Efficient Representation and Encoding of Distributive Lattices

by

Corwin Sinnamon

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on joint work ([18]) with Ian Munro which appeared in the proceedings of the ACM-SIAM Symposium of Discrete Algorithms (SODA) 2018. I contributed many of the important ideas in this work and wrote the majority of the paper.

Abstract

This thesis presents two new representations of distributive lattices with an eye towards efficiency in both time and space. Distributive lattices are a well-known class of partially-ordered sets having two natural operations called meet and join.

Improving on all previous results, we develop an efficient data structure for distributive lattices that supports meet and join operations in $O(\log n)$ time, where n is the size of the lattice. The structure occupies $O(n \log n)$ bits of space, which is as compact as any known data structure and within a logarithmic factor of the information-theoretic lower bound by enumeration.

The second representation is a bitstring encoding of a distributive lattice that uses approximately $1.26n$ bits. This is within a small constant factor of the best known upper and lower bounds for this problem. A lattice can be encoded or decoded in $O(n \log n)$ time.

Acknowledgements

Many thanks to my supervisor, Ian Munro, for his guidance and his many thoughtful conversations. He always had time for me, and he was always on my side. Equal thanks to John Brzozowski, who supervised me in a long and fruitful research assistantship just prior to my Master's degree, for I would not be where I am without his mentorship and his kindness.

Thanks also to the many professors who inspired me during my seven years at the University of Waterloo, most notably Eric Blais, Naomi Nishimura, Anna Lubiw, Kathryn Hare, and Prabhakar Ragde. Finally, thanks to Bryce Sandlund, Kshitij Jain, Ankit Vadehra, Hicham El-Zein, Sebastian Wild, and the other friends and collaborators I have been lucky enough to know in the last two years.

Table of Contents

List of Figures viii

1 Introduction 1

1.1 Background . . . 2

1.2 Birkhoff's Representation Theorem . . . 3

1.2.1 Chains, Antichains, and the Covering Relation . . . 5

1.2.2 Persistence . . . 5

1.3 Overview . . . 7

2 Persistent Ideal Trees 9

2.1 Ideal Trees . . . 9

2.2 A Faster Data Structure . . . 11

2.2.1 Chain Decompositions and Wavefronts . . . 12

2.2.2 Strategy . . . 15

2.2.3 Computing WAVE . . . 16

2.2.4 Persistent Ideal Tree . . . 17

2.2.5 Computing WAVE^{-1} . . . 19

3 Compact Encoding of Distributive Lattices 25

3.1 Preliminaries . . . 26

3.2 Doubling Lattices . . . 26

3.3 Compression Strategy 29
3.4 Intuitive Algorithm 31
3.5 Decompression Algorithm 33
3.6 Compression Algorithm 34
3.7 Improving the Space Requirements 35

4 Conclusion and Open Problems **42**

References **43**

List of Figures

1.1 Some examples of Birkhoff's Representation Theorem for Distributive Lattices. 4

1.2 A distributive lattice on $n = 19$ elements with $m = 7$ join-irreducible elements (indicated by open circles). The nodes in the lattice are labelled with the corresponding ideals of the join-irreducible poset. 7

1.3 The join-irreducible poset for the distributive lattice in Figure 1.2. 8

2.1 The ideal tree for the distributive lattice in Figure 1.2. The linear extension τ is given in Figure 1.3. 10

2.2 One possible minimal chain decomposition of the join-irreducible poset in Figure 1.3. 13

2.3 The lattice in Figure 1.2 labelled with wavefronts instead of ideals. The chain decomposition is that of Figure 2.2. 14

3.1 (a) A distributive lattice L and its join-irreducible poset. (b) The result of DOUBLE(L, x). 28

3.2 Growing the distributive lattice of Figure 1.2 by a sequence of double operations. 32

3.3 An example of decoding the lattice with covering sequence $(2, 0, 0, 2, 0, 0, 1, 0)$. 36

3.4 A distributive lattice with covering sequence $(2, 1, 0, 1, 0, 0, 0, 2, 0, 1, 1, 0, 0, 0, 0)$. 39

Chapter 1

Introduction

A lattice is a partially-ordered set having two binary operations called *meet* (or greatest lower bound) and *join* (or least upper bound). The most pleasing and well-structured class of lattices is that of distributive lattices, whose meet and join operations satisfy a deceptively simple distributivity condition. Distributive lattices have found algorithmic applications in performing computations on arbitrary partially-ordered sets [6, 20].

The work in this thesis comes from the point of view of succinct [14, 17] or compact [19] data structures, where the notion is to minimize the storage requirements for combinatorial objects while still supporting the natural operations quickly. Such space reduction can lead to the representations residing in a faster level of memory than conventional methods with a resulting improvement in run time. To a lesser extent various forms of graphs, including partial orders have also been examined. An exception is the work on abelian groups [10], which leads to speculation on the applicability of the methods for symbolic computation [16]. This thesis builds on such notions as I address space-efficient techniques for representing distributive lattices.

A natural representation of a lattice, or indeed any partial order, is by the transitive reduction of the associated directed acyclic graph. It can be shown that the transitive reduction of a distributive lattice has at most $n \lg n$ edges[1], and thus it can be represented in $O(n \log^2 n)$ bits; this is in contrast to representations of arbitrary partial orders which require $O(n^2)$ bits, or even arbitrary lattices which require $O(n^{1.5})$ bits [15]. However, the transitive reduction graph does not seem to support efficient computation of the meet or join of a pair of elements.

[1] I use lg to denote $\log_2$.

The main result of Chapter 2 is a space-efficient data structure occupying $O(n \log n)$ bits that supports the evaluation of these meet and join operations in $O(\log n)$ time. The worst-case logarithmic query time is a significant improvement on all previous methods, excluding trivial structures that use quadratic space, and the space usage is no worse than any known method.

However, the data structure is not space-optimal as the lower bound for this problem is known to be $\Theta(n)$ bits. More precisely, the information-theoretic lower bound on the number of bits required to represent a distributive lattice lies between $0.88n$ and $1.257n$, asymptotically [9]. This raises the question of how to usefully represent a distributive lattice in so little space.

To answer this question, Chapter 3 presents an encoding method for distributive lattices that compresses the lattice to approximately $1.25899n + o(n)$ bits. Although this representation does not support any queries on the coded lattice, it requires only $O(n \log n)$ time to convert between the transitive reduction graph and the coded lattice.

This thesis is an extended version of [18] which appeared in the ACM-SIAM Symposium on Discrete Algorithms, 2018.

1.1 Background

Let $(X, \leqslant)$ be a partially-ordered set (*poset*). For $S \subseteq X$, we say $x \in X$ is an *upper bound* on S if $x \geqslant y$ for all $y \in S$. If x is the minimum among all upper bounds of S, then x is the *least upper bound* (LUB) of S. Similarly, x is a *lower bound* on S if $x \leqslant y$ for all $y \in S$, and x is the *greatest lower bound* (GLB) of S if x is the maximum among all lower bounds of S. Note that the GLB and LUB are always unique when they exist.

A partially-ordered set L is a *lattice* if every pair of elements has a LUB and a GLB. For lattices, LUB is called *join* (denoted $\vee$) and GLB is called *meet* (denoted $\wedge$). Hence, a poset is a lattice if and only if $x \vee y$ and $x \wedge y$ exist for all $x, y \in L$.

The following elementary properties of lattices follow easily from the definition.

- The meet and join operations are associative and commutative.
- If $x \leqslant y$, then $x \wedge y = x$ and $x \vee y = y$.
- A lattice must have a unique *top* element above all others and unique *bottom* element below all others in the lattice order. By convention, these are denoted by $\bot$ and $\top$, respectively.

A lattice is called *distributive* if meet and join satisfy the additional property that, for all $x, y, z \in X$,

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z).$$

The complementary statement that $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ can be deduced from this property. Thus, meet and join distribute over one another in a distributive lattice.

Throughout, let L be a finite distributive lattice on n elements. An element $x \in L$ is *join-irreducible* if it is not the minimum element and it is not expressible as $y \vee z$ for any $y, z \neq x$. Similarly $x \in L$ is *meet-irreducible* if it is not the maximum element and it is not expressible as $y \wedge z$ for any $y, z \neq x$. Let $\mathcal{J}(L)$ ($\mathcal{M}(L)$) denote the poset of join-irreducible (meet-irreducible) elements of L, where the partial order is inherited from the lattice order. It is well known that $\mathcal{J}(L)$ and $\mathcal{M}(L)$ are closely related; in particular $|\mathcal{J}(L)| = |\mathcal{M}(L)|$. The concept of irreducible elements of L is essential to understanding distributive lattices.

Henceforth, let $m = |\mathcal{J}(L)|$. It is worth noting that m can vary greatly with respect to n, lying anywhere in the range $[\lg n, n - 1]$.

1.2 Birkhoff's Representation Theorem

A famous theorem of Birkhoff from 1937 [2, 4] proves that the structure of L is completely determined by $\mathcal{J}(L)$ via the *lattice of ideals*.

An *ideal* of a poset $\mathcal{P} = (X, \leqslant)$ is a set $I \subseteq X$ such that whenever $x \in I$ and $y \leqslant x$, $y \in I$ as well. In other words, an ideal is a downward-closed subset of $\mathcal{P}$. Let $\mathcal{O}(\mathcal{P})$ be the set of ideals of $\mathcal{P}$. When ordered by inclusion, $\mathcal{O}(\mathcal{P})$ is a distributive lattice in which $\wedge = \cap$ and $\vee = \cup$.

For $x \in L$, define $\downarrow x = \{y \in L \mid y \leqslant x\}$.

Theorem 1. *[Birkhoff's Representation Theorem]*
For every distributive lattice L, the function $IDEAL\colon L \to \mathcal{O}(\mathcal{J}(L))$ defined by

$$IDEAL(x) = \ \downarrow x \cap \mathcal{J}(L)$$

is a lattice isomorphism. Thus, L is isomorphic to $\mathcal{O}(\mathcal{J}(L))$.

See Figure 1.1 for examples of Birkhoff's theorem on small distributive lattices. Figures 1.2 and fig:bigposet show a larger example.

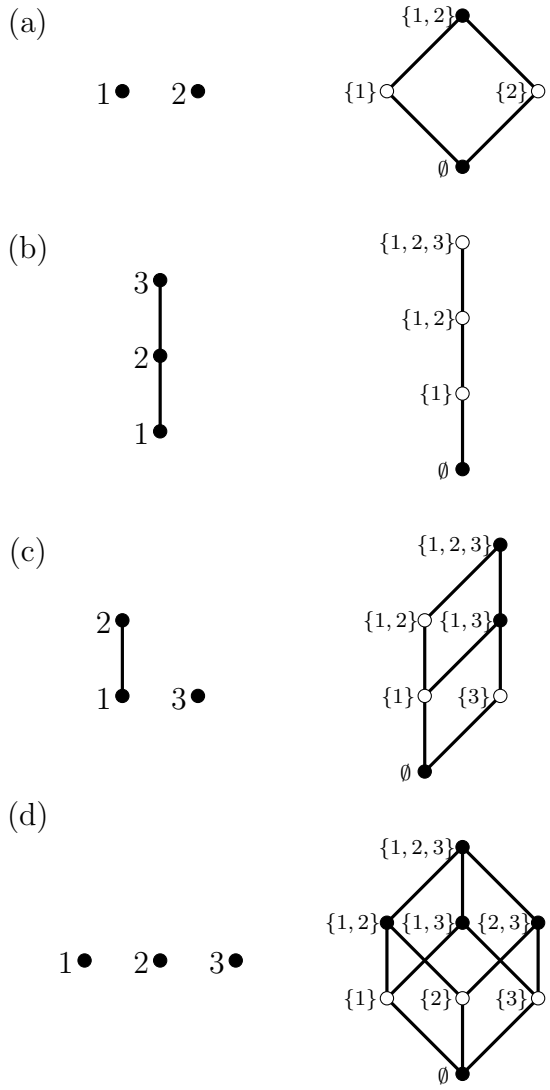


Figure 1.1: Some examples of Birkhoff's Representation Theorem for Distributive Lattices. The diagrams on the left show the join-irreducible posets of the distributive lattices on the right. The lattice nodes are labelled with the ideals of the posets on the left, as determined by the isomorphism IDEAL. The open circles in the lattices indicate the join-irreducible elements.

Note that a dual theorem holds for the lattice of *filters* (i.e., upward-closed sets) of $\mathcal{M}(L)$, ordered by inverse inclusion with $\wedge = \cup$ and $\vee = \cap$. It is slightly more intuitive to work with $\mathcal{J}(L)$ for our purposes, but the choice is not important.

Birkhoff's theorem suggests a general strategy for computing meets and joins in distributive lattices. It states that there is an isomorphism between L and the ideals of $\mathcal{J}(L)$, which we call IDEAL. The join and meet operations, which may appear complicated, are translated to union and intersection through this isomorphism. Thus, if both IDEAL and IDEAL^{-1} can be efficiently computed, then joins and meets may be computed by simple union and intersection operations. Specifically, $x \vee y = \text{IDEAL}^{-1}(\text{IDEAL}(x) \cup \text{IDEAL}(y))$, and $x \wedge y = \text{IDEAL}^{-1}(\text{IDEAL}(x) \cap \text{IDEAL}(y))$. Hence, the time to compute a meet or join is proportional to the time to compute IDEAL, IDEAL^{-1}, and a union or intersection over sets of size at most m. This strategy is discussed and extended in Chapter 2.

1.2.1 Chains, Antichains, and the Covering Relation

We review some basic definitions related to partially-ordered sets to be used in the next chapters. A *chain* of a poset $(X, \leqslant)$ is a subposet $C \subseteq X$ that is totally-ordered; that is, for all $x, y \in C$, either $x \leqslant y$ or $y \leqslant x$. An *antichain* of $(X, \leqslant)$ is a subposet $A \subseteq X$ such that, for all $x, y \in A$, $x \not\leqslant y$ and $y \not\leqslant x$.

The *width* of a poset is the size of the largest antichain in the poset. Dilworth's Theorem [7] states that the elements of any poset can be partitioned into a number of chains equal to the width of the poset. This is always a tight bound on the number of chains as no two elements in the same antichain can lie in a single chain. We call such a partition a *chain decomposition* of the poset.

For $x, y \in L$, say y *covers* x (or x is *covered* by y) if $x < y$ and for all $z \in L$, $x \leqslant z < y \implies x = z$ (i.e., there is no intermediate element z between x and y). This covering relation between elements of a poset uniquely determines the poset as the partial order is the transitive closure of the covering relation.

1.2.2 Persistence

As a critical tool in Section 2.2.3, we use the concept of a persistent data structure. A brief summary is given here; see [8] for more details.

A normal, *ephemeral* data structure might support a set of update operations that change its content or structure. Thus, a sequence of update operations sends the content

of the data structure through many states, or *versions*. A persistent data structure is one that simulates the ephemeral structure and its operations, but also remembers every past version of the structure.

A structure is *partially persistent* if every past version is accessible, but only the most recent version can be updated. For example, if V_0 is the initial version of the data structure and a sequence of update operations $u_1, u_2, \ldots, u_k$ are applied to V_0, then the persistent structure can simulate access operations on any version V_i, which is obtained by applying $u_1, u_2, \ldots, u_i$ to V_0. The structure resembles a list of versions $V_0, \ldots, V_k$, in which V_i is superseded by V_{i+1}.

A structure is *fully persistent* if every past version can be updated as well as accessed. In this case, the versions are not arranged in a list, but in a tree called the *version tree*. The version tree can branch by updating a past version, and so a version is superseded independently by each of its children in the version tree. The relevant sequence of updates that yields a version V from V_0 are those on the path from V_0 to V in the version tree.

The only persistent data structure we require is a fully persistent linked list. An ephemeral linked list consists of a set of nodes that each store some values and a pointer to the next node in the list. This basic list structure supports an access operation that traverses a list from its beginning by following pointers from one node to the next. It also supports the update operations of inserting or deleting a node by modifying the pointers in the adjacent nodes. The update operations only require constant time once the position of the node in the list is found.

It is possible to make a fully persistent list with these same operations using the techniques of Driscoll et al in [8]. The access operation of traversing a list still requires $O(1)$ time per node in the worst case. The update operations require $O(1)$ amortized time; that is, any sequence of k update operations takes $O(k)$ time. Each update will use an amount of space proportional to the number of words that it alters. For example, if an update operation changes a single pointer in one node, that will use a constant number of additional words. Therefore, we may treat the versions of the persistent list as if they were independent linked lists, and the space usage will be proportional to the sum over all updates of the number of updated words. We will use persistence to gain space improvements over storing the versions separately.

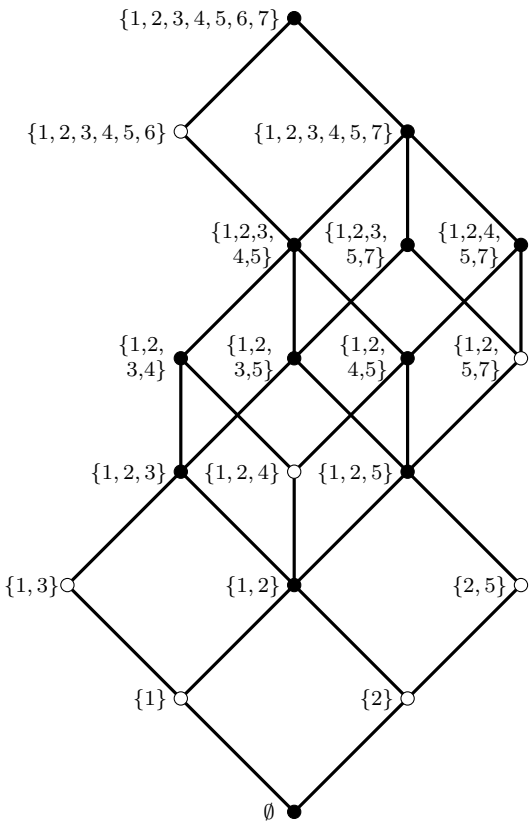


Figure 1.2: A distributive lattice on $n = 19$ elements with $m = 7$ join-irreducible elements (indicated by open circles). The nodes in the lattice are labelled with the corresponding ideals of the join-irreducible poset.

1.3 Overview

We consider four representations of a lattice L. The first two were previously known, and the last two are newly introduced here.

1. **Transitive Reduction Graph (TRG(L))** A natural representation for L is its transitive reduction graph, the directed acyclic graph (DAG) on the elements of L which has an edge (x, y) if and only if $x \leqslant y$ and there is no $z \in L$ such that $x < z < y$ (i.e., y covers x). We denote this representation by TRG(L). This is a

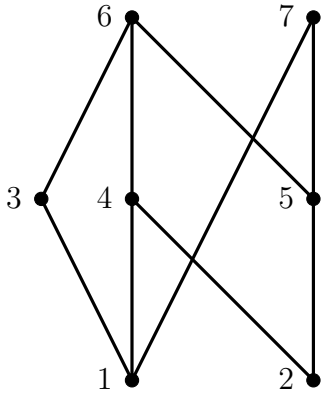


Figure 1.3: The join-irreducible poset for the distributive lattice in Figure 1.2.

very straightforward way to store a lattice while still being reasonably space-efficient: It can be shown that the transitive reduction graph of a distributive lattice has at most $n \lg n$ edges [11], and thus $\mathrm{TRG}(L)$ can be represented in $O(n \log^2 n)$ bits.

2. **Ideal Tree (TREE(L))** The second representation is the *ideal tree* data structure for distributive lattices due to Habib and Nourine [11, 12], described in Section 2.1. This representation requires only $O(n \log n)$ bits of space and computes meets and joins in $O(m)$ time, which may be as fast as $O(\log n)$ or as slow as $O(n)$.

3. **Persistent Ideal Tree (PTREE(L))** In Chapter 2, we construct a new data structure called the *persistent ideal tree*. Like the ideal tree, this representation requires $O(n \log n)$ bits of space, but it allows computation of meets and joins in $O(\log n)$ time.

4. **Compressed Encoding** Chapter 3 describes the final representation, a compressed encoding of L which requires approximately $1.25899n + o(n)$ bits of space. The constant here is obtained by applying arithmetic coding to a more direct compression of the lattice, which encodes the lattice in a binary string containing exactly n zeros and at most $\frac{3}{7}n$ ones. The compression code can be computed from $\mathrm{TRG}(L)$ in $O(n \log n)$ time, and it can be decompressed to $\mathrm{TRG}(L)$ in $O(n \log n)$ time.

Chapter 2

Persistent Ideal Trees

In this chapter, we discuss the previous approach to a data structure for distributive lattices and extend it to obtain our main result:

Theorem 2. *There is a data structure for distributive lattices that requires $O(n \log n)$ bits of space and supports meet and join operations in $O(\log n)$ time. It can be constructed in $O(n \log n + m^{2.5})$ time from TREE(L) or TRG(L).*

2.1 Ideal Trees

In 1996, Habib and Nourine [12] invented a data structure called an *ideal tree* that implicitly computes IDEAL and IDEAL^{-1} to perform meet and join operations. The ideal tree is a clean and elegant way to manipulate distributive lattices; see also [11]. We use their ideas extensively in this chapter.

Recall that L is a distributive lattice on n elements and $\mathcal{J}(L)$ is the join-irreducible poset of L, which has size m. Let τ be a linear extension of $\mathcal{J}(L)$ and write $\tau(p)$ for the position of p in τ. For each $x \in L \setminus \{\bot\}$, let the *block number* of x be defined by

$$\text{BLOCK}(x) = \max\{\tau(p) \mid p \in \text{IDEAL}(x)\}.$$

The block number of $\bot$ is not defined.

Definition 3 (Ideal Tree)**.** *The* ideal tree of L with respect to τ *is a tree on the elements of L in which the set of block numbers on the path from x to the root is equal to $\{\tau(p) \mid p \in IDEAL(x)\}$. The ideal tree is denoted TREE(L).*[1]

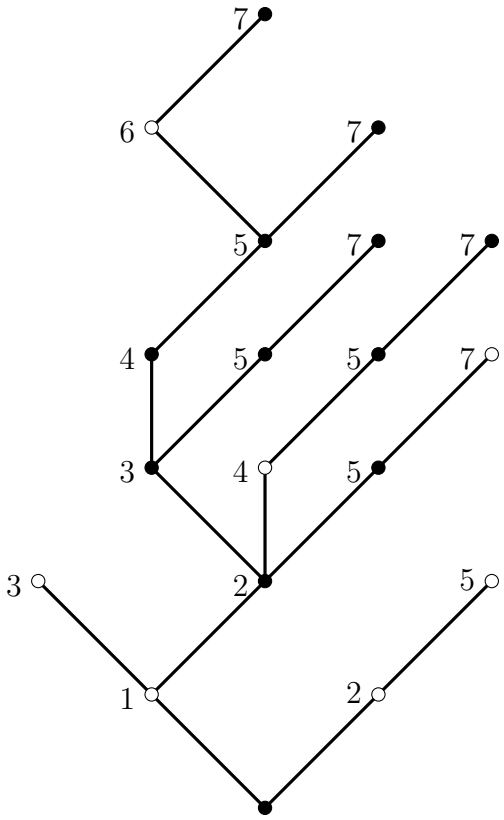


Figure 2.1: The ideal tree for the distributive lattice in Figure 1.2. The linear extension τ is given in Figure 1.3.

Proposition 4. *The ideal tree has the following properties.*

- $\perp$ *is the root of* TREE(L)*, and* TREE(L) *is a spanning tree of* TRG(L)*.*
- *The height of the ideal tree is exactly* m*.*
- *The ideal tree is unique up to* τ*. Hence, a lattice may have many different ideal trees, one for each linear extension of its join-irreducible elements.*

[1] In [12], the block numbers are assigned to edges rather than vertices. As well, our description of the ideal tree is "upside-down" relative to their definition. These changes are convenient for our purposes and the differences are superficial.

- *Along any path from the root to a node, the block numbers are strictly increasing.*
- *The children of any node have distinct block numbers.*

The ideal tree can be used to compute both IDEAL and IDEAL^{-1} in $O(m)$ time. One obtains $\text{IDEAL}(x)$ by following parent pointers from x to the root and recording the sequence of block numbers along the path. The set of block numbers encodes $\text{IDEAL}(x)$ by identifying $\tau(p)$ with p for all $p \in \mathcal{J}(L)$.

One can also compute $\text{IDEAL}^{-1}(I)$ for an ideal I of $\mathcal{J}(L)$ in $O(m)$ time, assuming that I is given as the sequence $\{\tau(p) \mid p \in I\}$ sorted in increasing order. The unique path in $\text{TREE}(L)$ beginning at the root and having block numbers $\{\tau(p) \mid p \in I\}$ leads to the desired node. By Proposition 4, the block numbers must appear in increasing order, and at each node along the path there can only be one child with a given block number. Thus, the node can be located by starting at the root and repeatedly moving to the child with the next block number in I.

If the children pointers in each node of the ideal tree are stored in increasing order, then this search can be performed in only $O(m)$ time. At each node, the children must be visited in increasing order until the child with the desired block number is found, and then the search can proceed from that child. By this method, the sequence of block numbers seen over the entire search must be a strictly increasing sequence, and hence only $O(m)$ children can be visited in total.

Theorem 5 (Habib and Nourine [12])**.** *The ideal tree computes meets and joins in $O(m)$ time. TREE(L) requires $O(n \log n)$ space, and can be constructed from TRG(L) in $O(n \log n)$ time.*[2]

2.2 A Faster Data Structure

We now develop a method of improving the efficiency of ideal trees that reduces the time for meet and join operations to $O(\log n)$ while staying within $O(n \log n)$ space.

Our objective is to relate the time for operations to the *width* of $\mathcal{J}(L)$, rather than its size. Recall that the width of a poset is the size of its largest antichain. The following lemma demonstrates that the width is preferable to the size of $\mathcal{J}(L)$ in many cases.

[2]More precisely, the preprocessing time is $O(|E|)$, where E is the edge set of $\text{TRG}(L)$, but $|E|$ always lies between $n-1$ and $n \lg n$ for distributive lattices.

Lemma 6. *For all distributive lattices L with n elements, width$(\mathcal{J}(L)) \leq \lg(n)$.*

Proof. Consider the poset $\mathcal{P}$ which consists of w non-comparable elements; that is, $\mathcal{P}$ is a single antichain of size w. Every subset of the w elements is, trivially, an ideal of the poset. Hence the ideal lattice $\mathcal{O}(\mathcal{P})$ contains exactly 2^w elements.

Now fix a distributive lattice L of size n and let $w = \text{width}(\mathcal{J}(L))$. By definition $\mathcal{J}(L)$ contains an antichain of size w as a subposet. It is easily seen that a poset has at least as many ideals as any of its subposets; hence $n \geq 2^w$ and $w \leq \lg(n)$. □

As with the ideal tree, we wish to quickly move between L and the ideals of $\mathcal{J}(L)$ to compute meets and joins. However, we cannot afford to look at an entire ideal of $\mathcal{J}(L)$ during any meet or join operation, since the ideals may have up to $m = |\mathcal{J}(L)|$ elements. To avoid this, we decompose $\mathcal{J}(L)$ into $w = \text{width}(\mathcal{J}(L))$ disjoint chains and identify each ideal by the largest elements of each chain that lie in that ideal. Since ideals are downward closed, everything below these chainwise largest elements must lie in the ideal. This lets us operate only on sets of size w or less. We now formalize this idea and see how to implement it.

2.2.1 Chain Decompositions and Wavefronts

As in the last section, we make use of a linear extension τ of $\mathcal{J}(L)$, and for all $x \in L$ we define $\text{BLOCK}(x) = \max\{\tau(p) \mid p \in \text{IDEAL}(x)\}$. We abuse the notation by identifying $p \in \mathcal{J}(L)$ with $\tau(p) \in \{1, 2, \ldots, m\}$, and extend all functions on $\mathcal{J}(L)$ to operate on $\{1, 2, \ldots, m\}$ as well.

A chain decomposition of a poset is a partition of the elements into disjoint sets $C_1, C_2, \ldots, C_k$ such that the elements within each C_i are totally ordered in the poset. By Dilworth's Theorem [7], every poset of width w admits a decomposition into exactly w chains.

Fix a chain decomposition $\mathcal{C} = \{C_1, C_2, \ldots, C_w\}$ of $\mathcal{J}(L)$, where w is the width of L. Define a function $\text{CHAIN}\colon \mathcal{J}(L) \to \{1, \ldots, w\}$ such that $p \in C_{\text{CHAIN}(p)}$ for all $p \in \mathcal{J}(L)$. The arrangement in Figure 2.2 has $\text{CHAIN}(1) = \text{CHAIN}(3) = 1$, $\text{CHAIN}(4) = \text{CHAIN}(6) = 2$, and $\text{CHAIN}(2) = \text{CHAIN}(5) = \text{CHAIN}(7) = 3$.

Since a chain is totally ordered, min and max are well-defined functions on the non-empty subsets of a chain. Note that min and max are well-behaved with respect to τ, since

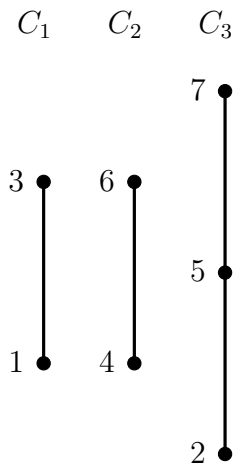


Figure 2.2: One possible minimal chain decomposition of the join-irreducible poset in Figure 1.3.

for $\emptyset \subsetneq S \subseteq C_i$ the minimum element of S with respect to chain order also has the smallest integer value $\tau(p)$ among all $p \in S$. The analogous property holds for max.

We identify an ideal by the set of *chainwise maximum* elements of I, which we call the *wavefront* of the ideal.

Definition 7. *The* wavefront *of an ideal I of $\mathcal{J}(L)$ with respect to $\mathcal{C}$ is defined by*

$$\mathcal{W}(I) = \{\max(C_i \cap I) \mid C_i \cap I \neq \emptyset, 1 \leq i \leq w\}.$$

The wavefronts, rather than the ideals, will be our way of identifying and manipulating lattice elements.

Fortunately, union and intersection of ideals translate easily to the language of wavefronts. Observe that the wavefront of the union of two ideals is the chainwise maximum of their wavefronts, which contains precisely the largest element in each chain that appears in either wavefront. Similarly the wavefront of their intersection is the chainwise minimum of their wavefronts.

We make this precise: Write **max** for chainwise maximum and **min** for chainwise minimum. Then for any ideals I_1 and I_2, $\mathcal{W}(I_1 \cup I_2) = \mathbf{max}(\mathcal{W}(I_1), \mathcal{W}(I_2))$ and $\mathcal{W}(I_1 \cap I_2) = \mathbf{min}(\mathcal{W}(I_1), \mathcal{W}(I_2))$.

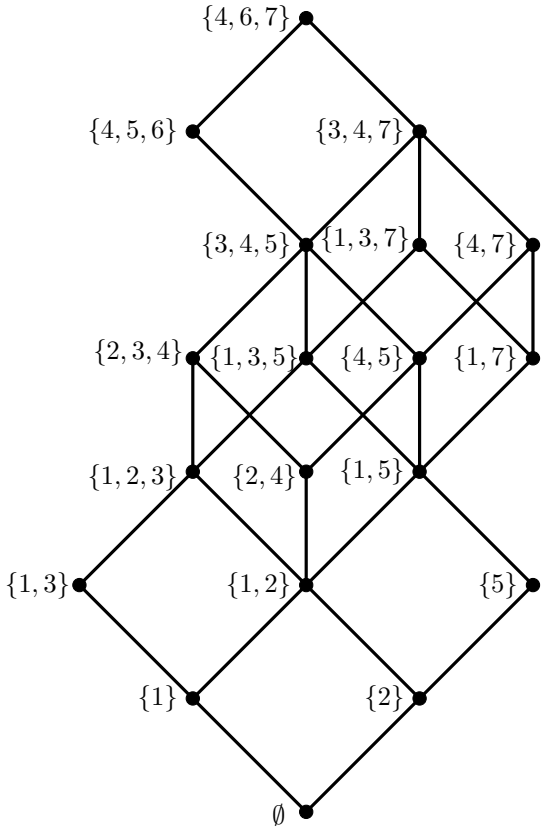


Figure 2.3: The lattice in Figure 1.2 labelled with wavefronts instead of ideals. The chain decomposition is that of Figure 2.2.

For computational purposes, we represent a wavefront $W \subseteq \mathcal{J}(L)$ by the string with characters $\{\tau(p) \mid p \in \mathcal{W}(I)\}$ sorted in decreasing order. For example, the top node in Figure 2.3 has wavefront $\{4, 6, 7\}$ so it is represented by the string 764. This representation allows efficient computation of **max** and **min**:

Lemma 8. *Given wavefronts W_1 and W_2, the chainwise maximum* $\mathbf{max}(W_1, W_2)$ *and chainwise minimum* $\mathbf{min}(W_1, W_2)$ *can be computed in $O(w)$ time.*

Proof. First, merge the wavefronts into a list of length $|W_1| + |W_2|$ sorted in decreasing order. As the strings representing W_1 and W_2 are sorted this way initially, merging takes $O(w)$ time. For each chain C_i, $1 \leq i \leq w$, the chainwise maximum contains only the

largest element of C_i in this combined list; since it is sorted in decreasing order, this is the element of C_i that occurs earliest in the list.

To compute $\mathbf{max}(W_1, W_2)$, walk the list in order and, for each chain C_i, keep track of whether an element of C_i has already appeared in the list. If an element of C_i has already appeared, then any other element of C_i encountered can be deleted. The resulting list only contains the first element of each chain that appeared in the original list, and hence represents the chainwise maximum. Similarly, the chainwise minimum consists of the elements in each chain that occur last in the list. Hence, $\mathbf{min}(W_1, W_2)$ can be computed by walking the list in reverse and deleting elements as before. □

The price of using wavefronts is that a chain decomposition of $\mathcal{J}(L)$ must be computed as part of the preprocessing for our data structure. This can be done in $O(m^{2.5})$ time.

Proposition 9. *A minimal chain decomposition of $\mathcal{J}(L)$ can be computed in $O(m^{2.5})$ time.*

Proof. Let G be a bipartite graph with vertex set $\mathcal{J}(L) \times \{\ell, r\}$ that includes an edge $\{(p, \ell), (q, r)\}$ whenever $p < q$ in $\mathcal{J}(L)$. A maximum matching M on G can be computed in $O(m^{2.5})$ time using the Hopcroft-Karp algorithm [13].

This matching induces a partition of $\mathcal{J}(L)$ by putting p and q in the same group whenever $\{(p, \ell), (q, r)\} \in M$. We shall see that this partition is a minimal chain decomposition of $\mathcal{J}(L)$.

As M is a matching, two elements p and q lie in the same group if and only if there is a sequence of elements $p_1, p_2, \ldots, p_k \in \mathcal{J}(L)$ such that M contains the edges $\{(p, \ell), (p_1, r)\}$, $\{(p_1, \ell), (p_2, \ell)\}$, $\ldots$, $\{(p_{k-1}, \ell), (p_k, r)\}$, and $\{(p_k, \ell), (q, r)\}$. By construction, having these edges in G implies that $p < p_1 < p_2 < \cdots < p_k < q$. Thus, each group of elements must be totally-ordered in $\mathcal{J}(L)$. Moreover, any unmatched vertex (p, ℓ) must be the maximum element of its chain, and any unmatched vertex (q, r) must be the minimum element of its chain. Hence the number of unmatched nodes in $\mathcal{J}(L) \times \{\ell\}$ (or in $\mathcal{J}(L) \times \{r\}$) is equal to the number of chains. As a maximum matching must minimize the number of unmatched nodes on either side of the bipartition, it follows that M induces a decomposition of $\mathcal{J}(L)$ into a minimum number of chains. □

2.2.2 Strategy

The chain decomposition will make it possible to use wavefronts instead of ideals in our computations while essentially using the same strategy described in Section 1.2. Let us restate this strategy in terms of the *lattice of wavefronts* of $\mathcal{J}(L)$.

Define WAVE$:= \mathcal{W} \circ$ IDEAL. In other words,

$$\text{WAVE}(x) = \{\max(C_i \cap \text{IDEAL}(x)) \mid C_i \cap \text{IDEAL}(x) \neq \emptyset, 1 \leq i \leq w\}.$$

The function WAVE is an isomorphism between L and the lattice of wavefronts, in which **min** and **max** are the meet and join operations, respectively. Thus, for all $x, y \in L$,

$$x \vee y = \text{WAVE}^{-1}(\mathbf{max}(\text{WAVE}(x), \text{WAVE}(y))),$$

$$x \wedge y = \text{WAVE}^{-1}(\mathbf{min}(\text{WAVE}(x), \text{WAVE}(y))).$$

Since **min** and **max** can be computed in $O(w)$ time by Lemma 8, it only remains to compute WAVE and WAVE^{-1} in comparable time. To this end, we develop two data structures, one that computes WAVE in $O(w)$ time, and one that computes WAVE^{-1} in $O(\log n)$ time. This yields the main theorem of this chapter.

Theorem 2 (Restated)**.** *There is a data structure for distributive lattices that requires $O(n \log n)$ bits of space and supports meet and join operations in $O(\log n)$ time. It can be constructed in $O(n \log n + m^{2.5})$ time from TREE(L) or TRG(L).*

The $O(m^{2.5})$ term in the preprocessing time comes from computing a chain decomposition of $\mathcal{J}(L)$ as described in Proposition 9. Otherwise, the data structure can be created in $O(n \log n)$ time.

2.2.3 Computing WAVE

We extend the ideal tree data structure to allow access to WAVE(x) in $O(w)$ time for each $x \in L$. Consider the path in TREE(L) from a node x to the root. This path contains all the block numbers of IDEAL(x), but only a subset of the block numbers lie in the wavefront.

Our goal is to traverse the path from x to the root, looking at all of the nodes whose block numbers lie in the wavefront and bypassing all of the others. This motivates the following definition.

Definition 10. *Consider a path in TREE(L) with nodes $x_k, x_{k-1}, \ldots, x_1, x_0$, where $x_0 = \perp$ is the root. Let the* short path *of x_k be the subpath which contains x_0, and for $i \geq 1$, contains x_i if and only if BLOCK$(x_i) \in$ WAVE(x_k). Denote the short path of x_k by SHORT(x_k).*

We wish to traverse the short path of a node instead of the full path in TREE(L). Since the short path has length at most $w+1$, this will reduce the time to compute WAVE from $O(m)$ to $O(w)$ as desired.

One obvious representation of the short path is a linked list beginning at x_k and ending at x_0, with the intermediate nodes in decreasing order by their block numbers. This is indeed a possible solution; however, it requires $O(n \log^2 n)$ bits of space to store the short paths of all the nodes as separate linked lists, and we do not wish to use so much space. We reduce the space with the observation that the difference between the short path of each node and the short path of its *parent* in TREE(L) is always small.

Lemma 11. *Let $x, y \in L$ where x is the parent of y in TREE(L). Then SHORT(y) can be obtained from SHORT(x) by adding y and possibly deleting one element.*

Proof. Since x is the parent of y in the ideal tree, we have IDEAL(y) = IDEAL(x) $\cup \{y\}$. Moreover, $y \in$ WAVE(y) since the block numbers are increasing from the root in TREE(L)

Let $i =$ CHAIN(BLOCK(y)). If WAVE(x) $\cap C_i = \emptyset$, then WAVE(y) = WAVE(x) $\cup \{y\}$. In this case, SHORT(y) is obtained from SHORT(x) by adding y.

Otherwise, there is a node $z \in$ SHORT(x) with CHAIN(BLOCK(z)) $= i$. Since BLOCK(y) is larger than BLOCK(z) and they lie in the same chain, WAVE(y) = WAVE(x)$\setminus$ {BLOCK(z)} $\cup$ {BLOCK(y)}. Hence, SHORT(y) is obtained from SHORT(x) by adding y and removing z. □

This lemma shows that the total space required to store the wavefronts is $O(n \log n)$ if we only store the differences between a node and its parent. Then a short list could be reconstructed whenever it is needed by inserting and deleting elements according to these differences. However, this is not a realistic strategy as there could be as many as m insertions and deletions in this reconstruction, even though the short list has only $w+1$ elements at the end.

We wish to store all the short lists explicitly as linked lists, so that any one of the lists can be accessed and traversed in $O(w)$ time, while staying within $O(n \log n)$ bits of space. Persistence will make this possible; see Section 1.2.2 for a summary of persistence.

2.2.4 Persistent Ideal Tree

Instead of storing the short lists separately, we represent them as different versions of the same *fully persistent list*. This list has n versions, one for each element of L, and we write "Version x" for the list corresponding to SHORT(x).

Definition 12. *The* persistent ideal tree *of L with respect to τ is a fully persistent linked list on L in which Version x stores the short path of x. The persistent ideal tree is denoted PTREE(L).*

Although PTREE(L) resembles a list more than it does a tree, the structure of the ideal tree is essentially contained in it. In fact, the version tree of PTREE(L) has exactly the same structure as TREE(L).

Proposition 13. *The persistent ideal tree can be used to compute WAVE(x) in $O(w)$ time for all $x \in L$. PTREE(L) can be constructed in $O(nw)$ time from the ideal tree and the chain decomposition of $\mathcal{J}(L)$. It occupies $O(n \log n)$ bits of space.*

Proof. To compute WAVE(x) for an element x, walk Version x of the list and record the sequence of block numbers; the result is WAVE(x) sorted in decreasing order. This can be done in constant time per step, as if each list were stored independently, for a total of $O(w)$ time.

The construction is straightforward using Lemma 11. First create Version $\bot$, a list containing only $\bot$. For all other nodes $y \in L$, Lemma 11 shows how to construct Version y inductively. Let x be the parent of y in TREE(L). Version y is obtained from Version x by adding node y to the front of the list, then scanning the list for an element z such that CHAIN(BLOCK(y)) = CHAIN(BLOCK(z)). If z is found, then it is deleted from Version y; that is, the predecessor of z is updated to point at the successor of z. Note that every list can only have one element belonging to each chain, so there can only be one such z. If the list does not contain any element in the same chain as BLOCK(y), then no further action is taken.

Node y can be attached to the front of the list in $O(1)$ time, and z can be scanned for and deleted in $O(w)$ amortized time. Hence, each version can be constructed in $O(w)$ amortized time, and PTREE(L) can be constructed from TREE(L) in $O(nw)$ time. As each pointer update adds $O(\log n)$ bits of space, the total space required is $O(n \log n)$ bits. □

Unfortunately, unlike the ideal tree, the persistent ideal tree is not effective for computing WAVE^{-1}. To do so, one would need to match a given wavefront with the block numbers of a single version of the persistent ideal tree, and it is not clear how to do this in less than $O(m)$ time. We solve this with a second data structure which can be constructed from PTREE(L).

2.2.5 Computing WAVE^{-1}

We now see how to recover a lattice element from its wavefront. One can view WAVE(x) as a string over the alphabet $\{0, 1, \ldots, m\}$ of length at most w. Define WCODE(x) to be the string with characters $\{\tau(p) \mid p \in \text{WAVE}(x)\}$ sorted in decreasing order; for example, the code for an element x with WAVE(x) $= \{3, 4, 7\}$ is the string WCODE(x) $= 743$.

Computing WAVE^{-1} is thereby reduced to solving a version of the static dictionary problem in which the keys are strings of length w. Each element of L has an associated key string that is determined by its wavefront, and a search for that key in the dictionary must simply identify its lattice element.

This dictionary problem has been solved, in some variations, by ternary search trees [1]. We include our own solution, which uses very similar ideas, to justify the small space requirements. Since the common solutions to this problem allow insertion and deletion of strings, they cannot use less space in the worst case than is required to store all of the key strings explicitly; this alone will usually require more than $O(n \log n)$ bits of space. Our solution sacrifices insertions and deletions, which we do not need, to let the entire structure occupy $O(n \log n)$ bits of space. As well, our solution only supports successful searches; there are no guarantees about what it returns if the query string is not stored in the set. Fortunately, all of the searches performed by our meet and join operations will be for query strings corresponding to existing wavefronts.

Lemma 14. *Let S be a set of n strings of length k over the alphabet $\{1, \ldots, n\}$. For any function $f\colon S \to \{1, \ldots, n\}$, there is a data structure that takes $O(n \log n)$ bits of space and computes $f(s)$ in $O(k + \log n)$ time.*

Proof. Suppose the strings of S are listed out explicitly in lexicographic order, and we are searching for a particular query string $a_1 a_2 \ldots a_k$ in S. Let $S(i)$ denote the ith string in the sorted list.

The algorithm performs a binary search on each character independently as follows. First, a_1 is compared with c, the first character of $S(\lfloor n/2 \rfloor)$: If $a_1 < c$ then the list is reduced to the strings whose first character is strictly less than c, and if $a_1 > c$ then the list is reduced to the strings whose first character is strictly greater than c. In either case, the search recurses on the reduced list.

If $a_1 = c$, then the list is reduced to the interval of strings that begin with c, and the search moves on to search in the reduced range for a string whose second character is a_2. The search repeats recursively until all the characters have been matched, at which point

there can only be one string that has not been eliminated from the list, and it must be an exact match for the query string.

This search is fast. With every comparison, either a character a_i is matched and removed from consideration, or a_i is not matched with c and the list is reduced by at least half. There are exactly k character matches in a successful search, and the list can only be halved $\lceil \lg n \rceil$ times before it is reduced to a single string. Therefore the total number of character comparisons in the search is $O(k + \log n)$.

Unfortunately, we cannot afford to list the strings explicitly as that requires $O(kn \log n)$ bits. To reduce the space, the search can be represented as a ternary decision tree, in which each node corresponds to one of the comparisons made during the search. To indicate what comparison should be done, each node must store a *comparison character* c and an *index* i. A node can have up to three children corresponding to the three outcomes of the comparison (either $a_i < c$, $a_i > c$, or $a_i = c$). We call these the $<$-child, $>$-child, and $=$-child of the node. A node may have fewer than three children if an option is unreachable in the decision tree by any string in S, but every node must have an $=$-child by construction. The decision tree has n leaves in bijection with the strings of S, hence the values of f can be stored in the leaves of the tree.

The decision tree can be represented in $O(\log n)$ bits per node. However, there may be more than $O(n)$ nodes in a tree. With one modification to the decision tree, it is possible to ensure that the number of nodes is at most $2n$: Any node that has only one child must be deleted from the tree, since any query string in S that reaches that node can only pass to its $=$-child. Clearly this can only reduce the search time and the size of the tree.

We now show that there are at most n non-leaf nodes in the decision tree using a charging argument. Observe that every node has either a $<$-child or a $>$-child, and they all must have an $=$-child. If a node has a $<$-child, then call it a $<$-node; otherwise, call it a $>$-node.

A node with comparison character c and index i is only reached after the first $i - 1$ characters $b_1 b_2 \ldots b_{i-1}$ have been determined implicitly by the decision tree. Define the *prefix* of the node to be the string $b_1 b_2 \ldots b_{i-1}$. Observe that every node has a different prefix. Moreover, if there is a node with prefix p in the decision tree then there must be at least two strings in S beginning with p, for otherwise the node would have no $<$-child or $>$-child.

Charge every $<$-node to the (lexicographically) first string in S that begins with the prefix of that node, and charge every $>$-node to the (lexicographically) last string in S that begins with the prefix of that node.

Claim. No string in the list can be charged more than once.

First, we show that <-node and a >-node cannot charge the same string. To a contradiction, suppose a string $b_1b_2 \dots b_k$ is charged by a <-node x with prefix $b_1b_2 \dots b_{i-1}$, and also by a >-node y with prefix $b_1b_2 \dots b_{j-1}$. Then $b_1b_2 \dots b_k$ is the first string in S beginning with $b_1b_2 \dots b_{i-1}$, and also the last string beginning with $b_1b_2 \dots b_{j-1}$. We cannot have $i = j$ since no two nodes have the same prefix. If $i < j$, then there is only one string in S beginning with $b_1b_2 \dots b_{j-1}$. Similarly, if $j < i$ then there is only one string beginning with $b_1b_2 \dots b_{i-1}$. We have a contradiction in either case, as there must be at least two strings beginning with any prefix of a node in the decision tree.

Now suppose a string $b_1b_2 \dots b_k$ is charged by two <-nodes: node x with prefix $b_1b_2 \dots b_{i-1}$ and node y with $b_1b_2 \dots b_{j-1}$, where $i < j$. Then $b_1b_2 \dots b_k$ is the first string in S beginning with $b_1b_2 \dots b_{i-1}$, and also the first beginning with $b_1b_2 \dots b_{j-1}$. Since y has a <-child, there must exist a string in S beginning with $b_1b_2 \dots b_{j-1}c$, where c is less than the comparison character of y. Yet this string occurs before $b_1b_2 \dots b_k$ lexicographically and begins with $b_1b_2 \dots b_{i-1}$, contradicting the assumption that $b_1b_2 \dots b_k$ is charged by x.

The case for two >-nodes follows by symmetry. Thus the claim has been proven.

This establishes a surjection from the internal nodes of the modified decision tree to S; hence, the modified decision tree contains at most n non-leaf nodes. There is also a leaf for each string s, which just stores the value $f(s)$. Therefore, the decision tree has at most $2n$ nodes. Each node requires $O(\log n)$ bits of space to store the comparison character, the index, and the pointers to its children, for a total of $O(n \log n)$ bits of space. □

We now apply this data structure to compute the function defined by $f(\text{WCODE}(x)) = x$. The persistent ideal tree easily facilitates the construction of the decision tree, but first the nodes must be sorted lexicographically by wavefront. We require two additional lemmas to sort the nodes.

Define CODE(x) to be the string of block numbers on the path from x to the root in TREE(L), or equivalently, the string with characters $\{\tau(p) \mid p \in \text{IDEAL}(x)\}$ sorted in decreasing order. For example, the code for an element x with $\text{IDEAL}(x) = \{1, 3, 4, 5, 7, 9\}$ is the string $\text{CODE}(x) = 975431$. Recall that WCODE(x) is the string of block numbers along the short path of x in decreasing order.

Lemma 15. *The lexicographic order of L by WCODE is the same as the lexicographic order by CODE.*

Proof. Let $x, y \in L$ with $\text{CODE}(x) < \text{CODE}(y)$ lexicographically. Say $\text{CODE}(x) = a_1a_2 \cdots a_ix_1x_2 \cdots x_j$ and $\text{CODE}(y) = a_1a_2 \cdots a_iy_1y_2 \cdots y_k$. Then $x_1 < y_1$, and hence there

is an element $p \in \mathrm{IDEAL}(y) \setminus \mathrm{IDEAL}(x)$ where $\tau(p) = y_1$. Moreover, $\tau(p)$ is the largest among all $p \in \mathrm{IDEAL}(y) \setminus \mathrm{IDEAL}(x)$.

We show that p must be in $\mathrm{WAVE}(y)$. Suppose otherwise; then there must be some $q \in \mathrm{IDEAL}(y) \cap C_{\mathrm{CHAIN}(p)}$ with $q \geqslant p$. Since $p \notin \mathrm{IDEAL}(x)$, we have $q \notin \mathrm{IDEAL}(x)$; hence $\tau(q)$ does not appear in $a_1 a_2 \cdots a_i$. But $\tau(q) > \tau(p)$, so $\tau(q)$ does not appear in $y_1 y_2 \cdots y_k$ either. Thus, $\tau(q)$ does not appear in $\mathrm{CODE}(y)$ and $q \notin \mathrm{IDEAL}(y)$. This yields a contradiction, so $p \in \mathrm{WAVE}(y)$.

In particular, $p \in \mathrm{WAVE}(y) \setminus \mathrm{WAVE}(x)$, and $\tau(p)$ is the first character of $\mathrm{WCODE}(y)$ that differs from $\mathrm{WCODE}(x)$. Since $\tau(p) > x_1$ it follows that $\mathrm{WCODE}(x) < \mathrm{WCODE}(y)$ lexicographically. □

Lemma 16. *Beginning with TREE(L), the nodes can be sorted by CODE in $O(n)$ time.*

Proof. The sorting algorithm is a kind of bucket sort. Create m lists, $\mathcal{L}_1, \ldots, \mathcal{L}_m$, which are initially empty; throughout the algorithm, $\mathcal{L}_j$ exclusively stores elements in block j. For each child x of $\perp$, add x to list $\mathcal{L}_{\mathrm{BLOCK}(x)}$. Then, for $j = 1, 2, \ldots, m$, walk $\mathcal{L}_j$ in order and add all of the children of $x \in \mathcal{L}_j$ to the appropriate lists for their block numbers. After every list has been traversed, we claim that the concatenation $\{\perp\}\mathcal{L}_1\mathcal{L}_2 \ldots \mathcal{L}_m$ must contain every element of L sorted lexicographically by ideal.

First, every node must be added to its list. Since block numbers are strictly increasing along every path leaving the root, the children of a node in $\mathcal{L}_j$ cannot be added to an earlier list $\mathcal{L}_i$, $i \leq j$; hence, if the algorithm visits a node it must also visit each of its children eventually. Thus when the algorithm reaches the end of $\mathcal{L}_m$ every node must have been visited.

Moreover, each list will be sorted lexicographically by ideal. This is trivially true of $\mathcal{L}_1$ which can only have one element. It then follows by induction for $\mathcal{L}_j$, $j \geq 2$, since the order within $\mathcal{L}_j$ is inherited from the parents of its elements in the concatenation $\{\perp\}\mathcal{L}_1\mathcal{L}_2 \ldots \mathcal{L}_{j-1}$. That is, for two elements x and y in block j, the algorithm adds x to $\mathcal{L}_j$ before y if and only if $\mathrm{parent}(x)$ appear before $\mathrm{parent}(y)$ in $\{\perp\}\mathcal{L}_1\mathcal{L}_2 \ldots \mathcal{L}_{j-1}$. This produces the correct sorted order because for each $x \in \mathcal{L}_j$, $\mathrm{CODE}(x) = j\mathrm{CODE}(\mathrm{parent}(x))$. Hence, the concatenation $\{\perp\}\mathcal{L}_1\mathcal{L}_2 \ldots \mathcal{L}_m$ gives the lexicographic order by ideal. □

These two lemmas show that L can be sorted by WCODE in $O(n)$ time.

Proposition 17. *There is a data structure that computes $WAVE^{-1}$ in $O(\log n)$ time, occupies $O(n \log n)$ bits, and can be constructed from the persistent ideal tree in $O(n \log n)$ time.*

Proof. Let S be the set of all wavefronts of lattice elements, expressed as strings of block numbers in decreasing order. Each has length at most w; pad the end of each string with zeros so that they have length exactly w.

Consider WAVE^{-1} as a function from S to L. The decision tree structure of Lemma 14 for S computes WAVE^{-1} in $O(\log n + w)$ time, which is $O(\log n)$ by Lemma 6, and it requires $O(n \log n)$ bits of space. We show that the decision tree can be constructed in $O(n \log n)$ time, beginning with $\text{PTREE}(L)$.

First, the n persistent nodes of $\text{PTREE}(L)$ must be sorted lexicographically by their wavefronts, corresponding to the sorted order in S. By Lemmas 15 and 16, the nodes can be sorted by WCODE in $O(n)$ time using $\text{TREE}(L)$.

Once the nodes are sorted by wavefront, the decision tree can be constructed by following the procedure in Lemma 14. The algorithm is restated here in terms of the persistent ideal tree.

Maintain a pointer for each wavefront, initially pointing at the first node of each short list in the persistent tree. The decision tree can be constructed as described in Lemma 14: Choose the middle element x of the sorted list of lattice nodes, and create a node in the decision tree with index 1 and character $\text{BLOCK}(x)$. Walk the sorted list of nodes in both directions to find the range of lattice elements with that block number. This divides the list into three pieces: he list of nodes with smaller block number, the list with the same block number, and the list with larger block number. Recurse on the first piece and the last piece, and assign the roots of the resulting decision trees to be the $<$-child and $>$-child, respectively, of the current node. For the list with the same block number as the middle element, advance all the pointers to the next node in the short list (this takes constant time per node), and then recurse on the sublist, now comparing their block numbers to the next index of the search string.

If it ever happens that all the nodes in the list have the same block number (i.e., the first and last pieces are empty), then delete the current node of the decision tree entirely, advance the pointers in each short list, and recurse on the same list with the next index. This has the effect of omitting any node in the decision tree that has only an $=$-child. The resulting decision tree is the data structure described in Lemma 14. Since each node of $\text{PTREE}(L)$ is considered only a constant number of times for each index from 1 to w, the construction time is $O(n \log n)$. □

Remark 18. *This procedure above maintains pointers to multiple nodes in PTREE(L), and moves them along their short lists at different rates. Given that we have not explicitly described the fully persistent list structure at the heart of the persistent ideal tree, it may*

not be obvious that multiple lists can be accessed at the same time. As long as no updates to the structure occur, the persistent structure of [8] allows for complete independence in the accesses. There are no updates to the persistent list in Proposition 17.

This concludes the description of our data structure for distributive lattices. Theorem 2 follows immediately from Lemma 6, Proposition 9, Proposition 13, and Proposition 17.

Chapter 3

Compact Encoding of Distributive Lattices

Counting the number of distributive lattices on n elements up to isomorphism is a long-standing and difficult problem. In 2002, Erné, Heitzig, and Reinhold placed bounds on these numbers with the following theorem.

Theorem 19. *[Enumeration Bounds [9]] Let d_n be the number of non-isomorphic distributive lattices on n elements. Then $1.81^{n-4} < d_n < 2.46^{n-1}$ for all $n \geq 4$, and $1.84^n < d_n < 2.39^n$ eventually.*

According to these bounds, we have $0.88n < \lg d_n < 1.257n$ for sufficiently large n; hence, it is possible to distinguish between all the distributive lattices on n elements using as few as $1.257n$ bits. It is natural to ask if there is a reasonable representation of distributive lattices that uses so little space. By "reasonable", we mean that there should be efficient methods to convert between this compression and a natural representation of the lattice such as its transitive reduction graph.

The techniques used to obtain Theorem 19 do not seem to directly yield such a representation. The authors invent a canonical form for distributive lattices which identifies each lattice by a sequence of integers, but they rely on careful analysis of generating functions to count the number of possible sequences. We make several refinements to this technique that allow for a compact encoding of the sequence itself.

In this chapter, we first define a compression method that encodes a distributive lattice in $\frac{10}{7}n + O(\log n)$ bits. As $\frac{10}{7} \approx 1.429$, the size of this encoding is already within a small

constant factor of the optimal. We then compress this further using arithmetic coding to reduce the size to approximately $1.25899n + o(n)$ bits. Our compression can be efficiently constructed from $\mathrm{TRG}(L)$ and vice versa.

3.1 Preliminaries

As in Chapter 2, let L be a distributive lattice on n elements and let $m = |\mathcal{J}(L)|$. Fix a linear extension τ of $\mathcal{J}(L)$, and identify $p \in \mathcal{J}(L)$ with its position in τ.

Recall that an element y *covers* an element x (or x is *covered* by y) if $x < y$ and for all $z \in L$, $x \leqslant z < y \implies x = z$.

Let $C(x)$ be the set of elements that cover x.

Lemma 20. *An element $x \in L$ is join-irreducible if and only if it covers a unique element of L.*

Proof. Since L is finite, every element except for $\perp$ covers at least one element of L, and $\perp$ is not join-irreducible by definition. If $x \in L$ is join-irreducible, it cannot cover any two distinct elements y and z, for then $y \vee z = x$. If x covers a unique element $x' \in L$, then for all $y, z < x$ we have $y, z \leq x'$. Hence $y \vee z \leq x'$ whenever $y, z < x$. Since there is no intermediate element between x' and x, we can only have $y \vee z = x$ when $y = x$ or $z = x$. □

3.2 Doubling Lattices

The encoding is an application of a special case of the doubling method for lattices, originally due to Alan Day [5] and cleverly exploited in [9] to count distributive lattices. We use it to construct a distributive lattice by a sequence of m "doubling" operations, each adding one join-irreducible element to the lattice.

We define the operation DOUBLE that accepts a distributive lattice L and an element $x \in L$ and outputs a larger lattice L' that is obtained by creating a new join-irreducible element x' that covers x, and "completing" the lattice with this new element. That is, L' is constructed to be the smallest distributive lattice that contains $L \cup \{x'\}$.

Definition 21. *Let $x \in L$, and let $y_1, y_2, \ldots, y_k$ be the elements of L greater than or equal to x. Then DOUBLE(L, x) adds k new elements $y'_1, y'_2, \ldots, y'_k$ to L and extends the order relation $\leqslant$ as follows:*

- *For* $1 \leq s, t \leq k$, $y'_s \leqslant y'_t \iff y_s \leqslant y_t$
- *If* $z \in L$, *then* $z \leqslant y'_s \iff z \leqslant y_s$.

See Figure 3.1 for an example of the double operation.

The double operation has an intuitive effect on the transitive reduction graph of L. TRG(DOUBLE(L, x)) is obtained from TRG(L) as follows:

- Beginning at node x, follow upward edges to visit all y reachable from x in TRG(L). Call these nodes $y_1, y_2, \ldots, y_k$.
- Create new nodes $y'_1, y'_2, \ldots, y'_k$. Add an edge (y_s, y'_s) for $s = 1, 2, \ldots, k$. Also add an edge (y'_s, y'_t) whenever (y_s, y_t) is in TRG(L), for $1 \leq s, t \leq k$.

From the definition, it is easy to check that the resulting DAG accurately represents DOUBLE(L, x). This procedure takes time proportional to the number of edges between elements of $\{y_1, y_2, \ldots, y_k\}$ in TRG(L).

We now prove that DOUBLE does actually produce a distributive lattice, and that it modifies $\mathcal{J}(L)$ in a predictable way.

Lemma 22. *The poset DOUBLE(L, x) is a distributive lattice, and $\mathcal{J}$(DOUBLE(L, x)) = $\mathcal{J}(L) \cup \{x'\}$, where x' is the newly-created element covering x.*

Proof. As in the definition, let $y_1, y_2, \ldots, y_k$ be the elements of $\{y \in L \mid y \geqslant x\}$, and let $y'_1, y'_2, \ldots, y'_k$ be the duplicate elements created by DOUBLE(L, x).

Let $\mathcal{P}$ be the poset that extends $\mathcal{J}(L)$ with an element q, and extends the order relation of $\mathcal{J}(L)$ by the rule $q \geqslant p$ if and only if $p \in$ IDEAL(x). We know that $\mathcal{O}(\mathcal{P})$ (the lattice of ideals ordered by inclusion) is a distributive lattice, and that it contains $\mathcal{O}(\mathcal{J}(L)) \cong L$ as a sublattice since $\mathcal{J}(L)$ is a subposet of $\mathcal{P}$. We show that $y'_1, y'_2, \ldots, y'_k$ correspond to the ideals in $\mathcal{O}(\mathcal{P}) \setminus \mathcal{O}(\mathcal{J}(L))$, and that $y'_1, y'_2, \ldots, y'_k$ relate to L in the same way as the ideals in $\mathcal{O}(\mathcal{P}) \setminus \mathcal{O}(\mathcal{J}(L))$ relate to $\mathcal{O}(\mathcal{J}(L))$.

Clearly, the ideals of $\mathcal{O}(\mathcal{P}) \setminus \mathcal{O}(\mathcal{J}(L))$ are exactly the ideals containing q. Let $I \in \mathcal{O}(\mathcal{P})$ and suppose $q \in I$. Then IDEAL$(x) \subseteq I$ since $q \geqslant p$ for all $p \in$ IDEAL(x) (that is, IDEAL$(x) \leqslant I$ in $\mathcal{O}(\mathcal{P})$). Since q is a maximal element of $\mathcal{P}$, the set $I \setminus \{q\}$ is also an ideal of $\mathcal{P}$, and even an ideal of $\mathcal{J}(L)$ since $q \notin I \setminus \{q\}$. In particular, there is an element $y \in L$ such that IDEAL$(y) = I \setminus \{q\}$, and since $I \setminus \{q\}$ must still contain IDEAL(x) as a subset we have $y \geqslant x$ in L. Thus, $\mathcal{O}(\mathcal{P}) \setminus \mathcal{O}(\mathcal{J}(L)) = \{\text{IDEAL}(y) \cup \{q\} \mid y \geqslant x\}$.

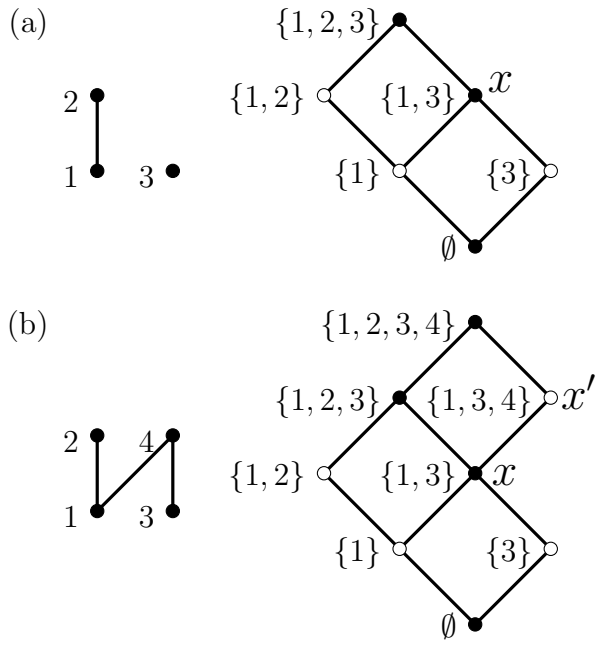


Figure 3.1: (a) A distributive lattice L and its join-irreducible poset. (b) The result of DOUBLE(L, x).

Define $\Phi\colon L \cup \{y'_1, y'_2, \ldots, y'_k\} \to \mathcal{O}(\mathcal{P})$ by $\Phi(z) = \text{IDEAL}(z)$ for $z \in L$, and by $\Phi(y'_s) = \text{IDEAL}(y_s) \cup \{q\}$ for $s = 1, \ldots, k$. Since IDEAL is an isomorphism between L and $\mathcal{O}(\mathcal{J}(L))$, we only need to show that $y'_1, y'_2, \ldots, y'_k$ have the same order relations in DOUBLE(L, x) as $\Phi(y'_1), \Phi(y'_2), \ldots, \Phi(y'_k)$ have in $\mathcal{O}(\mathcal{P})$.

Observe the following for $1 \le s, t \le k$:

- $\text{IDEAL}(y_s) \cup \{q\} \subseteq \text{IDEAL}(y_t) \cup \{q\} \iff \text{IDEAL}(y_s) \subseteq \text{IDEAL}(y_t)$.
- If $z \in L$, $\text{IDEAL}(z) \subseteq \text{IDEAL}(y_s) \cup \{q\} \iff \text{IDEAL}(z) \subseteq \text{IDEAL}(y_s)$.

If we rewrite these rules using Φ, it becomes clear that they are exactly those stated in Definition 21.

- $\Phi(y'_s) \subseteq \Phi(y'_t) \iff \Phi(y_s) \subseteq \Phi(y_t)$.
- If $z \in L$, $\Phi(z) \subseteq \Phi(y'_s) \iff \Phi(z) \subseteq \Phi(y_s)$.

Therefore, Φ is an isomorphism between DOUBLE(L, x) and $\mathcal{O}(\mathcal{P})$ extending IDEAL. The result now follows by Birkhoff's representation theorem. □

This lemma has an interesting consequence. Since DOUBLE adds a single join-irreducible element to the lattice, an appropriate sequence of m double operations applied to a trivial lattice is sufficient to create any distributive lattice with m join-irreducible elements. In fact, since the elements of $\mathcal{J}(L)$ can be added in different orders, there may be many ways to build a given lattice. The only restriction on the double operation is that the element added to $\mathcal{J}(L)$ must be maximal in that poset; therefore, the double operations can add join-irreducible elements in the order of any linear extension of $\mathcal{J}(L)$ and the resulting lattice must be isomorphic to L.

3.3 Compression Strategy

Now consider $C(x) \cap \mathcal{J}(L)$, the set of join-irreducible elements that cover x. By Lemma 20, every join-irreducible element is present in exactly one set $C(x) \cap \mathcal{J}(L)$ for $x \in L$. Hence, $\sum_{x \in L} |C(x) \cap \mathcal{J}(L)| = m$.

In fact, storing $|C(x) \cap \mathcal{J}(L)|$ for each $x \in L$ is sufficient to uniquely identify a distributive lattice up to isomorphism. The number $|C(x) \cap \mathcal{J}(L)|$ is the number of times that DOUBLE must be applied to x in order to grow L from the trivial lattice, simply because doubling on x adds one new join-irreducible element to the lattice and that element covers x.

This is the basis of our encoding: we store $|C(x) \cap \mathcal{J}(L)|$ for each x, ordered in a predictable way. This sequence of values serves as an instruction set for building the lattice by doubling.

Let $x_1, x_2, \ldots, x_n$ be the elements of L, sorted in some fashion to be decided later. The *covering sequence* of L is the sequence $(c_1, c_2, \ldots, c_n)$ where $c_i = |C(x_i) \cap \mathcal{J}(L)|$. The encoding we present is a space-efficient representation of the covering sequence.

Definition 23. *The* compression code *of a lattice with covering sequence* $(c_1, c_2, \ldots, c_n)$ *is the bitstring* $1^{c_1}01^{c_2}0 \ldots 1^{c_n}0$.

Simply put, the compression code is the covering sequence written in unary. Note that the compression code of L occupies $n + m$ bits as it contains exactly n zeros and $\sum_{x \in L} |C(x) \cap \mathcal{J}(L)| = m$ ones. It is trivial to convert between this compression code and the covering sequence in linear time.

In the worst case this uses $2n-1$ bits, which occurs when L is totally ordered. However, the worst case behaviour can be improved by accounting for some expensive cases. Later we will define the *improved compression code* and prove the main theorem of this chapter:

Theorem 24. *The improved compression code of L occupies at most $\frac{10}{7}n + O(\log n)$ bits. It contains exactly n zeros and no more than $\frac{3}{7}n$ ones, along with $O(\log n)$ bits of extra information. This compression can be created from $TRG(L)$ in $O(n \log n)$ time and it can be decompressed to $TRG(L)$ in $O(n \log n)$ time.*

Our final result is a simple corollary of this theorem. It uses arithmetic coding [3] to compress the improved compression code to a size equal the entropy lower bound plus a lower order term[1].

Corollary 25. *There exists a code representing for distributive lattices that uses at most $(-\lg(0.7) - \frac{3}{7}\log(0.3))n + o(n) \approx 1.25899n + o(n)$ bits. Converting between this representation and the improved compression code requires $O(n)$ time.*

Proof. The improved compression code is a binary string in which a zero occurs with probability at least $\frac{n}{10n/7} = 0.7$ and a one occurs with probability at most $\frac{3n/7}{10n/7} = 0.3$. Arithmetic coding of this string reduces the length to the ceiling of entropy lower bound, given by

$$\lceil (\#\text{ zeros})(-\lg(\text{probability of zero})) + (\#\text{ ones})(-\lg(\text{probability of one})) \rceil$$

$$= \lceil -n\lg(0.7) - \frac{3}{7}n\log(0.3) \rceil \approx 1.25899n$$

An extra $O(\log n)$ bits are required to store the length of the code.

However, ordinary arithmetic coding is not very efficient algorithmically. To obtain fast encoding and decoding times, we first split the improved compression code into pieces of size $\frac{\log n}{2}$ and code them separately. An extra $O(\log \log n)$ bits are required for each piece, but this only adds $O(\frac{n \log \log n}{\log n})$ bits overall. By precomputing the map between all possible $\frac{\log n}{2}$-bit strings and their encodings, each piece can be encoded or decoded in linear time. □

[1] Thanks to Patrick Nicholson for this suggestion.

3.4 Intuitive Algorithm

The lattice can be recovered from its covering sequence by the following decompression algorithm, which will be made precise later.

1: **procedure** DECOMPRESS($c_1, c_2, \ldots, c_n$)
2: Set K to the trivial lattice with element $\perp$
3: Create a list $\mathcal{L}$ containing $\perp$
4: **for** $i = 1, 2, \ldots, n$ **do**
5: $x \leftarrow \mathcal{L}[i]$
6: **for** $j = 1, 2, \ldots, c_i$ **do**
7: $K \leftarrow \text{DOUBLE}(K, x)$
8: Insert new nodes $y'_1, y'_2, \ldots, y'_k$ into $\mathcal{L}$
9: Return K

This algorithm reads the covering sequence as a list of instructions for building the lattice. The algorithm scans the list $\mathcal{L}$ one node at a time and, for each node x that it visits, it pops a number from the covering sequence and applies DOUBLE to the existing lattice that many times with node x. All the new nodes created by the double operations are inserted into $\mathcal{L}$ (in some unspecified way) so that they will be visited eventually by the algorithm. An example of this algorithm is shown in Figure 3.2.

As long as the final order of nodes in $\mathcal{L}$ is the "same" as that of the covering sequence (i.e., the order $x_1, x_2, \ldots, x_n$), the nodes of K will be correctly matched with their corresponding nodes in L and the correct values in the covering sequence. In that case, the double operations must correctly reproduce L; this can be shown using Lemma 22.

To make this algorithm precise, it remains to decide on a protocol for ordering the covering sequence. The encoding algorithm uses the protocol to decide on the order of the covering sequence, and the decoding algorithm uses it to decide the order that nodes are inserted into $\mathcal{L}$.

Describing and computing this protocol is the chief difficulty of the encoding procedure, even though many different protocols would work equally well. The only requirement of the protocol is that it can be easily computed by the decoding algorithm, which cannot know the final lattice structure until the end. The protocol serves as an agreement between the encoding and decoding algorithms that ensures correctness while allowing for a compact representation of the covering sequence.

One possible protocol works according to the following rule, expressed in terms of line 8 of DECOMPRESS: Insert $y'_1, y'_2, \ldots, y'_k$ at the end of $\mathcal{L}$ in the same respective order as

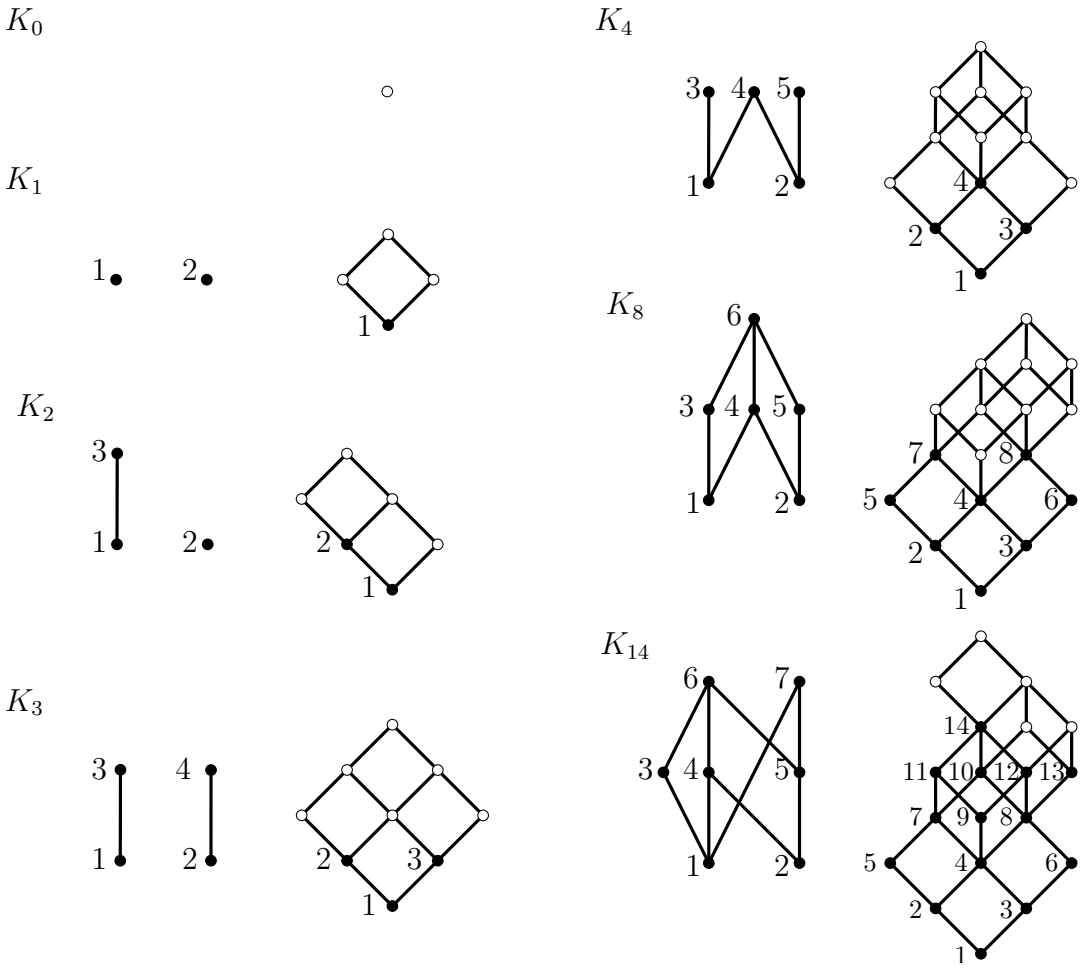


Figure 3.2: Growing the distributive lattice of Figure 1.2 by a sequence of double operations. The numbers give the order in which the nodes are visited, and open circles indicate unvisited nodes. Each image shows the result of one DOUBLE operation. The covering sequence for this lattice is $(2, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$.

$y_1, \ldots, y_k$ appear in the list. That is, $y'_1, \ldots, y'_k$ are inserted after all the preexisting nodes, and y'_s appears before y'_t if and only if y_s appears before y_t in $\mathcal{L}$.

We adopt a similar protocol, except that this one sorts first by height and only uses the above protocol to decide on the order between nodes of the same height. We say the height of $x \in L$ is the length of the shortest path from $\perp$ to x in $\mathrm{TRG}(L)$. Write $h(x)$ for

the height of x.

> **Rule:** Keep $\mathcal{L}$ sorted by height. When adding new nodes to $\mathcal{L}$, for $s = 1, \ldots, k$, insert y'_s at the end of the nodes of height $h(y'_s)$. If $h(y'_s) = h(y'_t)$, then the order is inherited from y_s and y_t in $\mathcal{L}$.

This rule is baked into the formal algorithm in the next section.

3.5 Decompression Algorithm

Proposition 26. *TRG(L) can be computed from its covering sequence in $O(n \log n)$ time.*

Proof. We give a formal description of the decompression algorithm. For each $x \in L$, the algorithm applies the double operation with x as its second argument $|C(x) \cap \mathcal{J}(L)|$ times, and it applies the double operations in the order determined by the **Rule** above. It keeps a set of $m + 1$ lists, $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_m$, in which $\mathcal{L}_j$ contains the nodes of K with height j. Initially $\mathcal{L}_0$ contains only $\perp$, and the other lists are empty.

Each node stores its height, denoted $h(x)$. As well, each node x is assigned an *index* $\ell(x) \in \{1, \ldots, n\}$ that stores the position of x in its list, $\mathcal{L}_{h(x)}$.

The algorithm works as follows, beginning with a covering sequence $(c_1, \ldots, c_n)$. For $i = 1, \ldots, n$, let x be the ith item in $\mathcal{L}_0\mathcal{L}_1 \cdots \mathcal{L}_m$. Repeat the following c_i times:

i Replace K by DOUBLE(K, x).

ii Find the newly created nodes $y'_1, y'_2, \ldots, y'_k$ and their counterparts $y_1, y_2, \ldots, y_k$. Sort these nodes such that $h(y_1) \leq h(y_2) \leq \cdots \leq h(y_k)$ and $\ell(y_s) < \ell(y_{s+1})$ whenever $h(y_s) = h_(y_{s+1})$. Set $h(y'_s) = h(y_s) + 1$ for $s = 1, \ldots, k$.

iii For $s = 1, \ldots, k$ in order, set $\ell(y'_s) = |\mathcal{L}_{h(y'_s)}| + 1$ and append y'_s to the end of $\mathcal{L}_{h(y'_s)}$.

Since each double operation runs in time proportional to the number of new edges it creates, the total time for the double operations is $O(|E|)$ where E is the edge set of TRG(L). It is known that $|E| \leq n \lg n$ for distributive lattices. Sorting $y_1, y_2, \ldots, y_k$ by height and index in step (ii) takes $O(k \log k)$ time, and everything else takes $O(k)$ time. Thus, the total time for the algorithm is $O(n \log n)$.

It only remains to prove that this algorithm reproduces L faithfully. Since the height of y'_s is always larger than that of x (for all $s = 1, \dots k$), a node is never appended to a list $\mathcal{L}_j$ that has already been traversed. Moreover, every node is added to a list exactly once since this only occurs when the node is created. This shows that every node will eventually be visited by the algorithm.

By Lemma 22, K is a distributive lattice at the beginning and end of every step. There are m double operations executed during the algorithm, so K has m join-irreducible elements at the end.

The most critical ingredient of this algorithm is the certainty that the values in the covering sequence are correctly matched with the nodes in K. This must be true because the order of the covering sequence is based on **Rule**, which in turn is described in terms of this very algorithm. In other words, since this algorithm is used to decide the order of the covering sequence, we can be certain the order is correct.

Hence, if we assume that (at some point in the algorithm) K is a correctly structured sublattice of L, then the nodes in K being processed by the algorithm must be accurately identified with the numbers in the covering sequence. Each double operation then adds a join-irreducible element to K, extending $\mathcal{J}(K)$ to a larger subposet of $\mathcal{J}(L)$. The algorithm must therefore terminate with $\mathcal{J}(K) = \mathcal{J}(L)$. □

3.6 Compression Algorithm

We now give an algorithm that efficiently builds the covering sequence of L from its ideal tree. Our algorithm first computes $|C(x) \cap \mathcal{J}(L)|$ for each $x \in L$ and then puts them in the order the covering sequence.

Proposition 27. *The covering sequence of L can be computed from $TRG(L)$ in $O(n \log n)$ time.*

Proof. First, compute $|C(x) \cap \mathcal{J}(L)|$ for all $x \in L$. By Lemma 20, an element x' of L is join-irreducible if and only if it covers a unique element x. Thus, the values of the covering sequence can be computed as follows: Initialize $c_y = 0$ for all $y \in L$, and then visit all join-irreducible elements $x' \in \mathcal{J}(L)$ and increment c_x, where x is the unique element covered by x'.

To order these values, simply run the decoding algorithm, beginning with a trivial lattice, while maintaining an injective map ϕ between the in-progress lattice K and the

completed lattice L that embeds K as a sublattice of L. When the decoding algorithm "asks" for a value in the covering sequence to associate with a node $x \in K$, supply it with the value $c_{\phi(x)}$. Now ϕ can be extended by identifying the new join-irreducible element of K with one of the elements in $C(\phi(x)) \cap \mathcal{J}(L) \setminus \phi(K)$.

Note that this method works with any protocol for ordering the compression code, not just the one we have chosen.

Finding the values of the covering sequence requires $O(|E|)$ time, where E is the set of edges in $\mathrm{TRG}(L)$. Ordering the values takes the same time as running the decompression algorithm, which is $O(n \log n)$. Thus, the total time is $O(n \log n)$. □

3.7 Improving the Space Requirements

The encoding representation above uses $n + m$ bits because the values in the covering sequence are represented in unary. In the worst case, this requires $2n - 1$ bits. This occurs when the lattice is totally ordered, and the compression code is the bitstring $10101010 \cdots 010$. Similarly inefficient examples can be constructed by adding sufficiently long chains above or below any distributive lattice; the result will always be a distributive lattice, and the compression code will begin or end with the string $10101010 \cdots 010$.

These examples are so inefficient because the decoding process repeatedly applies the double operation to the top element of the lattice K, and only one new node is created by each operation. We can avoid this inefficient case by storing the value of n along with the compression. This allows the decoding algorithm to detect when L has been completely generated without simply reaching the end of the covering sequence. If L has not been completely generated, then the decompression algorithm is guaranteed to double at least once more on the current lattice K, and if it reaches the last unvisited element of K then it knows — without reading the next bit of the encoding — that it must double at least once on this final element.

We can use this observation to omit some ones from the encoding. Suppose that, at some point during the decoding algorithm, L has not been completely generated and x is the last unvisited node in K. Then the algorithm must apply DOUBLE at least once; hence, the first 1 in the unary representation for $|C(x) \cap \mathcal{J}(L)|$ can be omitted from the compression and still be "read" implicitly by the decompression algorithm. The compression and decompression algorithms can be easily modified to handle these implicit ones.

With this improvement, all of the ones in the compression code for the totally ordered lattice become implicit; $10101010\cdots010$ is replaced by $000\cdots00$. See Figure 3.3 for another example. More generally, every one in the compression code causes a double operation that creates at least two new nodes, and hence there are at most $n/2$ ones in the encoding. Combined with the n zeros, the compression code has length at most $1.5n$. The space is therefore reduced to $1.5n + O(\log n)$ bits in the worst case. The $O(\log n)$ term reflects the space required to store n.

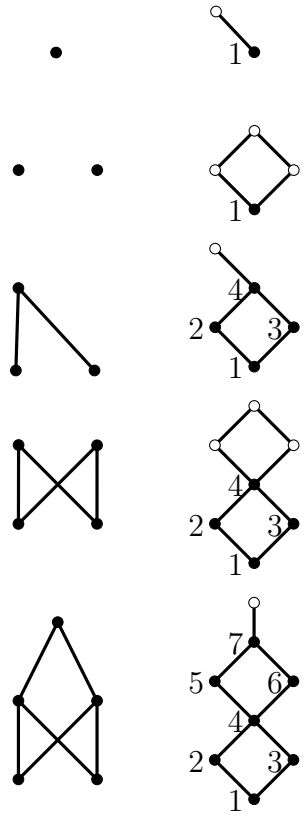


Figure 3.3: An example of decoding the lattice with covering sequence $(2, 0, 0, 2, 0, 0, 1, 0)$. The improved compression code is the bitstring 1000100000, in which three ones are omitted compared to the simple compression code. The numbers give the order in which the nodes are visited (as determined by **Rule**), and open circles indicate unvisited nodes.

A second, more complex improvement reduces the space to just $\frac{10}{7}n + O(\log n)$ bits. Just as the first improvement deals with the case when DOUBLE is applied to the top

node of K, this improvement deals with the cases where DOUBLE is applied to nodes just below the top node. In this case, and only this case, a double operation produces exactly two new nodes. This is a bad case; we wish for every 1 in the compression code to be responsible for the creation of at least $\frac{7}{3} = 2.\overline{333}$ new nodes, on average. If this is achieved, then the total number of ones in the improved compression will be at most $\frac{3}{7}n \approx 0.429n$. Combined with the n zeros in the code, this gives the $\frac{10}{7}n$ bit code we desire.

Suppose that at some point in the decompression algorithm, every node of K has been visited except for the top node and the nodes immediately below it. Let $\top_K$ be the top node of K, and let $x_1, x_2, \ldots, x_d$ be the nodes just below it. Note that $d \geq 2$, for if $d = 1$ then $\top_K$ must have been created by a double operation on x_1 and we assume that x_1 has not been visited yet.

Consider all of the double operations that will be applied to $x_1, x_2, \ldots, x_d$. If two or more double operations are applied to $x_1, x_2, \ldots, x_d$, then the first double operation creates exactly two nodes, and all of the following double operations create at least three nodes, since the distance from the top of the lattice increases to three after the first operation. Hence, a sequence of i double operations on $x_1, x_2, \ldots, x_d$ creates at least $3i-1$ new nodes. For $i \geq 2$, this implies that each double operation creates at least 2.5 ($\geq \frac{7}{3}$) new nodes on average. Thus, we can restrict our attention to the case where only one double operation is applied to $x_1, x_2, \ldots, x_d$.

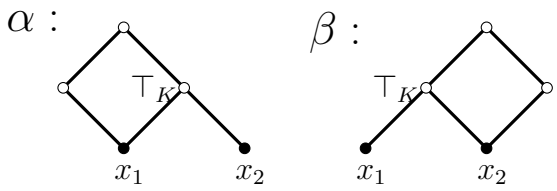
We can also eliminate the case where $d \geq 3$. Suppose that double is applied just once on $x_1, x_2, \ldots, x_d$; without loss of generality, assume it is applied to x_1. Although DOUBLE(K, x_1) only creates two new nodes, the fact that $d \geq 3$ implies that the *most recent* double operation, which created K from some smaller lattice K', must have created at least four nodes. We prove this in the following claim.
Claim: Suppose $d \geq 3$ and let $z \in K'$ such that DOUBLE$(K', z) = K$. Then $|\{y \in K' \mid y \geqslant z\}| \geq 4$.

Suppose otherwise. One can easily check that $\{y \in K' \mid y \geqslant z\}$ is a sublattice of K', as the upper set of any element of any lattice is always a lattice. The only lattices with one, two, or three elements are totally ordered; hence this sublattice has no two elements with the same height. The double operation makes a copy of $\{y \in K' \mid y \geqslant z\}$ and attaches it to K', increasing the height of each element by one. Hence, only of the new nodes can be just below $\top_K$ in K. The only other node that can be on that row is $\top_{K'}$. Thus, $d \leq 2$, a contradiction.

This implies that the total number of new nodes created by the *two* double operations (DOUBLE(K', z) and DOUBLE(K, x_1)) is at least six, and thus each one creates an average of at least 3 nodes. This exceeds $\frac{7}{3}$, as required.

We have not altered our encoding yet, we have only done analysis. We now make a change in how the covering sequence is encoded in order to handle the remaining "bad" cases, in which $d = 2$ and only one double operation is applied to x_1 and x_2.

We have eliminated every case except for the two cases pictured below, which we call α and β. Case α doubles on x_1 and β doubles on x_2. They correspond to the bit strings 100 and 010 respectively; these are the pieces of the compression code that are read while processing x_1 and x_2.



We also consider case γ representing the bit string 00, in which double is not applied to x_1 or x_2.

The algorithm may encounter cases α, β, and γ any number of times while decoding the compression code. These cases can be detected easily: Every time the decoding algorithm reaches the second highest row of K, it checks if there are only three unvisited nodes in K. If so, it reads a few bits ahead in the compression code to see if α, β, or γ cases occur. Every time α occurs, it appears in the code as the string 100, β appears as 010, and γ appears as 00. Of course, a compression code may contain substrings 100, 010, and 00 in places that do not correspond to these α, β, and γ cases; these cases only occur when the corresponding strings are associated with exactly two nodes on the second highest row of the lattice.

Our modification swaps these strings, so that the most frequent of these three cases is represented by the string 00, while the other two are represented by 100 and 010. For example, when the decoding algorithm detects one of these special cases, it may read the string for γ, but act as if it had read the string for α. This has the significant advantage that the shortest of these codes is the most common, thus minimizing the total number of bits across all the occurrences of these cases. Along with the bitstring code and the value of n, the improved compression code stores two extra bits to indicate whether α, β, or γ is the most frequent case. This describes our final encoding of L.

Definition 28. *Let $(c_1, c_2, \ldots, c_n)$ be the covering sequence for L. Then the simple compression code for L is $1^{c_1}01^{c_2}0\cdots1^{c_n}0$. The* improved compression code *modifies the simple compression code in two ways:*

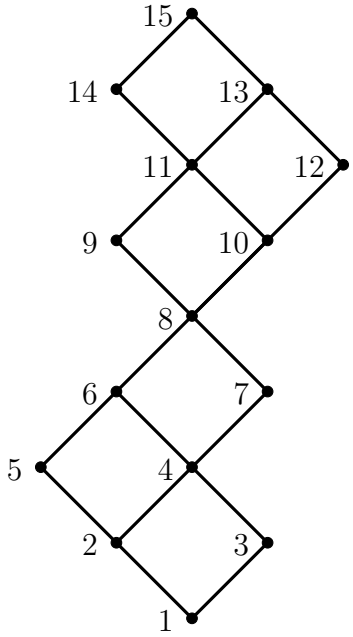


Figure 3.4: A distributive lattice with covering sequence $(2, 1, 0, 1, 0, 0, 0, 2, 0, 1, 1, 0, 0, 0, 0)$. The numbers in the figure indicate the order that the nodes are visited in the decompression algorithm. The simple compression code is (1)1010010000(1)10010100000, where the parenthesized ones would be omitted by the first improvement.

- *Any 1 in the code that corresponds to an operation* $DOUBLE(K, \top_K)$ *in the decompression algorithm is omitted. The value of* n *is stored along with the code.*
- *All of the occurrences of* α *(with string* 100*),* β *(with string* 010*), and* γ *(with string* 00*) are located. The most common of these cases swaps its string with* 00*. A number between 1 and 3 is stored with the code to indicate whether* α*,* β*, or* γ *is the most frequent case.*

See Figure 3.4 for an example. As α is the most frequent case, the simple compression code is changed into the improved code as shown here:

$$110\,\underbrace{100}_{\alpha}\,\underbrace{100}_{\alpha}\,\underbrace{00}_{\gamma}\;110\,\underbrace{010}_{\beta}\,\underbrace{100}_{\alpha}\,\underbrace{00}_{\gamma}\,0$$

$$10\,\overbrace{00}^{\alpha}\,\overbrace{00}^{\alpha}\,\overbrace{100}^{\gamma}\;10\,\overbrace{010}^{\beta}\,\overbrace{00}^{\alpha}\,\overbrace{100}^{\gamma}\,0$$

By omitting ones that can be made implicit, and by swapping the strings for α and γ, the compression code becomes 10000010010010001000.

The changes to the encoding and decoding algorithms are straightforward. It is easy to detect the α, β, and γ cases during decoding, and then to interpret the code correctly by swapping cases as needed.

We now prove that the improved code uses at most $\frac{10}{7}n$ bits. Let f_α, f_β, and f_γ be the number of occurrences of the three cases.

In α and β, two new nodes are created and the algorithm moves to the next row; there are no guarantees about what doubling occurs later in the algorithm. In γ, however, the algorithm must proceed to apply double to $\top_K$ at least once, for otherwise it would visit all the nodes of K without doubling. The only exception is when $K = L$, and this case is not significant since it occurs only once. If $\top_K$ is doubled $i \geq 1$ times, this reflects $i - 1$ consecutive ones in the compression code (since the first 1 is implicit) and it creates $2^i - 1$ new nodes in the lattice (since the ith DOUBLE operation creates 2^{i-1} new nodes).

For all $i \geq 1$, it is true that $(2^i - 1) - \frac{2}{3} \geq \frac{7}{3}(i - 1)$, and we have equality for $i = 2$. But we only require $(2^i - 1) \geq \frac{7}{3}(i - 1)$ to ensure that every 1 in the code creates at least $\frac{7}{3}$ nodes on average. This extra "two-thirds of a node" can be charged to the γ case that occurred just before. Although the γ case does not double x_1 or x_2, and therefore does not create any new nodes directly, we can assume that γ "creates" $\frac{2}{3}$ of a node by the doubling that must follow.

Thus, the total number of nodes created across all occurrences of α, β, and γ is $2f_\alpha + 2f_\beta + \frac{2}{3}f_\gamma$. We now show that the ratio of the number of created nodes to the number of ones in the improved code associated with α, β, or γ is at least $\frac{3}{7}$. The total number of ones in the compression code depends on whether α, β, or γ is the most common case.

If $f_\alpha \geq f_\beta, f_\gamma$, then the number of ones in the compression code is $f_\beta + f_\gamma$. In this case, we have

$$\frac{2f_\alpha + 2f_\beta + \frac{2}{3}f_\gamma}{f_\beta + f_\gamma} \geq \frac{(\frac{1}{3}f_\beta + \frac{5}{3}f_\gamma) + 2f_\beta + \frac{2}{3}f_\gamma}{f_\beta + f_\gamma} = \frac{7}{3}.$$

Similarly, if $f_\beta \geq f_\alpha, f_\gamma$ then the number of ones in the compression code is $f_\alpha + f_\gamma$. Then

$$\frac{2f_\alpha + 2f_\beta + \frac{2}{3}f_\gamma}{f_\alpha + f_\gamma} \geq \frac{2f_\alpha + (\frac{1}{3}f_\alpha + \frac{5}{3}f_\gamma) + \frac{2}{3}f_\gamma}{f_\alpha + f_\gamma} = \frac{7}{3}.$$

Finally, if $f_\gamma \geq f_\alpha, f_\beta$ then the number of ones in the compression code is $f_\alpha + f_\beta$. Then

$$\frac{2f_\alpha + 2f_\beta + \frac{2}{3}f_\gamma}{f_\alpha + f_\beta} \geq \frac{2f_\alpha + 2f_\beta + (\frac{1}{3}f_\alpha + \frac{1}{3}f_\beta)}{f_\alpha + f_\beta} = \frac{7}{3}.$$

In all three cases the ratio of nodes created to number of ones in the compression code is at most $\frac{7}{3}$. Thus, the total number of ones in the compression code is at most $\frac{3}{7}n$. As the number of zeros is still n, we conclude that the compression code uses $\frac{10}{7}n$ bits. As well, these modifications can only reduce the length of the code, so $m+n$ is still an upper bound. We have now proved Theorem 24.

Chapter 4

Conclusion and Open Problems

This thesis has introduced two novel representations of finite distributive lattices. The first is the persistent ideal tree data structure that performs meet and join operations in $O(\log n)$ time while staying space efficient. To my knowledge, it is more efficient than any known data structure for distributive lattices using less space than a simple $O(n^2 \log n)$ lookup table.

However, the construction time of $O(n \log n + m^{2.5})$ for the persistent ideal tree is high, owing exclusively to the chain decomposition of $\mathcal{J}(L)$ that must be computed. I wonder whether this expenditure can be avoided.

The second representation is a compact encoding of the lattice using kn bits of space where comes very close to the information-theoretic lower bound for representing distributive lattices. One might ask whether the space requirements of the compression code can be reduced much further; such a result may yield a new upper bound on the number of distributive lattices of size n.

Finally, a natural question is whether there exists a data structure that has the best of both worlds: It occupies $O(n)$ bits of space while supporting meet and join operations efficiently. It seems unlikely that the persistent ideal tree could be modified to require less space, or that the compression code could be modified to support meet and join operations, so perhaps an entirely new structure is needed to solve this problem.

References

[1] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369. Society for Industrial and Applied Mathematics, 1997.

[2] Garrett Birkhoff. Rings of sets. *Duke Mathematical Journal*, 3(3):443–454, 1937.

[3] John G. Cleary, Radford M. Neal, and Ian H. Witten. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[4] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[5] Alan Day. A simple solution to the word problem for lattices. *Canad. Math. Bull*, 13:253–254, 1970.

[6] Karel De Loof, Hans De Meyer, and Bernard De Baets. Exploiting the lattice of ideals representation of a poset. *Fundamenta Informaticae*, 71(2, 3):309–321, 2006.

[7] Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, pages 161–166, 1950.

[8] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[9] Marcel Erné, Jobst Heitzig, and Jürgen Reinhold. On the number of distributive lattices. *The Electronic Journal of Combinatorics*, 9(1):23, 2002.

[10] Arash Farzan and J. Ian Munro. Succinct representation of finite abelian groups. In Barry M. Trager, editor, *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, pages 87–92. ACM, 2006.

[11] Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Applied Mathematics*, 110(2):169–187, 2001.

[12] Michel Habib and Lhouari Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165(2):391–405, 1996.

[13] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[14] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989.

[15] Daniel J. Kleitman and Kenneth J. Winston. The asymptotic number of lattices. *Annals of Discrete Mathematics*, 6:243–249, 1980.

[16] J. Ian Munro. Succinct data structures ... potential for symbolic computation? In Sergei A. Abramov, Eugene V. Zima, and Xiao-Shan Gao, editors, *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*, pages 5–8. ACM, 2016.

[17] J. Ian Munro and S. Srinivasa Rao. Succinct representation of data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2004.

[18] J. Ian Munro and Corwin Sinnamon. Time and space efficient representations of distributive lattices. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 550–567. Society for Industrial and Applied Mathematics, 2018.

[19] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016.

[20] Lhouari Nourine and Olivier Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71(5-6):199–204, 1999.