maximal element is J. Let $G(L) = (V, U)$ be the *covering graph* (Hasse diagram) of the distributive lattice L. If (I, J) is an edge of $G(L)$, then I and J differ only by a single element of $M(L)$ and we call this element *the label* of



(a) A distributive lattice L (b) $M(L)$

Fig. 1. A labelling of a distributive lattice. The nondarkened vertices are the meet-irreducible elements (i.e. $M(L)$). The numbering of vertices in (b) corresponds to a linear extension of $M(L)$.

the edge (I,J). The following lemma is a direct consequence of the labelling proposed in [2].

Lemma 1. *Let I,J be two elements of L with $J <_L I$, then all paths between I and J in $G(L)$ are labelled with the same set of labels.*

Proof. It is clear that the labels on any path between I and J correspond to the elements of $I \setminus J$. □

Let us also associate with each ideal $I \neq \{1,\ldots,n\}$ in the poset $M(L)=(\{1,\ldots,n\}, \leqslant_L)$ a *label* defined by $Label(I) = \min_\sigma(M(L) \setminus I)$. Note that $I \cup \{Label(I)\}$ is an ideal as well, and the relation $I_1 \rightarrow I_2 \Leftrightarrow I_2 \setminus I_1 = \{Label(I)\}$ defines a spanning tree $T(L)=(V,F)$ of the covering graph $G(L)$ of the ideal lattice of $M(L)=(\{1,\ldots,n\}, \leqslant_L)$. In other words, $T(L)$ is the set of all paths starting from $\top$ in $G(L)$ whose labelling respects σ^{-1}, i.e., whose labels form a monotone decreasing sequence. The edges from a vertex of $T(L)$ are assumed to be ordered from left to right by decreasing value of their labels. For example, the upper covers of g in the lattice of Fig. 1 are the ideals k and j, the edge between g and k has the label 3, which is smaller than the label 4 of the edge between g and j. Therefore, the parent of g is k in $T(L)$, as it is shown in Fig 2.

This spanning tree, called the *ideal tree*, was first introduced in [9], where the following was proved.

Theorem 1. *The tree representation $T(L)=(V,F)$ can be computed from the covering graph $G(L)=(V,U)$ in $O(|V|+|U|)$ time.*

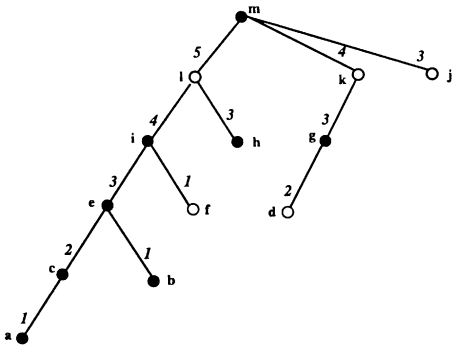


Fig. 2. The spanning tree for the lattice in Fig. 1.

It was also proved in [9] that this tree is a convenient representation for a distributive lattice (i.e., all classical operations on the lattice, such as reachability, least upper bound and greatest lower bound, can be done efficiently using this tree). Note that this tree is not unique for L, but a *unique ideal tree* corresponds to each linear extension of $M(L)$. We also note that tree representations have been suggested for lattices in the past, see e.g. [7], but these do not represent an encoding of the lattice, i.e., they do not allow the efficient retrieval of the lattice from the tree.

In the following, we show the reverse of Theorem 1, i.e., that the covering graph $G(L) = (V, U)$ of a distributive lattice L can also be computed in $O(|V| + |U|)$ time from its ideal tree $T(L) = (V, F)$.

For algorithmic purposes, we use the following data structure: for each vertex I of $T(L)$, define

- $Parent(I)$: the vertex immediately before I on the unique path from $\top$ to I in $T(L)$.
- $Label(I)$: the label of the edge between I and $Parent(I)$.
- $Child(I)$: the list of children of I in $T(L)$, sorted in decreasing order of their labels.

Since storing the lists $Child(I)$ corresponds to storing the edges of $T(L)$, to store the lists $Child(I)$ requires $O(|F|)$ space. Since $T(L)$ is a tree, however, we have $|F| = |V| - 1$. So, to store the ideal tree in this data structure clearly requires only $O(|V|)$ space. This is *less* than the size of the edge set of the covering graph $G(L)$, which can be as large as $\Omega(|V| \cdot \log|V|)$, e.g. for boolean lattices.

Lemma 2. *If I covers J in $G(L)$ then $Label(I) \geqslant Label(J)$; and if we also have $Label(I) > Label(J)$ then the edge $(I,J) \in F$.*

Proof. Since I covers J, $I \supset J$ and $(M(L) \setminus I) \subset (M(L) \setminus J)$. Therefore, $Label(I) = \min_\sigma(M(L) \setminus I) \geqslant Label(J) = \min_\sigma(M(L) \setminus J)$. If we also have $Label(I) > Label(J)$, then the unique element in $I \setminus J$ must be $Label(J)$ and therefore, $I \setminus J = Label(J)$. Thus the edge (I,J) must have as its label the element $Label(J)$, which means that the edge (I,J) is in the tree $T(L)$. □

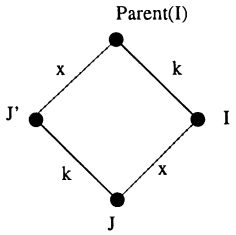


Fig. 3.

We discuss now how to get all immediate predecessors of an ideal I in L using only the information stored for $T(L)$. Let I be a vertex of $T(L)$ with $Label(I)=k$. Using Lemma 2, a vertex with a label greater than k cannot be covered by I in L. Thus all immediate predecessors of I in L have a label less than or equal to k. Furthermore, all immediate predecessors of I in L with label less than k are children of I in $T(L)$. In order to get the remaining predecessors of I, it remains to compute efficiently the set of immediate predecessors whose label is equal to k, that we denote by $Im(I)$. In the following lemma, we show that an immediate predecessor of $Parent(I)$ in L with label greater than $k=Label(I)$ has a child J with label k if and only if J belongs to $Im(I)$.

Lemma 3. *Let I be an ideal with $Label(I)=k$ and $Im(I)=\{J \mid J \in ImPred(I)$ in L and $Label(J)=k\}$.*

1. *If $J \in Im(I)$ then there exists $J' \in ImPred(Parent(I))$ in L such that $J \in Child(J')$.*
2. *If $J' \in ImPred(Parent(I))$ such that $Label(J') > k$ then there exists $J \in Child(J')$ with label k and $J \in Im(I)$.*

Proof. Let us first note a simple (quadrilateral) property for distributive lattices: Suppose that $Parent(I)$ has another immediate predecessor J' in L, then there exists a J which is an immediate predecessor of I and J'. (See Fig. 3.) This is true, since both $I=Parent(I)\setminus\{k\}$ and $J'=Parent(I)\setminus\{x\}$ are immediate predecessors of $Parent(I)$ in L, which means that both k and x are minimal elements of $M(L)\setminus Parent(I)$, and so $J=Parent(I)\setminus\{k,x\}$ is also an ideal and an immediate predecessor of both I and J' in L.

1. Any $J \in Im(I)$ is an immediate predecessor of I in L but not a child of I in $T(L)$. Then by the quadrilateral property $J'=Parent(J)$ is a child of $Parent(I)$ and therefore $Parent(J) \in ImPred(Parent(I))$.
2. Let $J' \in ImPred(Parent(I))$ such that $Label(J')=k' > k$ and the label of the edge $(J', Parent(I))$ is x. Then by the quadrilateral property, there exists J such that J is an immediate predecessor of both I and J' in L. Moreover, the label of the edge

(J,J') is k and of (J,I) is x in $G(L)$. Since $k < k'$, it follows by Lemma 2 that J is a child of J' in $T(L)$. □

Theorem 2. *Let $T(L) = (V,F)$ be an ideal tree and $I \in V$ a vertex of $T(L)$. Then $ImPred(I) = Child(I) \cup \{Child(J';k) \mid J' \in ImPred(Parent(I))$ and $Label(J') > Label(I)\}$, where $Child(J';k)$ denotes the child of J' labelled k.*

Proof. Lemmas 2 and 3 show that this set is exactly the set of immediate predecessors of I in L. □

Based on Theorem 2, the covering graph of L can be reconstructed from the ideal tree in $O(|V| + |U|)$ time, using the following algorithm.

Algorithm 1. Retrieval of the covering graph

Data: An ideal tree $T(L) = (V,F)$

Result: The adjacency list of immediate predecessors, sorted in nonincreasing order of their labels, for every vertex $I \in V$ in the covering graph $G(L) = (V,U)$ of the corresponding lattice L.

```
begin
    ImPred(⊤) = Child(⊤);
1   for each k ∈ [1...n] do
        Compute SORT[k] the list of elements with label k in T(L);
    for k = n downto 1 do
        {Computing the immediate predecessors of lattice elements labelled k}
        for each I ∈ SORT[k] do
            ImPred(I) ← {};
            J' ← the first element in ImPred(Parent(I));
            While J' ≠ I do
2               Let J be the child of J' with label k; { J is the first child of J'}
                ImPred(I) ← ImPred(I) ∪ {J};
                J' ← the next element in ImPred(Parent(I));
3           ImPred(I) ← ImPred(I) ∪ Child(I);
        for each I ∈ SORT[k] do
            Delete I from the list of children of its parent;
end
```

Theorem 3. *For a distributive lattice L, Algorithm 1 reconstructs from its ideal tree $T(L) = (V,F)$ the covering graph $G(L) = (V,U)$ in $O(|V| + |U|)$ time.*

Proof. The computation of the lists $SORT[k]$, $k \in [1...n]$, in line 1 can be done in linear time in the size of the lists by exploring $T(L)$.

The principal *for* loop scans at each step k the list $SORT[k]$. The second *for* loop scans all elements in $SORT[k]$ and computes the list of immediate predecessors for each element I in $SORT[k]$.

We use an inductive argument to show that the list of immediate predecessors of a vertex I is constructed recursively and the immediate predecessors are added to it in nonincreasing order of their labels: Indeed, the immediate predecessors of the top element are sorted, since they correspond exactly to its children. Let I be a vertex with label k, then the *while* loop computes the second part, i.e., $Im(I)$, of the set of immediate predecessors for I (see Lemma 3 and Theorem 2). In fact, we scan the list of predecessors of $Parent(I)$ in nonincreasing order of their labels until we reach the vertex I, since $ImPred(Parent(I))$ is also sorted in nonincreasing order of the labels by our inductive hypothesis. All these elements $J' \in ImPred(Parent(I))$ have a child with label k, which is an immediate predecessor of I (see Lemma 3). Since the children of I are also sorted in decreasing order of their labels, the list of immediate predecessors of I is automatically sorted, when we attach $Child(I)$ to the end of $ImPred(I)$) in line 3 of the algorithm.

Since each I is deleted from the list of children of its parent as soon as $ImPred(I)$ is constructed, children with greater labels are deleted before children with smaller labels. This is why J will be the first child of J' in line 2 in the *while* loop, so that we do not have to search for it. Therefore, only a constant processing time is required for each immediate predecessor of I, i.e., the total time needed is $O(|V|+|U|)$ indeed. □

4. Building the ideal tree and ideal lattice for posets

Let us consider now applications of the ideal tree to direct computations of the ideal lattice of a partial order. First, we present an algorithm to generate the ideal tree (and thus the ideals) directly from the poset.

4.1. Building the ideal tree for a partial order

We denote by $T(P)$ the ideal tree of $I(P)$ defined by σ. In the following, we give an incremental algorithm to build $T(P)$ from P *without* computing the whole lattice $I(P)$. The only assumption we make is that the elements of P are added in the order they appear in σ. Let us denote by $P+\{x\}$ the poset obtained by adding a maximal element x to P. The ideal tree $T(P+\{x\})$ corresponds to the ideal tree of $I(P+\{x\})$ defined by the new linear extension σ,x. The algorithm assumes that $T(P)$ has already been computed and it builds $T(P+\{x\})$ from $T(P)$. Clearly, the ideals in $T(P)$ do not contain the element x. Thus, the remaining ideals to be generated are the ones containing the element x, and therefore they must contain $\downarrow x$. The following proposition characterizes these ideals.

Proposition 1. *Let I be an ideal such that $I \supseteq \downarrow x$. Let I be partitioned into $I = \downarrow x \uplus J$, the disjoint union of $\downarrow x$ and the set of remaining elements, J. Then I is an ideal of $P+\{x\}$ iff J is an ideal of $B(x)$.*

Proof. Suppose that J is not an ideal of $B(x)$. Then there exist $y \in B(x)$ and $z \in J$ such that $y <_P z$ and $y \notin J$. Since $y||_P x$ by the definition of $B(x)$, y does not belong to $\downarrow x$. Thus $J \uplus \downarrow x$ is not an ideal, a contradiction.

Conversely, suppose that J is an ideal of $B(x)$ and $J \uplus \downarrow x$ is not an ideal of $P + \{x\}$. Then there exist $y \in P$ and $z \in J \uplus \downarrow x$ such that $y <_{P+\{x\}} z$ and $y \notin J \uplus \downarrow x$. Since $\downarrow x$ contains all the predecessors of x and since x is maximal in $P + \{x\}$, we have $y||_{P+\{x\}} x$. Thus y belongs to $B(x)$. Element z cannot belong to $\downarrow x$, otherwise we would have $y <_P x$. Thus z belongs to J. Since y does not belong to J and since we have $y <_P z$, J is not an ideal of $B(x)$, a contradiction. □

For an ideal I of P, let $Code(I)$ be the set of labels on the path from the vertex $\top$ to I in $T(P)$. $Code(I)$ corresponds to the elements of P *not* contained in I, so $Code(I)$ is a filter of P and $P \setminus Code(I)$ is an ideal of P. Moreover, we may assume that $Code(I)$ is a list sorted according to σ^{-1} (i.e., the sequence of labels on the path from $\top$ to I in the $T(P)$ corresponding to σ^{-1}).

$T(P)$ contains all ideals of $P + \{x\}$ not containing x, and all these ideals will have x as the first element in their $Code$ in $T(P + \{x\})$, since $Code$ always contains the elements in decreasing order. This means that the root of $T(P)$ is a child of the top element of $T(P + \{x\})$, with label x. The remaining ideals I in $T(P + \{x\})$ all contain x, and the sets $J = I \setminus \downarrow x$ form the ideal lattice $I(B(x))$. Let us denote by $T(B(x))$ the ideal tree of $I(B(x))$, which corresponds to the linear extension $\sigma|B(x)$ of $B(x)$, obtained as the restriction of σ to the suborder $B(x) \subseteq P$. Based on Proposition 1, the main problem we are faced with is the efficient generation of ideals for $B(x)$. In the following, we show that $T(B(x))$ is isomorphic to a subtree of $T(P)$ and it can be identified efficiently.

Let us first define a useful operation on ideal trees.

Gluing operation: Let T and T' be two ideal trees and x a label corresponding to a poset element which is not contained in any ideals in T and is a maximal element of every ideal of T'. We define a *gluing operation* between T and T', denoted by $T \bowtie^x T'$ as follows:

- Link the root of T as the first child of the root of T' (this guarantees that children are sorted according to σ^{-1}).
- Label the edge between T and T' by x.

As an example, consider for T the subtree rooted at vertex l in Fig. 2 and let T' be the subtree rooted at m containing the vertices $\{m, k, j, g, d\}$. The whole tree can be viewed as the result of gluing together T and T' with $x = 5$.

Proposition 2. *Let P be a partial order and $T(P)$ its ideal tree. Then $T(P + \{x\}) = T(P) \bowtie^x T(B(x))$, where x is a maximal element of $P + \{x\}$.*

Proof. Trivial, using Proposition 1. □

Proposition 3. *$T(B(x))$ is isomorphic to a subtree of $T(P)$ rooted at $\top$ of $T(P)$.*

Proof. Clearly $\downarrow x$ is an ideal of $P+\{x\}$. By Proposition 1, each ideal J of $B(x)$ corresponds to another ideal of P that is $J \uplus Pred(x)$. The empty ideal of $B(x)$ corresponds to $Pred(x)$.

No ideal $J \uplus Pred(x)$ can have a label from $ImPred(x)$ in its $Code$. Furthermore, if an ideal I satisfies such a condition, then so does necessarily its parent. So the ideals $J \uplus Pred(x)$ must form a subtree of $T(P)$, which is rooted at $Pred(x) \uplus B(x) = \top$ of $T(P)$. □

By the repeated application of Proposition 2, we obtain the following.

Proposition 4. *Let $P=(X,E)$ be a poset and $\sigma = 1,\ldots,n$ a linear extension of P. Then $T(P) = T(B(1)) \bowtie^1 \cdots \bowtie^n T(B(n))$.*

So to recursively build $T(P)$, we only have to copy and glue subtrees, starting with one vertex corresponding to the empty ideal (empty poset). This is implemented in the following procedures. Procedure $Left(n)$ creates the leftmost subtree, $r \leftarrow Left(n-1)$, containing all ideals that do not include n. Procedure $Right(n,r,root)$ then creates the remainder of the tree (i.e., all ideals containing n) by traversing the tree rooted at r.

```
Procedure : Left(n)
begin
  Create a new vertex Root;
  if n = 0 then
    return(Root);
  r ← Left(n − 1);
  {Copy the subtree Root from r};
  Right(n, r, Root);
  {Gluing the left tree r and the right tree Root};
  Parent(r) ← Root;
  AddChild(Root, r);
  Label(r) ← n;
  Return(Root);
end
```

```
Procedure: Right(n, r, root) begin
1  for each child s of r such that Label(s) ∉ ImPred(n) do
       Create a copy t of s;
       Parent(t) ← Root;
       AddChild(Root, t);
       Label(t) ← Label(s);
       Right(n, s, t);
end
```

Algorithm 2. Building the ideal tree

Data: A partial order P given by the lists of immediate predecessors for elements of P.

Result: The ideal tree $T(P) = (V, F)$ rooted at R.

begin

Compute $\sigma \leftarrow 1, 2, \ldots, n$ a linear extension;

$R \leftarrow Left(n)$

end

Theorem 4. *Algorithm* 2 *builds the ideal tree* $T(P) = (V, F)$ *from the poset* P *in* $O(\Delta(P) \cdot |V|)$ *time and* $O(|V|)$ *space*.

Proof. We note first that during one call to $Right(n, r, root)$ an element is copied only once.

By the proof of Proposition 3, a child s of r should be copied exactly when $Label(s) \notin ImPred(n)$. The children of r can be divided into two parts: the children which are copied; and the children that are visited without being copied. The size of the second part is bounded by $\Delta(P)$, since the number of children in this part is limited by the number of different labels they can have, which is at most $|ImPred(n)|$. So the cost of processing the second part of the children of r is at most $O(\Delta(P))$, while the cost of processing any child of r from the first part can be charged to the copy of this child. This means that the total time needed to compute $T(B(n))$ is $O(\Delta(P) \cdot |T(B(n))| + |T(B(n)))$. Summing this up over the elements of P results in the claimed time complexity of $O(\Delta(P) \cdot |T(P)|)$ for the algorithm, while its space requirement is clearly $O(|T(P)|)$. □

4.2. Building the ideal lattice

The algorithm proposed in [6] takes the covering graph of P as data, and computes the covering graph of the ideal lattice of P in $O(|V| + |U| + w(P) \cdot |P|^2)$ time. In this section, we give an algorithm requiring $O(|V| + |U|)$ time, which makes its time complexity linear in the size of the output. Moreover, the input to our algorithm can be *any* adjacency graph of P whose transitive closure is P.

Propositions 1 and 3 imply the following theorem for distributive lattices, well known in the folklore.

Theorem 5. *Let* P *be a partial order*, $I(P)$ *its ideal lattice and* x *a maximal element of the poset* $P + \{x\}$. *Then* $I(P + \{x\})$ *can be obtained from* $I(P)$ *by creating an isomorphic copy of the sublattice* $[Pred(x), \top]$ *of* $I(P)$, *with the isomorphism edges getting the label* x.

Fig. 4 demonstrates Theorem 5 for $P = \{1, 2, \ldots, 5\}$ and $x = 6$. The sublattice $[Pred(x), \top]$ contains the lattice elements g, j, k, and m. The primed elements are their corresponding copies. The isomorphism edges are the dashed lines in Fig. 4(a). Fig. 4(b) shows how the gluing operation, defined on the ideal tree earlier, corresponds to the copy isomorphism. Observe that $Pred(6) = g$ and $Code(g) = (4, 3) = B(6)$ in $T(P)$.

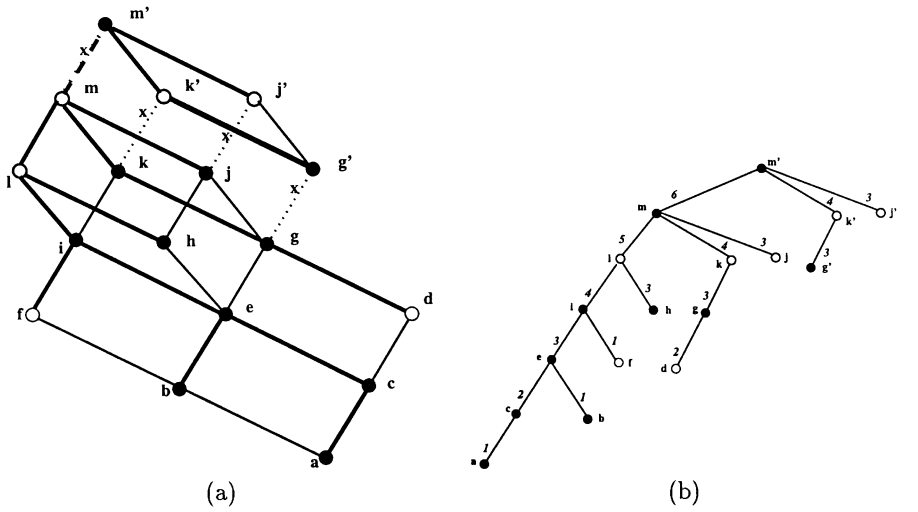


Fig. 4. Example of computing the ideal lattice. The nondarkened vertices are the meet-irreducible elements representing the underlying poset.

We modify the data structure used for storing the ideal tree, in order to incorporate storing the covering edges of the lattice. For each $I \in I(P)$, we store

- *ImSucc*(I), the list of immediate successors of I in $I(P)$. The first immediate successor represents the parent of I in the ideal tree (the heavy edge in Fig. 4(a)).
- *Child*(I), the list of children of I in the ideal tree, sorted in decreasing order of their labels.
- *Label*(I), the label of I in the ideal tree.

Clearly, the space used by this data structure is only $O(|V| + |U|)$ memory units.

Algorithm 3. Building the ideal lattice

Data: Adjacency lists of predecessors of any acyclic graph of P
Result: The covering graph $G(I(P)) = (V, U)$ of the ideal lattice of P
begin
Compute $\sigma \leftarrow 1, \ldots, n$ a linear extension of P;
$i \leftarrow 1$;
$\top \leftarrow$ a newly created vertex; *ImSucc*$(\top) \leftarrow Nil$;
While $i < n$ **do**
1 Find the ideal I corresponding to $\{1, \ldots, i-1\} \setminus B(i)$;
2 Create a copy $[I', \top']$ of the sublattice $[I, \top]$;
3 Link $[I', \top']$ and $[I, \top]$ by isomorphism;
4 Connect the previous ideal tree with the subtree in $[I, \top]$ by the gluing operation; $i \leftarrow i + 1$;
end

Theorem 6. *Algorithm* 3 *builds the covering graph* $G(I(P)) = (V, U)$ *of the ideal lattice* $I(P)$ *for a poset* P *in* $O(|V| + |U|)$ *time and space* (*without counting the storage required by the input*).

Proof. The use of the ideal tree allows us to implement line 1 of the algorithm in *CAT*. The ideal I in line 1 of the algorithm corresponds to $\{1,\ldots,i-1\}\setminus B(i)=Pred(i)$, so $Code(I)$ consists of the elements of $B(i)$ sorted according to σ^{-1}. Let us denote this sequence of elements by $y_1, y_2, \ldots, y_k$. When looking for I, we must find in the tree the path from $\top$ to I, i.e., the path whose sequence of labels is exactly $Code(I)$. Let this path be $\top = I_0, I_1, \ldots, I_k = I$. All codes are sorted according to σ^{-1}, so $Label(I_1)$ must be the first element of $B(i)$ in σ^{-1}, which is y_1. Similarly, we must have $Label(I_j) = y_j$ for $1 < j \leqslant k$. The children of each ideal are also sorted by their labels according to σ^{-1}. So, when looking for the child of I_0 whose label is y_1 we may first encounter some other children of I_0 whose label does not belong to $B(i)$. Note, however, that all these labels must be maximal elements of $Pred(i)$, i.e., they are from $ImPred(i)$, and must be before y_1 in σ^{-1}. Similarly, when searching for the child of I_j whose label is y_j $(1 \leqslant j \leqslant k)$, any child whose label is not from $B(i)$ must have a label from $ImPred(i)$, and this label comes between y_{j-1} and y_j in σ^{-1}. This implies that no two of the searched and discarded children can have the same label, and there can be at most $|ImPred(i)|$ such children before we find I. Thus, we have to search at most $|ImPred(i)| + |B(i)|$ vertices of the current tree in one execution of line 1 of the algorithm. Furthermore, note that we do *not* need to know a priori the sets $B(i)$ to perform this search: We process the children of the I_j in decreasing order of their labels. If J is the next child of I_j and $Label(J) \in ImPred(i)$, then we discard J and take the next child of I_j; if $Label(J) \notin ImPred(i)$, then $Label(J) = y_j$ and we have just found the next element in $Code(I)$ (and in $B(i)$). Let E be the set of covering edges in P. For the whole algorithm, line 1 requires $\sum_{i=1}^{n}(|ImPred(i)| + |B(i)|) = |E| + \sum_{i=1}^{n} |B(i)| \leqslant |E| + |V| \leqslant |U| + |V|$ time.

To perform lines 2 and 3, we have to create J' for every $J \in [I, \top]$; we have to update $ImSucc(J)$ by adding to it J' for every $J \in [I, \top]$; we have to add K' to the list $Child(J')$ for every $K \in Child(J)$; and we have to create $ImSucc(J')$ by adding to it K' for every $K \in ImSucc(J)$. Since all the input lists are in decreasing order of the labels of their elements, the resulting lists will also be sorted in this way, and the insertion or creation of every new item can be done in constant time per item. Line 4 can clearly be done in constant time. So all these operations can be realized in $O(|V| + |U|)$ time for the whole algorithm. The space requirements are clearly $O(|V| + |U|)$, as discussed above. □

5. Listing the ideals of a poset

In this section, an algorithm is given for efficiently listing the ideals of a poset in an $\mathscr{H}(1,2)$-code sequence, where the consecutive ideals differ in at most two elements. (We note that there are simple examples of posets [15] whose ideals do not have

a combinatorial Gray code in which consecutive ideals differ only in exactly one element.) Algorithm 2 could also be used to list the ideals, however, that algorithm has to store the whole ideal tree, which makes its space requirements exponential in the size of the poset. The new algorithm has the same running time, but its space requirements are polynomial in $|P|$. Our new algorithm is also based on a traversal of the ideal tree, but it simulates a depth-first search of the tree *without* storing it, which allows us to reduce the space requirements to $O(w(P) \cdot |P|)$.

Let I be an ideal of P. We denote by $S(I)$ the labels of the children of I sorted according to σ^{-1}. Since the labels are elements of P, we can view $S(I)$ also as a subset of P. Let $S(I) = \{y_1, \ldots, y_k\}$, with $y_1 >_\sigma \cdots >_\sigma y_k$. We denote by $\{I_1, \ldots, I_k\}$ the set of children of I, where $Label(I_i) = y_i$. The complexity of the algorithm is essentially determined by the time required to compute from $S(I)$ $S(I_i)$, sorted according to σ^{-1}.

By definition, all elements in $S(I_i)$ are maximal in the ideal I_i, $Code(I_i)=(Code(I), y_i)$, and $Code(I_i)$ is sorted according to σ^{-1}. Therefore, all elements in $S(I_i)$ are before y_i in the linear extension σ. The elements of $S(I)$ are maximal in the ideal I, and thus, the elements of $S(I) \setminus \{y_i\}$ are also maximal in I_i. As a consequence, all elements of $S(I)$ which are before y_i in σ, will belong to $S(I_i)$. This proves the following lemma.

Lemma 4. *Let* $S_1(I_i) = \{y_{i+1}, y_{i+2}, \ldots, y_k\}$, *i.e., the set of labels equal to* $S(I) \setminus \{y_1, \ldots, y_i\}$. *Then* $S_1(I_i) \subseteq S(I_i)$.

From Lemma 4 we know a subset of $S(I_i)$. The next lemma shows how to compute the remaining elements of $S(I_i)$.

Lemma 5. *Let* $S_2(I_i)$ *be the set of elements which are maximal in* $I \setminus \{y_i\}$, *but not maximal in I, i.e., they are the immediate predecessors of* y_i *which become maximal when* y_i *is removed from I. Then* $S_2(I_i) \subseteq S(I_i)$.

Proof. By definition, all elements in $S_2(I_i)$ are maximal in I_i and are before y_i in σ, since they are predecessors of y_i. Thus $S_2(I_i) \subseteq S(I_i)$ indeed. □

Lemma 6. $S(I_i) = S_1(I_i) \uplus S_2(I_i)$.

Proof. Follows directly from Lemmas 4 and 5. □

Lemma 6 gives us a way to compute $S(I_i)$ from the knowledge of $S(I)$. Part of this set, $S_1(I_i)$ is inherited from the parent. The following Proposition yields a method to compute this inherited set in constant time per ideal during the depth-first search.

Proposition 5. *If I is an ideal and* $\{I_1, \ldots, I_k\}$ *are its children with labels* $y_1, y_2, \ldots, y_k$, *respectively, then*

- $S_1(I_1) = S(I) \setminus \{y_1\}$;
- $S_1(I_i) = S_1(I_{i-1}) \setminus \{y_i\}$, *for* $i = 2, \ldots, k$.

Proof. Follows directly from the definition of $S_1(I_i)$. □

To compute the elements in $S_2(I_i)$, we use counters associated with the elements in P. The counter counts for an element of P how many of its *immediate successors* are still present in the order at the current step. Thus, to *temporarily* remove an element from the order, it is sufficient to decrease the counter of its immediate predecessors. When the counter reaches 0 for an element, then this element becomes maximal in the new current order.

Theorem 7. *Algorithm* 4 *lists the ideals of* P *in* $O(\Delta(P)\cdot i(P))$ *time* (*not counting the time needed by the preprocessing steps*) *and uses* $O(w(P)\cdot|P|)$ *space. Furthermore, each ideal is listed twice and two consecutive ideals differ in at most two elements.*

Algorithm 4: Listing all ideals

Data : Adjacency lists of immediate predecessors for elements of P.
Result : The ideals of P listed in an $\mathscr{H}(1,2)$ code order.
begin
  Compute $\sigma \leftarrow 1,\ldots,n$ a linear extension of P;
  Sort the adjacency lists of P and $\mathrm{Max}(P)$ according to σ^{-1};
  $I \leftarrow \{1,\ldots,n\}$;
  Output I;
  $i \leftarrow 1$; $S(1) \leftarrow \mathrm{Max}(P)$;
  $Code \leftarrow \{\}$; {$Code$ is a stack}
  While $i > 0$ **do**
    if $S(i)$ *is not empty* **then**
      $x \leftarrow$ the first element of $S(i)$;
1     $Flag \leftarrow F$;
      $Push(Code, x)$;
      $I \leftarrow I \setminus \{x\}$;
      $S(i) \leftarrow S(i) \setminus \{x\}$;
      Output I;
2   Remove x from P;
3     $S_2 \leftarrow$ the elements which became maximal after the removal of x;
      $S(i+1) \leftarrow Merge(S(i), S_2)$;
      $i \leftarrow i + 1$;
    else
4   If $Flag = BT$ then output I;
      {$Flag$ is used to avoid outputting I more than once during backtracking}
      $x \leftarrow Pop(Code)$;
      $I \leftarrow I \cup \{x\}$;
5     $Flag \leftarrow BT$;
6   Insert x in P;
      $i \leftarrow i - 1$;
end

Proof. For each ideal I, the algorithm computes the set $S(I)$. This set is always sorted according to the reverse order of σ. The index i represents the level on which the current I can be found in the ideal tree $T(P)$, with $\top$ being on level 1, and the set $S(i)$ corresponds to the current $S(I)$. If I is the top element of $T(P)$, then $S(I) = \mathrm{Max}(P)$. Using $S(I)$, the algorithm lists all the children of I for each ideal I in $T(P)$. Thus all the ideals of P are listed, since $T(P)$ is a spanning tree of the ideal lattice of P.

Since $S(I)$ is sorted according to the reverse order of σ, computing $S_1(I)$, for the current ideal I, is done in constant time by Proposition 5: we simply remove the current first element of $S(I)$. S_2 represents the second part of the maximal elements, the set $S_2(I_i)$ of Lemma 6, for the current child $I \setminus \{x\}$ of I. Using the counters mentioned after Proposition 5, we can compute S_2 by identifying those elements of $ImPred(x)$ whose counter has become 0 after removing x. Since the length of this list is at most $O(\Delta(P))$, this requires no more than $O(\Delta(P))$ time. The adjacency lists $ImPred(x)$ are sorted in σ^{-1} order in the preprocessing step, so S_2 is automatically obtained in this order too. The merging of the two sorted lists, in order to obtain the list $S(i+1)$ for the newly listed child $I \setminus \{x\}$ of I, requires no more than $O(S(I \setminus \{x\}))$ time, and can be charged to the children of $I \setminus \{x\}$. Thus the total time needed to list the child $I \setminus \{x\}$ and its list of children, $S(I \setminus \{x\})$, is equal to $O(\Delta(P))$ for each ideal I.

Each edge (I,J) in the tree $T(P)$ is visited twice: first time is downwards from I to J (corresponding to the *if* case in the algorithm), and then upwards from J to I (corresponding to the *else* case in the algorithm). In the first visit, the element corresponding to the label of the edge is removed from the poset while computing $S_2(J)$. This costs at most $\Delta(P)$ steps, to decrease the counters of its immediate predecessors. When visited upwards, the element is restored in the order. This costs at most $\Delta(P)$ steps too, to increase the counters. Since the tree has $i(P) - 1$ edges, the total time spent updating the counters is $2\Delta(P)(i(P) - 1) = O(\Delta(P) \cdot i(P))$.

Each ideal I is listed twice: First, at the time we traverse the edge $(Parent(I), I)$ in $T(P)$, in the *if* case of the algorithm; second, after the *last* time we reached I during backtracking, after we traversed the edge (I,J) upwards, where J is the last element of $S(I)$. This is accomplished by using the indicator variable *Flag*: I was reached for the last time in a backtracking step if and only if the next step is also a backtracking (upward) move on the tree. This also means that any two consecutive ideals I and J we output differ in at most two elements: If J is a child of I in $T(P)$, then J was obtained from I by deleting an element x. Similarly, if $J = Parent(I)$, then J is obtained from I by adding $Label(I)$ to I. The remaining possibility is that both J and I are children of the same parent in $T(P)$, which means that J can be obtained from I by adding $Label(I)$ and deleting $Label(J)$.

Concerning the space complexity, the size of the stack *Code* may not exceed $|P|$, since the longest path in the ideal tree has a length of $|P|$. The lists $S(i)$ may contain at most $w(P)$ elements. The sorted adjacency lists of P require at most $\Delta(P) \leq w(P)$ space per element. Therefore, the total space requirement of the algorithm is $O(w(P) \cdot |P|)$ indeed. □

Let us call Algorithm 4′ the following modified version of Algorithm 4:

1. Delete lines 1 and 5.
2. Change line 4 to Output I.

Algorithm 4′ clearly traverses the ideal tree in the same order as Algorithm 4, but it outputs each ideal *every* time it is visited. Therefore, if I and J denote two such consecutive ideals, then we always have $I = J \cup \{x\}$ or $I = J \setminus \{x\}$ for some appropriate element x, i.e., the consecutive ideals differ in *exactly one* element. (It is no longer true, however, that each ideal is listed exactly twice.) This proves the following.

Corollary 1. *Algorithm* 4′ *lists the ideals of a general poset so that consecutive ideals differ in exactly one element and it requires* $O(\Delta(P) \cdot i(P))$ *time and* $O(w(P) \cdot |P|)$ *space*.

Note that both Algorithms 4 and 4′ make the ideals available in *less* time and space than what would be needed for listing them in traditional list or incidence vector form, which would be proportional to their total size $O(|P| \cdot i(P))$. In fact, if I and J denote two consecutive ideals, then we can "encode" them by simply listing only the element(s) they differ in, i.e., $(I \setminus J) \cup (J \setminus I)$.

6. Conclusions and open problems

We have presented efficient algorithms to reconstruct a distributive lattice from its ideal tree, and to build the ideal lattice for a partially ordered set. These algorithms are *CAT*, and so they have the best possible time complexity, up to a constant factor. We have also presented new algorithms for building the ideal tree and for listing an $\mathscr{H}(1,2)$ code for the ideals of a poset. These algorithms can be made to run in constant time per ideal for some large special classes of posets [17,12,10], but finding a constant amortized time algorithm, which has been the goal for years [15], remains an open problem for the general case.

Acknowledgements

The authors wish to thank anonymous referees for their suggestions which have greatly improved the presentation of the paper.

References

[1] G. Birkhoff, Lattice Theory, Coll. Publ. XXV, Vol. 25, 3rd Edition, American Mathematical Society, Providence, RF, 1967.
[2] R. Bonnet, M. Pouzet, Extensions et stratifications d'ensembles dispersés, C.R. Acad. Sci. Sér. A 268 (1969) 1512–1515.

[3] J.P. Bordat, Calcul des idéaux d'un ordonné fini, Rech, Opér./Oper. Res. 25 (4) (1991) 265–275.
[4] R. Cooper, K. Marzullo, Consistent detection of global predicates, Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, May 1991, pp. 163–173.
[5] C. Diehl, C. Jard, J.-X. Rampon, Computing on-line the covering graph of the ideal lattice of posets, Technical Report, IRISA, Rennes, France, February 1993.
[6] C. Diehl, C. Jard, JX Rampon, Reachability analysis on distributed executions, in: Jouannaud Gaudel (Ed.), TAPSOFT, Lecture Notes in Computer Science, Vol. 688, Orsay, Springer, Berlin, April 1993, pp. 629–643.
[7] B. Ganter, K. Reuter, Finding all closed sets: a general approach, Order 8 (1991) 283–290.
[8] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New-York, 1980.
[9] M. Habib, L. Nourine, Tree structure for distributive lattices and its applications, Theoret. Comput. Sci. 165 (1996) 391–405.
[10] M. Habib, L. Nourine, G. Steiner, Gray codes for the ideals of interval orders, J. Algorithms 25 (1997) 52–66.
[11] R. Jégou, R. Medina, L. Nourine, Linear space algorithm for on-line detection of global predicates, in: J. Desel (Ed.), Structures in Concurrency Theory, Springer, Berlin, 1995, pp. 175–189.
[12] Y. Koda, F. Ruskey, A gray code for the ideals of a forest poset, J. Algorithms 15 (1993) 324–340.
[13] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard, et al. (Eds.), Parallel and Distributed Algorithms, Elsevier, North-Holland, Amsterdam, 1989, pp. 215–226.
[14] C. Papadimitriou, K. Steiglitz, Combinatorial Optimization, Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[15] G. Pruesse, F. Ruskey, Gray codes from antimatroids, Order 10 (1993) 239–252.
[16] G. Pruesse, F. Ruskey, Generating linear extensions fast, SIAM J. Comput. 23 (1994) 373–386.
[17] F. Ruskey, Listing and counting subtrees of a tree, SIAM. J. Comput. 10 (1981) 141–150.
[18] C. Savage, A survey of combinatorial gray codes, SIAM Rev. 39 (1997) 605–629.
[19] M.B. Squire, Enumerating the ideals of a poset, preprint, North Carolina State University, 1995.
[20] G. Steiner, An algorithm for generating the ideals of a partial order, Oper. Res. Lett. 5 (1986) 317–320.