Order 3 (1987), 331–344.

Free Lattice Algorithms

RALPH FREESE
Department of Mathematics, University of Hawaii, Honolulu, HI 96822, U.S.A.

Communicated by G. Grätzer

(Received: 11 March 1985; accepted: 5 November 1986)

Abstract. In the late 1930s Phillip Whitman gave an algorithm for deciding for lattice terms v and u if $v \leqslant u$ in the free lattice on the variables in v and u. He also showed that each element of the free lattice has a shortest term representing it and this term is unique up to commutivity and associativity. He gave an algorithm for finding this term. Almost all the work on free lattices uses these algorithms. Building on the work of Ralph McKenzie, J. B. Nation and the author have developed very efficient algorithms for deciding if a lattice term v has a *lower cover* (i.e., if there is a w with w covered by v, which is denoted by $w \prec v$) and for finding them if it does. This paper studies the efficiency of both Whitman's algorithm and the algorithms of Freese and Nation. It is shown that although it is often quite fast, the straightforward implementation of Whitman's algorithm for testing $v \leqslant u$ is exponential in time in the worst case. A modification of Whitman's algorithm is given which is polynomial and has constant minimum time. The algorithms of Freese and Nation are then shown to be polynomial.

AMS subject classifications (1980). 06B25, 68C25.

Key words. Free lattice, cover, algorithm.

0. Introduction

In the late 1930s Phillip Whitman gave an algorithm for deciding for lattice terms v and u if $v \leqslant u$ in the free lattice on the variables in v and u. He also showed that each element of the free lattice has a shortest term representing it and this term is unique up to commutivity and associativity. He gave an algorithm for finding this term. Almost all the work on free lattices uses these algorithms. Building on the work of Ralph McKenzie, J. B. Nation and the author have developed very efficient algorithms for deciding if a lattice term v has a *lower cover* (i.e., if there is a w with w covered by v, which is denoted by $w \prec v$) and for finding them if it does. We have been able to implement Whitman's algorithms and our own algorithms on a microcomputer using muLISP, a fast version of LISP for microcomputers well suited to manipulating symbolical mathematical expressions. This program has proved useful in studying certain problems associated with free lattices, see [8].

This paper studies the efficiency of these algorithms. In the second section we study the time complexity of Whitman's algorithm itself, since all of the other algorithms depend on it. This problem has been studied by computer

scientists who were able to show that there is a polynomial time algorithm for deciding if $v \leqslant u$ [13]. We show that the straightforward implementation of Whitman's algorithm is exponential. The algorithm of Hunt, Rosenkrantz, and Bloniardz [13] has the disadvantage that its minimum time is rather long, and thus it is often slower that Whitman's algorithm. We show how to modify Whitman's algorithm so as to have the advantages of both algorithms. Somewhat surprisingly there is a simple modification of Whitman's condition (W) which greatly improves the speed of Whitman's algorithm.

In the third section we give an overview of the results about covers in free lattices and show that the algorithms of [10] have polynomial time complexity.

In fourth section we give a brief introduction to the programming language LISP and how our lattice theory program is written in LISP. In the last section we make a few remarks about the expected time of Whitman's algorithm.

1. Preliminaries

A lattice term is either a generator (i.e., a variable) or formally a join or meet of simpler terms called respectively *joinands* and *meetands*. We allow the join and meet operation to have an arbitrary finite number of arguments, so that $x \vee y \vee z$ is a valid term. We define the *type* of a term v, denoted type(v), to be either $\vee$ or $\wedge$ or 'gen' depending on whether v is formally join or a meet or a generator. For terms v and u we define the *value* of (v, u), denoted val(v, u), to be true if $v \leqslant u$ and false otherwise. Whitman's algorithm for determining val(v, u) is recursively defined as follows, where v_i and u_j denote typical subterms of v and u:

(1) If type$(v) =$ gen,
  (i) if type$(u) =$ gen, then val$(v, u) =$ true iff $v = u$.
  (ii) else if type$(u) = \wedge$, then val$(v, u) =$ true iff val$(v, u_j) =$ true for each meetand u_j of u.
  (iii) [otherwise type$(u) = \vee$] val$(v, u) =$ true iff val$(v, u_j) =$ true for some joinand u_j of u.

(2) else if type$(v) = \vee$, val$(v, u) =$ true iff val$(v_i, u) =$ true for all i.

(3) else if type$(u) =$ gen, val$(v, u) =$ true iff val$(v_i, u) =$ true for some i.

(4) else if type$(u) = \wedge$, val$(v, u) =$ true iff val$(v, u_j) =$ true for all j.

(5) [otherwise type$(v) = \wedge$ and type$(u) = \vee$] val$(v, u) =$ true iff for some i val$(v_i, u) =$ true or for some j val$(v, u_j) =$ true.

The condition of case (5) is known as Whitman's condition and is denoted (W).

For the purposes of analyzing the algorithm, we will represent lattice terms as follows. If w is a variable we represent it with itself. Otherwise we represent w as a list whose first element is type(w) and whose other elements are the representations of the joinands (or meetands) of w, i.e., Polish notation. Thus $x \vee y \vee (z \wedge t)$ is represented as $(\vee\ x\ y\ (\wedge\ z\ t))$.

2. The Complexity of Whitman's Algorithm

In this section we discuss the efficiency of Whitman's algorithm, i.e., the time it takes as function of the input length. This problem has been studied by computer scientists. Hunt, Rosenkrantz, and Bloniarz, using dynamic programming techniques, have shown that there is a polynomial time algorithm for deciding if $v \leqslant u$ in the free lattice, where v and u are lattice terms, [13]. They have also proved the remarkable result that the problem of deciding if $v \leqslant u$ in the free lattice in a variety generated by a finite lattice is co-NP complete. We will show that, at least in the worst cases, the straight-forward implementation of Whitman's algorithm (for example, as in the last section) is exponential. On the other hand, Whitman's algorithm is often faster than the polynomial algorithm. We present an algorithm which is polynomial but retains the advantages of Whitman.

As in the last section, we represent lattice terms as either a variable or a list whose first element is either $\vee$ or $\wedge$ and the remaining elements are representations of lattice terms. The *input length* is total number of $\vee$'s, $\wedge$'s, and variables (counting repeats) at all levels. To see that Whitman's algorithm is exponential, let x_i and y_i be variables and define lattice terms v_n and u_n inductively as follows. Let $v_1 = x_1$ and $u_1 = y_1$. Inductively define

$$v_{n+1} = x_{n+1} \wedge x_n \wedge (y_n \vee v_n), \qquad u_{n+1} = y_{n+1} \vee y_n \vee (x_n \wedge u_n).$$

(The representation of v_{n+1} is $(\wedge\, x_{n+1}\, x_n\, (\vee\, y_n\, \bar{v}_n))$, where $\bar{v}_n$ is the representation of v_n.)

Notice that the input length of v_{n+1} is a constant amount more than that of v_n, and similarly for u_{n+1} and u_n. In testing if $v_n \leqslant u_n$, we first break down v_n to see if any of its meetants is less than or equal to u_n. Continuing, we eventually test if $v_{n-1} \leqslant u_{n-1}$. Later we see if v_n is less than or equal to any joinand of u_n. This again leads to testing if $v_{n-1} \leqslant u_{n-1}$. Thus, to test $v_n \leqslant u_n$ we must test $v_{n-1} \leqslant u_{n-1}$ twice. This proves the following theorem.

THEOREM 2.1. *Whitman's algorithm is exponential.* □

In order to study Whitman's algorithm, we need to define the term tree, $T(t)$, associated with a term t. The term tree for a variable x has one node labelled x. The tree for a term $t = t_1 \vee t_2 \vee \cdots \vee t_n$ has the root, labelled with $\vee$, and n nodes connected to the root with the trees for the t_i's attached to them. We view trees as partially ordered sets with the root as the greatest element. The tree for $x \wedge y \wedge (r \vee s)$ is shown in Figure 1.

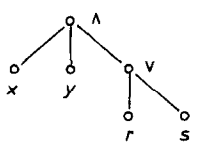


Fig. 1.

Notice that subterms correspond to principal ideals (all the nodes below or equal to a fixed node) in the term tree. Also note that the input length of a term t is just the number of nodes of the term tree $T(t)$. In dealing with the question $v \leqslant u$ we need the direct product of the term trees $T(v) \times T(u)$. Direct products of two trees are characterized by having a greatest element, which we still call the root, and the filter above each element is the direct product of two chains. We let a be a typical subterm of v and b a typical subterm of u. Then (a, b) is a node of $T(v) \times T(u)$. Recall the *value* of the node (a, b), denoted $\mathrm{val}(a, b)$, is true if $a \leqslant b$, and false otherwise. If $a \neq v$, then a has a unique upper cover in $T(v)$, which we denote a^*. Thus, a^* is the unique subterm of v which has a as an immediate subterm. If $b \neq u$ as well, then (a, b) has precisely two upper covers in $T(v) \times T(u)$, (a^*, b) and (a, b^*) (Figure 2).

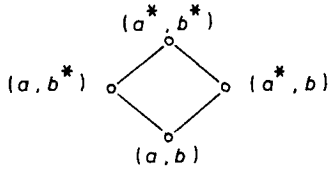


Fig. 2.

Notice that Whitman's algorithm implies the value of any node (a, b) is determined by the values of the nodes immediately below. The basic idea of the algorithm of Hunt, Rosenkrantz, and Bloniarz in [3] is this. Make a list of all subterms of v and a list of all subterms of u and order the lists by size. Form a matrix whose rows are indexed by the first list and whose columns are indexed by the second. The (a, b)th entry of the matrix is $\mathrm{val}(a, b)$. If the matrix is filled in the right order, the value of any entry can be determined from previously filled in entries. Of course the last entry of the matrix is the (v, u) entry, which is the answer. Since the matrix has order less than or equal to the input size of the problem (which is the input length of v plus the input length of u) it is not difficult to see that this algorithm is polynomial.

In many respects the above algorithm is very efficient. However, there are also a great many cases where Whitman's algorithm is much faster. If, for example, $a = a_1 \vee \cdots \vee a_k$ and $a_m \not\leqslant b$, then in determining $\mathrm{val}(a, b)$ Whitman's algorithm does not need the value of any node of $T(v) \times T(u)$ below (a_i, b) for $i > m$. Thus, Whitman's algorithm is able to prune away a great deal of the tree. An extreme case of this is the problem $x \wedge v \leqslant x \vee u$, where x is a generator and v and u are long terms. Whitman's algorithm will solve this instantaneously but the algorithm above will have to compute the whole matrix, taking time at least the square of the problem input length. Of course the disadvantage with Whitman's algorithm is that it can reach some of the nodes several (possibly exponentially many) times, whereas the algorithm above avoids this. We are seeking an algorithm with the advantages of both.

We will consider *Whitman-like* algorithms, i.e., algorithms that proceed through $T(v) \times T(u)$ as in Whitman's algorithm except that at certain nodes (a, b) it might record and/or lookup $\mathrm{val}(a, b)$ in a table. To study these algorithms we need a few more definitions. Recall that the *type* of a term a, $\mathrm{type}(a)$, is either $\vee$ or $\wedge$ or gen, depending on whether a is a join, meet, or generator. Let $\mathrm{type}(a, b) = (\mathrm{type}(a), \mathrm{type}(b))$. Recall that $\mathrm{val}(a, b)$ is determined by the values $\mathrm{val}(a_l, b)$ and $\mathrm{val}(a, b_j)$. We say that the node (a, b) *accesses* the (a_l, b) (and (a_l, b) is *accessed by* (a, b)) if the value of this node is used to calculate $\mathrm{val}(a, b)$. Note that (a, b) can be accessed by at most two nodes, namely (a^*, b) and (a, b^*), see Figure 2.

LEMMA 2.2. *In a Whitman-like algorithm a node* (a, b) *of type* $(\vee, \wedge)$ *with* $b \neq u$ *is never accessed. If* $b = u$ *such a node is accessed only if the type of* (v, u) *is* $(\vee, \wedge)$.

Proof. Suppose that $b \neq u$ and let (a, b) be a counter-example which is maximal in the ordering of $T(v) \times T(u)$. Obviously, $\mathrm{type}(a, b^*)$ is either $(\vee, \wedge)$ or $(\vee, \vee)$. In both cases Whitman's algorithm determines $\mathrm{val}(a, b^*)$ entirely from the values $v(a_l, b^*)$. Thus, (a, b) is not accessed by (a, b^*). Suppose that (a, b) is accessed by (a^*, b) and that $\mathrm{type}(a^*, b) = (\wedge, \wedge)$. In this case, $\mathrm{val}(a^*, b)$ is determined without $\mathrm{val}(a, b)$ as above. If $\mathrm{type}(a^*, b) = (\vee, \wedge)$ then we violate the maximality of (a, b). The proof of the second statement is similar. □

LEMMA 2.3. *An element* (a, b) *of type* $(\wedge, \wedge)$ *is not accessed by* (a^*, b) *unless* $b = u$ *and* $\mathrm{type}(v) = \vee$.

Proof. Again let (a, b) be a counter-example maximal in the ordering of $T(v) \times T(u)$. If the $\mathrm{type}(a^*, b) = (\wedge, \wedge)$, then $\mathrm{val}(a^*, b)$ is determined by the values $\mathrm{val}(a^*, b_l)$, and so (a^*, b) does not access (a, b). Hence, $\mathrm{type}(a^*, b) = (\vee, \wedge)$. By Lemma 2.2, $b = u$ and $\mathrm{type}(v) = \vee$. □

LEMMA 2.4. *An element* (a, b) *of type* $(\vee, \vee)$ *is not accessed by* (a, b^*).

Proof. Left to the reader. □

We will now conside a Whitman-like algorithm which looks up and records at all nodes of type $(\wedge, \vee)$, $(\mathrm{gen} \vee)$, $(\wedge, \mathrm{gen})$. By this we mean an algorithm which maintains a lookup table of previously determined values. When it reaches a node of one of the types above, it first looks to see if it already has found the value. If it has it takes that and does not access the nodes below. If the value has not been determined, it is determined recursively from the nodes below a la Whitman. Then it records the value on the table.

LEMMA 2.5. *With the algorithm described above a node* (a, b) *is not accessed twice by* (a^*, b) *nor is it accessed twice by* (a, b^*).

Proof. Suppose that (a, b) is accessed twice by (a, b^*) and is the largest node accessed twice from one side. Then (a, b^*) must be accessed from both sides. By the last three lemmas applied to the (a, b^*) and the fact that b^* is

obviously not a generator, the type of (a, b^*) is either $(\wedge, \vee)$, (gen, $\vee$), or (gen, $\wedge$). The last case cannot occur. Otherwise (a, b^*) would be accessed by (a^*, b^*), with type$(b^*) = \wedge$. But such a node would not access (a, b^*) unless type$(a^*) = \vee$, so that type$(a^*, b^*) = (\vee, \wedge)$. By Lemma 2.2, $b^* = u$. But then (a, b^*) cannot be accessed by (a, b^{**}), since b^{**} does not exist. Hence, the type of (a, b^*) must be either $(\wedge, \vee)$ or (gen, $\vee$). But, since we record at these nodes, the second time we reached (a, b^*) we would find its value on the table and not access (a, b) the second time. We leave the case that (a, b) is accessed twice by (a^*, b) to the reader. □

The next lemma will let us improve our algorithm.

LEMMA 2.6. *With the algorithm described before Lemma* 2.5 *if a node* (a, b) *is accessed by both* (a^*, b) *and* (a, b^*), *then it is accessed by* (a, b^*) *first.*

Proof. Let (a, b) be a counter-example. Then there are two accessed paths from (v, u) to (a, b), one through (a, b^*) and the other through (a^*, b). Let (c, d) be the node lowest in the ordering of $T(v) \times T(u)$ which lies in both paths. The interval of $T(v) \times T(u)$ between (a, b) and (c, d) is the direct product of two chains. The type of (c, d) must be $(\wedge, \vee)$ since by the minimality of (c, d) the two paths from (c, d) to (a, b) cannot start out the same. Let (c_i, d) and (c, d_j) be the two nodes below (c, d) and above (a, b). Whitman's algorithm will access (c_i, d) before (c, d_j). We claim the path through (c_i, d) is the one through (a, b^*), and the one through (c, d_j) is the one through (a^*, b). Otherwise the paths would cross since the interval is a product of two chains, violating the minimality of (c, d). □

The above lemma shows that it is possible to determine at each node whether to record or lookup or do neither and suggests the following algorithm: we have a table which initially has no values. We proceed as in Whitman's algorithm. When we are determining the value of (a, b) if we access (a_i, b), where type(a_i, b) is either (gen, $\vee$) or ($\wedge$, gen) or $(\wedge, \vee)$, we lookup on our table before continuing in the Whitman descent. If we access (a, b_j), and it is one of the above types, we determine the answer and then record it on the table.

If the input length of the problem (v, u) is n then there are at most n^2 nodes in $T(v) \times T(u)$. By Lemma 2.5, each node is accessed at most twice. How much time is spent at each node (a, b)? The original Whitman's algorithm only needed to determine the type of (a, b) and call the appropriate procedure. This can be done in constant time when we represent lattice words as lists with the operation symbol as the first element (i.e., Polish notation). In the algorithm above we must also lookup and/or record on a table. There are ways of doing this in nearly constant time.

A sketch of such a LISP program runs as follows. First make copy of v and a copy of u (time proportional to n). Maintain two counters. If we reach a node (a, b) where we wish to record and the first element of the list representing a is

a meet symbol and for b it is a join symbol (so that the type of (a, b) is $(\wedge, \vee)$), we replace the meet symbol of a with the current value m of our first counter (and then increment the counter) and replace the join symbol of b with the current value k of the second counter (using, e.g., RPLACA). Then we can create a symbol (i.e., LISP atom) mGk and set its value to val(a, b). (Alternately, we could maintain a rectangular array and set the (m, k)th entry to val(a, b).) It is possible that the first element of a and/or the first element of b is already a number. Then we use that number. We are still able to determine the type since in v only meet symbols are replaced by numbers and in u only join symbols are replaced. To record val(a, b) when a is an atom, say x, assign the symbol xGk to val(a, b), where k is the number we placed into the first element of b. Assuming that the value of mGk can be recovered in constant time, this algorithm is proportional to n^2. Since it first copies v and u, its minimum time is proportional to n.

It is possible to modify the algorithm so that it copies as it goes, only copying that part of v and u which is actually reached. (With LISP it is less natural to copy from the top down, but not difficult.) With this algorithm the minimum time is constant. Thus it retains the advantage of Whitman's algorithm of being particularly fast on many problems while keeping the maximum time to essentially n^2. A listing of the LISP program for this algorithm is available from the author.

As mentioned earlier, Whitman's algorithm is very good at paring down the direct product tree $T(v) \times T(u)$. Somewhat surprisingly, there is a simple modification of Whitman's algorithm which not only greatly improves this ability but also simplifies the proof of certain free lattice theorems.

LEMMA 2.7. *Let*

$$v = x_1 \wedge \cdots \wedge x_k \wedge v_1 \wedge \cdots \wedge v_r$$

and

$$u = y_1 \vee \cdots \vee y_s \vee u_1 \vee \cdots \vee u_t$$

be elements of a free lattice, where the x_i*'s and the* y_j*'s are generators. Then* $v \leqslant u$ *if and only if*

$$\exists i, j\ x_i = y_j \quad \text{or} \quad \exists i\ v_i \leqslant u \quad \text{or} \quad \exists j\ v \leqslant u_j. \tag{W+}$$

Proof. This follows easily from (W). □

3. Algorithms on Covers

We say that b *covers* a in a lattice L if $a < b$ and there is no $c \in L$ with $a < c < b$. We write $a \prec b$ and say that b is an *upper cover* of a and a is a *lower cover* of b. Whitman noticed a few particular covers in free lattices. For example, he noted

that in FL(X), $\vee(X-\{x\})\prec\vee X$, for any $x\in X$ when X is finite.

The study of covers was revived with McKenzie's work on lattice varieties. He noted that if $a\prec b$ in FL(X) and $\psi(a,b)$ is the unique largest congruence separating a from b, then $L=\mathrm{FL}(X)/\psi$ is a splitting lattice, i.e., there is a unique largest variety not containing L, namely the variety defined by the law $a=b$. Splitting lattices and the associated equations play an important role in the study of lattice varieties (see, for example, [16]), lattice structure theory (e.g., [4]), congruence varieties (e.g., [2], [3]), and modular lattices (e.g., [5]). Moreover covers in free lattices are fascinating in their own right. There were three important results. The first is an unpublished result of Dean's.

THEOREM 3.1. *There are elements of FL(3) which have no lower covers. In fact, $x\wedge(y\vee z)$ is such an element.*

The second is McKenzie's result.

THEOREM 3.2. *One can recursively decide for lattice terms u and v if $v\prec u$ in the free lattice.*

The third result is Alan Day's.

THEOREM 3.3. *FL(X) is weakly atomic when X is finite. That is, every interval contains a covering.*

McKenzie's algorithm started out with X as the variables in v and u, and then formed the subset S of FL(X) obtained by starting with X and alternately closing under joins and meets until v and u were in the subset. S is closed under one of the lattice operations, contains the least and greatest elements, and is finite. Thus, it is a lattice (although not a sublattice). McKenzie's algorithm examined the homomorphic images of S and showed that $v\prec u$ if and only if one of these images had a certain property (a bounded homomorphic image of a free lattice).

When X has three elements and we form S by closing X under joins then meets then joins then meets, S has 677 elements. Closing once more probably gives more than 10 000 elements. So the algorithm is impractical.

J. B. Nation and I became interested in the following problem: *can one recursively decide for a lattice term u if its interpretation into the free lattice has a lower cover*? It turns out that it is recursive (see [10]). We have developed a fast algorithm for deciding if u has any lower covers and finding them if it does.

Our original algorithm used McKenzie's algorithm, and thus was not efficient. We were unable to use this algorithm to decide other questions about free lattices such as is there an element with no lower and no upper cover and what are the finite interval sublattices of free lattices.

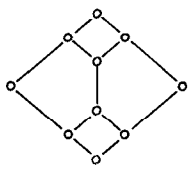
We present a sketch of the algorithms and a simple formula for w_*, when it exists. First, if u is a join in FL(X) then u has a lower cover if and only if one of its joinands does. Thus we may assume that u is join-irreducible. In this case, u has a lower cover if and only if it is completely join-irreducible. If u does have

a lower cover we denote it u_*. In this situation we let $\varkappa(u)$ be the canonical meetand of u_* not above u. It is not hard to see that $\varkappa(u)$ is completely meet-irreducible with upper cover $\varkappa(u)^* = u \vee \varkappa(u)$, and $\varkappa(u)$ is the largest element above u_* but not above u. Note $\varkappa$ defines a bijection between the completely join-irreducible elements and the completely meet-irreducible elements. It is possible to show that u and $\varkappa(u)$ have essentially the same complexity. Thus, there are only finitely many candidates for $\varkappa(u)$, so we can use McKenzie's algorithm to find out if u is completely join-irreducible. This, of course, is not efficient.

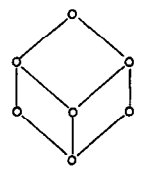
A more efficient procedure is this. For w join-irreducible in FL(X) define a subset $J(w)$ of FL(X) as follows: if w is a meet of generators then $J(w) = \{w\}$. If the canonical form of w is

$$w = \bigvee_i \left(\bigwedge_j w_{ij} \right) \wedge \bigwedge_k x_k$$

where $x_k \in X$, then $J(w) = \{w\} \cup \bigcup_{i,j} J(w_{ij})$. $L(w)$ is the join closure in FL(X) of $J(w)$ with a least element adjoined. This is a finite lattice and w has a lower cover if and only if it satisfies the law: for all a, b, c if $a \wedge b = a \wedge c$ then $a \wedge b = a \wedge (b \vee c)$. (A lattice satisfying this law and its dual is called *semi-distributive*.) Since $L(w)$ relatively small, this is not hard to check. Figure 3 below gives two examples.



$L[(x \vee (y \wedge z)) \wedge (y \vee (x \wedge z))]$



$L[x \wedge (y \vee z)]$

Fig. 3.

It is easy to see that the first lattice is semidistributive, and that the second fails the above law. Thus $(x \vee (y \wedge z)) \wedge (y \vee (x \wedge z))$ has a lower cover but $x \wedge (y \vee z)$ does not. (The latter is Dean's result cited above.)

The efficient syntactic algorithm which we have implemented on the microcomputer is this.

THEOREM 3.4. *Form $J(w)$ as above. Then w is completely join irreducible if and only if each $u \in J(w) - \{w\}$ is and*

$$w \not\leqslant \bigvee K(w)$$

where

$$K(w) = \{v \in J(w) : v \vee w_\dagger \not\geqslant w\}$$

and

$$w_\dagger = \bigvee \{v \in J(w) : v < w\}.$$

Not only is this algorithm fast, it has some nice corollaries. The important one is this: a necessary (but not sufficient) condition for a join irreducible element to be completely join irreducible is that for each canonical meetand $w_i = \bigvee_j w_{ij}$ of w there is exactly one j with w_{ij} not below w. Note that w, being completely join irreducible, is independent of the generating set X and that if any u in $J(w)$ fails to be completely join irreducible, then w is not completely join irreducible.

Although it is much more difficult to find the lower cover w_* of a completely join irreducible element w, the corollary above is the key to an efficient method. Since $w_* = w \wedge \varkappa(w)$, it suffices to calculate $\varkappa(w)$. Let

$$k^\dagger = \bigwedge\{\varkappa(v) : v \in J(w) - \{w\}, \varkappa(v) \geqslant \bigvee K(w)\}.$$

Then we have the following simple formula for $\varkappa(w)$.

THEOREM 3.5. *If w is completely join irreducible in $FL(X)$ then $\varkappa(w)$ is given by the following*

$$\varkappa(w) = \bigvee\{x \in X : x \vee w_\dagger \not\geqslant w\} \vee \bigvee\{k^\dagger \wedge \varkappa(v) : v \in J(w) - \{w\}, w \not\leqslant \varkappa(v)\}.$$

Although this algorithm asks you to calculate $\varkappa(v)$ for all v in $J(w) - \{w\}$, there are at most the input size of w elements in $J(w)$. From this it is easy to prove the following theorem.

THEOREM 3.6. *The problem of deciding if a lattice word has a lower cover and of finding such a lower cover if it exists can be done in polynomial time.*

Proof. First it is necessary to put w into canonical form, and thus we must show that this can be done in polynomial time. This has been done by Hunt, Rosenkrantz, and Bloniarz [13]. The rest of the proof to the reader. □

4. A Sketch of the Program with an Introduction to LISP

There has been an increased interest among mathematicians in the computer language LISP, see, for example, [12, 17]. Because of its mathematical structure recursive nature, it is well suited to symbolic math programming. We will give a brief outline of the fundamentals of this language. Its data structures are atoms (numbers and character strings) and binary trees. List are coded into the binary trees and for simplicity we will stick to lists. A list is a finite sequence whose members are either atoms or lists. The empty list is denoted by NIL and this is also used to denote false. The basic LISP functions are ATOM, EQ, CAR, CDR, and CONS. ATOM returns T (for true) if its argument is an atom, and NIL otherwise. EQ returns T if both of its arguments are the same atom, and NIL otherwise. The CAR of a list is its first element and the CDR is the list obtained by removing the first element. If the second argument of CONS is the list

L then it returns L with the first argument of CONS adjoined as a new first element. The syntax for $f(x)$ is $(f\,x)$. $f(g(x))$ has the form $(f\,(g\,x))$. For example, if L is a nonempty list then (CONS (CAR L) (CDR L)) returns L. The function COND takes an arbitrary number of arguments, each an ordered pair. It evaluates the first part of each pair until it finds one whose value is anything except NIL and then returns the value of the second half. If all evaluate to NIL then NIL is returned.

Lattice terms are represented as lists. The generators are just atoms, e.g., x, y, z. If $w_1, w_2, \ldots, w_n$ are LISP representations of lattice terms then the join is represented by the list $(+\, w_1\, w_2 \cdots w_n)$ and the meet by $(*\, w_1\, w_2 \cdots w_n)$. Our basic function is LSSQL. This has two arguments, lattice terms v and u. It gives T if $v \leqslant u$ in the free lattice and NIL otherwise. A definition, using the basic form of Whitman's algorithm, is given below. Note how closely it resembles Whitman's algorithm.

```
(DEFUN LSSQL (LAMBDA (V U)
  (COND
    ((ATOM V)
      (COND
        ((ATOM U) (EQ V U) ) )
        ((EQ (CAR U) *) (LSSQL-V-ALL V (CDR U)))
        (T (LSSQL-V-EX V (CDR U))) ) )
    ((EQ (CAR V) +) (LSSQL-ALL-U (CDR V) U))
    ((ATOM U) (LSSQL-EX-U (CDR V) U))
    ((EQ (CAR U) *) (LSSQL-V-ALL V (CDR U)))
    (T (OR (LSSQL-EX-U (CDR V) U)
           (LSSQL-V-EX V (CDR U)) ) ) ) ))
```

The first line is used to define LISP functions. The function body reads like this: 'if V is an atom (generator) then if U is an atom the value of LSSQL is the value of (EQ V U), otherwise if U is a meet then the value is the value of (LSSQL-V-ALL V (CDR U))', etc. The T's are used for the default cases. Notice that Whitman's condition (W) is given by the last clause. The function LSSQL-V-ALL has two arguments, a lattice term v and a list of lattice terms. It returns T if v is less than or equal to each of the elements of the list. It can be defined as follows:

```
(DEFUN LSSQL-V-ALL (LAMBDA (V LST)
  (COND
    ((NULL LST) T)
    (T (AND (LSSQL V (CAR LST))
            (LSSQL-V-ALL V (CDR LST)) )) ) ))
```

The value of (NULL LST) is T if LST is the empty list. The meaning of the function AND, as well as the other helper functions to LSSQL should be clear.

The definition of LSSQL using the algorithm of Section 2 is somewhat more complicated, but still relatively easy.

Next we define two functions + and *. The first takes an arbitrary number of lattice terms as arguments and gives canonical form of their join provided each is in canonical form. The second is defined dually. One of the beauties of LISP is that the functions as well as the data are lists. The function EVAL evaluates a list as a function. So with this scheme an arbitrary term w can be put into canonical form simply by (EVAL w). With this setup it is easy to define functions for the algorithms of the previous section. For example, the function (CJI w) gives T if w is completely join irreducible and the function (KAPPA w X) gives $\kappa(w)$ in FL(X). The functions J and L give $J(w)$ and $L(w)$. The function (LOWER-COVERS x X) gives a list of all the lower covers of w in FL(X).

If L is a list of lattice terms, the function (DRAW L) draws the partially ordered set on L with the order inherited from the free lattice. The algorithm for drawing L is interesting. It arranges L into levels, the first level being the minimal elements, the second being the minimal elements which remain, etc. These are then positioned symmetrically on the screen. The hard part is deciding how to arrange the elements within the levels. For example, consider the 8 element Boolean algebra (Figure 4).

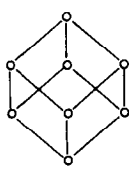


Fig. 4.

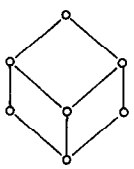


Fig. 5.

In the above scheme, suppose we have positioned the least element and the three atoms. There are six ways to arrange the coatoms, five of which are bad. Our algorithm works roughly as follows. Suppose the first $i-1$ levels have been positioned. We then order these elements essentially with the left to right order on the screen (for ties the one higher on the screen is lower in this order). Each element of the ith level has its list of lower covers, which, of course, have already been positioned. We order the elements within these lists of lower covers using the above order. Now the elements of the ith level are positioned according to the lexicographic order on their lists of lower covers. In the Boolean algebra above, if the atoms are labelled 1, 2, and 3, then the coatoms have lists of lower covers (1 2), (1 3), and (2 3). Note that this leads to the 'correct' picture above.

The algorithm also applies this procedure to the dual of L as well. In the lattice of Figure 5 if we position the atoms wrong (i.e., if the 'middle' atom is not placed in the middle), then no positioning of the coatoms will help. However, using the information about the dual solves this problem.

5. Remarks

When comparing the actual running time for deciding if $v \leqslant u$, Whitman's original algorithm does very well. Of course, if we use the terms v_n and u_n of Theorem 2.1, Whitman's algorithm is extremely slow. When n is 20 Whitman's algorithm takes about 16 hours while our polynomial algorithm only takes a few seconds. Nevertheless, on most problems Whitman's algorithm is about the same or a little faster than the other algorithm. This leads to the question: is the expected time of Whitman's algorithm polynomial? To answer such a question, we would first have to decide on the appropriate probability space for our lattice terms. The combinatorics of such a space would necessarily be difficult. We can make a few observations. Although Whitman's algorithm may access a particular node (a, b) several times, each path from (v, u) to (a, b) will be traveled at most once, as is easy to see. The number of paths from (v, u) down to (a, b) is $\binom{d(a)+d(b)}{d(a)}$, where $d(a)$ is the depth of a in $T(v)$. Thus, in order for the repeated accessing of (a, b) to significantly slow down the Whitman algorithm, it would be necessary that $d(a)$ and $d(b)$ be not too small compared to the size of $T(a)$ and $T(b)$. If, for example, $T(v)$ and $T(u)$ are balanced, then the depth of the trees is logarithmic in their size and this doubling effect would not be significant. Moreover, for terms which alternate between join and meet, the binomial coefficient above can be replaced with $\binom{(d(a)+d(b))/2}{d(a)/2}$. Thus, the words would have to have depth 8 before the repeated accesses could begin to have a noticeable effect.

Another reason that Whitman's algorithm appears fast, is that in our program, the words are usually in a form with the shorter joinands or meetands first and very often these determine if $v \leqslant u$.

Despite these observations, I do not think that Whitman's algorithm could have polynomial expected time. It is known that the expected depth of a random tree is a constant times $\sqrt{n}$, where n is the number of nodes. Assuming that we give lattice terms a probability measure such that expected depth is also on the order of $\sqrt{n}$, and consider words v and u with no variable in common, it is possible to show that the expected time is exponential.

Acknowledgements

This research was partially supported by the National Science Foundation Grant No. DMS 85-21710.

The author would like to thank Dale Myers, J. B. Nation, and Steve Tschantz for several stimulating talks about this subject.

References

1. P. Crawley and F. P. Dilworth (1973) *Algebraic Theory of Lattices*, Prentice-Hall, Englewood Cliffs, NJ.

2. A. Day (1977) Splitting lattices generate all lattices, *Algebra Univ* **7**, 163–169.
3. A. Day and R. Freese (1980) A characterization of identities implying modularity, I, *Canad. J. Math.* **32**, 1140–1167.
4. R. Freese (1973) Breadth two modular lattices, *Proc. Univ. Houston Lattice Theory Conf. (Houston, Texas, 1973)*, Math. Dept., Univ. Houston, 1973, pp. 409–456.
5. R. Freese (1979) Projective geometries as projective modular lattices, *Trans. Amer. Math. Soc.* **251**, 329–342.
6. R. Freese (1980) Free modular lattices, *Trans. Amer. Math. Soc.* **261**, 81–91.
7. R. Freese (1982) Some order theoretic questions about free lattices and free modular lattices, *Ordered Sets*, (I. Rival, ed.), D. Reidel, Dordrecht.
8. R. Freese (1985) Connected components of the covering relation in free lattices, in *Universal Algebra and Lattice Theory* (S. Comer, ed.), Lecture Notes in Mathematics, 1149, Springer-Verlag, New York, pp. 82–93.
9. R. Freese and J. B. Nation (1978) Projective lattices, *Pacific J. Math.* **75**, 93–106.
10. R. Freese and J. B. Nation (1985) Covers in free lattices, *Trans. Amer. Math. Soc.* **288**, 1–42.
11. G. Gratzer (1978) *General Lattice Theory*, Academic Press, New York.
12. Douglas Hofstadter (1983) The pleasures of LISP: the chosen language of artificial intelligence, *Sci. Amer.* Feb. 1983, pp. 14–28.
13. H. B. Hunt, III, D. J. Rosenkrantz, and P. A. Bloniarz, On the computational complexity of algebra on lattices 1, Preprint.
14. B. Jónsson (1982) Varieties of lattices: Some open problems, *Colloq. Math. Soc. Janos Bolyai*, **29**, Contributions to Universal Algebra (Esztergom), North Holland, pp. 421–436.
15. B. Jónsson and J. B. Nation (1977) A report on sublattices of a free lattice, *Colloq. Math. Soc. Janos Bolyai*, **17**, Contributions to Universal Algebra (Szeged), North Holland, pp. 223–257.
16. R. McKenzie (1972) Equational bases and nonmodular lattice varieties, *Trans. Amer. Math. Soc.* **174**, 1–43.
17. Mitchell Wand (1984) What is LISP, *Amer. Math. Monthly*, pp. 32–42.
18. P. Whitman (1941) Free lattices, *Ann. Math.* **42**, 325–330.
19. P. Whitman (1942) Free lattices II, *Ann. Math.* **43**, 104–115.