Syracuse University

SURFACE

Electrical Engineering and Computer Science -
Technical Reports

College of Engineering and Computer Science

10-1990

A Space and Time Efficient Coding Algorithm for Lattice Computations

Deb Dutta Ganguly

Sanjay Ranka
Syracuse University

Follow this and additional works at: https://surface.syr.edu/eecs_techreports

Part of the [Computer Sciences Commons](#)

Recommended Citation

Ganguly, Deb Dutta and Ranka, Sanjay, "A Space and Time Efficient Coding Algorithm for Lattice Computations" (1990). *Electrical Engineering and Computer Science - Technical Reports*. 82.
https://surface.syr.edu/eecs_techreports/82

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

A Space and Time Efficient Coding Algorithm for Lattice Computations

Deb Dutta Ganguly and Sanjay Ranka

October 1990

*School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100*

(315) 443-2368

A Space and Time Efficient Coding Algorithm for Lattice Computations

Deb Dutta Ganguly and Sanjay Ranka
4-116 CST, School of CIS
Syracuse University
Syracuse, NY 13210
ranka@top.cis.syr.edu
(315)-443-4457

October 30, 1990

Abstract

This paper presents an encoding algorithm to enable fast computation of the least upper bound (LUB) and greatest lower bound (GLB) of a partially ordered set. The algorithm presented reduces the LUB computation to an OR operation on the codes. The GLB computation is reduced essentially to an AND operation on the codes. The time complexity of our encoding algorithm is $\mathcal{O}(n + e)$ where n is the number of nodes and e is the number of edges. With respect to space requirements the algorithm presented gives good results for small lattices (code length was 50 bits for a 300 node lattice), but it gives truly remarkable results for larger lattices (*e.g.* for a 950 node lattice it used 110 bits).

Keywords and Phrases

class inheritance, GLB, LUB, lattice, poset.

List of Figures

- Figure 1 : A lattice
- Figure 2: Adjacency Matrix
- Figure 3: Transitive Closure Matrix
- Figure 4: Codes after Pass 1
- Figure 5: Codes after Pass 2
- Figure 6: Codes after Pass 1
- Figure 7: Codes after Pass 2
- Figure 8: Algorithm Encode
- Figure 9: Pass 1
- Figure 10: procedure *name_children*
- Figure 11: procedure *max_pure_siblings*
- Figure 12: procedure *compute_prefix_len*
- Figure 13: Pass 2
- Figure 14: procedure *glb_info*
- Figure 15: Number of nodes vs. Avg. code length
- Figure 16: Number of nodes vs. Avg. computation time

1 Introduction

Lattice operations are used to determine object properties by conjunction, disjunction, or exclusion of certain class properties. In [1] Hassan Ait-Kaci et al. discuss the applications of lattice computations in languages that support (multiple) inheritance in partially ordered classes. [2] presents a overview of the research in partially ordered data types : semantic networks, the first order approach, the initial algebra approach and the denotational approach.

Kifer and Subrahmanian [3] have developed a theoretical foundation for multivalued logic programming. They present a procedure for processing queries to such programs and show that if certain constraints (over lattices) associated with such queries are solvable, then their proof procedure is effectively implementable. Thus, an engine for solving such constraints over lattices is critical to the practical implementation of generalized annotated logic programming of [3]. An important contribution of the Kifer-Subrahmanian work is that they show that their generalized annotated logic programming formalism is applicable to various important issues relating to expert systems. In particular, uncertainty of various different kinds (e.g. bilattices, Bayesian uncertainty propagation) can be handled in their framework. Additionally, their framework can be used to reason about databases that contain inconsistencies. As inconsistencies can easily arise in knowledge based systems (due either to errors in the data, or due to genuine differences of opinions amongst multiple experts), it is vital that databases behave well in the presence of such inconsistencies. Furthermore, Kifer and Subrahmanian [3] demonstrate that their framework can also be used for temporal reasoning.

However, the query processing procedure developed by them lacks an important component, viz. their procedure is completely contingent upon certain constraints being solvable. However, no such procedures are developed by Kifer and Subrahmanian [3]. We address this problem in this paper. The solution to this problem presented here would make the Kifer Subrahmanian procedure for processing queries implementable.

We first define a few basic notions [4].

Definition 1 : A binary relation $\leq$ on a set P is called a *partial ordering* in P iff $\leq$ is reflexive, antisymmetric and transitive. The ordered pair $\langle P, \leq \rangle$ is called a *partially ordered set* or a *poset*.

Definition 2: Let $\langle P, \leq \rangle$ be a partially ordered set and let $A \subseteq P$. Any

element $x \in P$ is an *upper bound* for A , if for all $a \in A$, $a \leq x$. Similarly any element $x \in P$ is a *lower bound* on A if for all $a \in A$, $x \leq a$.

Definition 3: Let $\langle P, \leq \rangle$ be a partially ordered set and let $A \subseteq P$. An element $x \in P$ is a *least upper bound* for A if x is an upper bound for A and $x \leq y$ for every upper bound y for A . Similarly, the *greatest lower bound* for A is an element $x \in P$ such that x is a lower bound and $y \leq x$ for all lower bounds y .

A least upper bound if it exists is unique and the same is true for a greatest lower bound. The least upper bound is abbreviated as “LUB” and the greatest lower bound is abbreviated as “GLB”.

Definition 4: A *lattice* is a partially ordered set $\langle L, \leq \rangle$ in which every pair of elements $a, b \in L$ has a greatest lower bound and a least upper bound.

Lattices can be encoded by a brute-force approach using transitive closure (Section 2), such that the AND operation on two codes gives the LUB. This method uses n bits to encode each node of a lattice with n nodes. The total amount of space required is thus $\mathcal{O}(n^2)$. This may be prohibitive for large lattices. [1] presents an algorithm which uses “modulation” to reduce the code-length. Our algorithm is simpler and has $\mathcal{O}(n + e)$ time complexity. The LUB operations can still be completed by OR operations. The GLB computation is reduced to an AND operation on the codes followed by a simple step. The algorithm gives good results for small lattices (average code length was 50 for a 300 node lattice), but it gives truly remarkable results for larger lattices (e.g. for a 950 node lattice it used 110 bits for encoding).

Section 3 describes a simple version of the algorithm when applied on a tree. Section 4 discusses the changes necessary to apply the same basic paradigm for poset encoding. Section 5 describes the algorithm and section 6 proves its correctness. Section 7 discusses the implementation. Section 8 concludes the paper.

Figure 1 : *A lattice*

2 Transitive Closure

In this section we discuss the transitive closure technique for encoding lattices. Consider the lattice in Figure 1. Its adjacency matrix A is shown in Figure 2. The edges are directed downwards. The adjacency matrix has a '1' in the row headed by x under the column headed by y iff there is an edge from x to y in the lattice. Otherwise a position in the adjacency matrix has a '0'. A row headed by x is a representation of the set of all the immediate lower bounds of x . Similarly a column headed by y can be viewed as a representation of the set of all the immediate upper bounds of y . Since we are interested in LUB here, we will take the latter view.

Next the transitive closure A^* of A is calculated by matrix multiplication. This is given by:

$$A^* = \bigcup_{i=0}^n A^i$$

	a	b	c	d	e	f	g	h
a	0	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0	0
c	1	0	0	0	0	0	0	0
d	1	0	0	0	0	0	0	0
e	1	0	0	0	0	0	0	0
f	0	1	1	0	0	0	0	0
g	0	0	1	1	1	0	0	0
h	0	0	0	0	0	1	1	0

Figure 2: Adjacency Matrix of the lattice in Figure 1

	a	b	c	d	e	f	g	h
a	1	0	0	0	0	0	0	0
b	1	1	0	0	0	0	0	0
c	1	0	1	0	0	0	0	0
d	1	0	0	1	0	0	0	0
e	1	0	0	0	1	0	0	0
f	1	1	1	0	0	1	0	0
g	1	0	1	1	1	0	1	0
h	1	1	1	1	1	1	1	1

Figure 3: Transitive Closure Matrix of the lattice in Figure 1

This computation converges in $\mathcal{O}(\log_2 n)$ matrix multiplications of $n \times n$ boolean matrices. First $A^1 = I \cup A$ is calculated, from this A^2 and so on till two consecutive matrices are the same. A^* is shown in Figure 3. Clearly the 1's in the column headed by y indicate the upper bounds of y . Now an AND operation on two columns will yield the set of the common upper bounds. For example AND of the columns under 'b' and 'c' gives $[00000101]^T$ which is the code under the column headed by node 'f' the common upper bound and the LUB. Note that in a lattice it is possible for two nodes to have more than one LUB. In that case the AND of the codes will yield a code which represents the set of all common upper bounds.

This method uses n bits to encode each node of a lattice with n nodes. The total amount of space required is thus $\mathcal{O}(n^2)$ bits. This may be prohibitively high for large lattices.

3 Tree Encoding

In this section we will discuss a coding algorithm for a tree which forms the basis of our lattice encoding algorithm. The algorithm works in two passes. In Pass 1, the lattice is swept layer¹ by layer from layer 0, the bottommost

¹the word layer has been used for trees here (as opposed to the usual 'level') and lattices in subsequent sections to maintain consistency.

layer of minimal elements, to the topmost layer of maximal elements. At each layer the nodes are encoded such that the sibling nodes get distinct codes. In Pass 2, the lattice is swept from the top to down. Now the non-sibling nodes are distinguished by prefixing them with their respective parent's codes. The bitwise-OR of any two codes yields the code of the LUB.

In Pass 1 if there are n nodes with a common parent then an n bit code of the form $2^i, 0 \leq i \leq n - 1$ is used *i.e.*, '1' at the i^{th} position and '0's at all other positions. Thus, at layer 0 the children of the same parent are assigned distinct codes. Then at every subsequent layer for each node such a code is prefixed to the bit-wise OR of the children's codes. For instance, the bitwise OR of node a 's code (01) and node b 's code (10) is 11, to which 01 is prefixed yielding node i 's code (0111) in the left most subtree in Figure 4. The prefixing ensures that we can distinguish the codes of siblings. In Pass 1 the nodes with the same parent are assigned distinct codes but the nodes with different parents may still have identical codes. Pass 2 makes them distinct. Figure 4 illustrates the result of applying Pass 1 to the binary tree shown.

Pass 2 starts at layer $n-2$, where layer n is the topmost (maximal) layer. It prefixes the existing codes to yield the final codes. Let $d = length(parent.code) - length(current.code)$. Each code is prefixed by the leftmost d bits of its parent. Thus yielding the codes in the Figure 5.

Example: Consider the nodes a and d in Figure 5. The bitwise OR of their codes is 011111, which is the code of m , the LUB of nodes a and d .

Our encoding algorithm ensures that the LUB of two nodes has a '1' at all the positions at which the bitwise OR of the codes of the two nodes has a 1 and may be a few more. Now consider the nodes a and b in Figure 5. The bitwise OR of their codes is 011101. Nodes i, m, o *subsume* this code, of these node i is at the lowest layer hence it is the LUB.

3.1 Analysis of Tree Encoding

For a tree with a constant branching factor b the above algorithm will use b bits at each layer. If the tree has l layers then $b \times l$ bits would be required for encoding each node of the tree. Thus when $b = 2$ the entire tree uses $2 \times l$ bits. If the tree has n nodes then in terms of the n only $2 \times \log_2 n$ bits are used. The entire tree would use $2n \times \log_2 n$ bits compared to n^2 bits used by the Transitive Closure method. The algorithm works well for a

Figure 4 : Codes after Pass 1

Figure 5 : Codes after Pass 2

tree, but cannot be directly used on a lattice. The main difference between a lattice and a tree, in the context of this algorithm, is that every node in a tree has an indegree² of one (*pure* nodes) whereas in a lattice some nodes can have an indegree greater than one (*impure* nodes). This necessitates a few modifications to the above algorithm. We discuss the basic modification in the next section.

4 Lattice Encoding

In this section we discuss the basic modification required for applying the above paradigm to a lattice. We observed that the problem arises because of the *impure* nodes. To overcome this problem we assign distinct prefixes to the *impure* nodes in Pass 1 which are not used again.

Pass 1 starts at layer 1 and first the prefix length of the current layer is calculated. This is the sum of the number of *impure* nodes and the maximum number of *pure* sibling nodes³(*e.g.* in Figure 6 at layer 1 there is one node *c* which is *impure* and nodes *d* and *e* form the largest set of *pure* sibling nodes. Hence the prefix length at this layer is 3). Next the *impure* nodes are assigned distinct prefixes of the form 2^i , (*i.e.*, a ‘1’ in the i^{th} bit position) which are not used again (*e.g.* in Figure 6 node *c* gets the prefix 100). This gives a unique identity to the node (lemma 1) since only ancestors and descendants of this node can now have a 1 at the i^{th} position. While coding the *pure* nodes we follow the same strategy as for a tree (*e.g.* in Figure 6 at layer 1 node *c* is assigned the code 100 which is not used again). After this at the next layer each node gets the bitwise OR of its children codes. Then each code is prefixed similarly. The process continues for every subsequent layer until we reach the topmost layer. Consider the lattice in Figure 1. Figure 6 illustrates the result of applying Pass 1 of the modified algorithm to the lattice in Figure 1.

In Pass 2 we prefix the codes of a *pure* nodes with the leftmost d bits of its parent, where $d = \text{length}(\text{parent.code}) - \text{length}(\text{child.code})$. The code of an *impure* node is prefixed by d 0’s so that its code length remains same as that of the other nodes. Thus we get the codes in Figure 7. The part of the

²the number of parents of a node

³We note that in most cases in practice the number of nodes with indegree greater than 1 is few.

Figure 6 : *Codes after Pass 1*

Figure 7 : Codes after Pass 2

code in boldface was added during Pass 2.

Our encoding algorithm ensures that the LUB of two nodes has a ‘1’ at all the positions at which the bitwise OR of the codes of the two nodes has a 1 and may be a few more. We say that the LUB *subsumes* the bitwise OR of the codes.

Definition 5: *b.code subsumes a.code* iff

$$\forall i((2^i \& a.code) = 2^i \implies (2^i \& b.code) = 2^i)$$

; where $\&$ denotes the bitwise AND operation

Example : Consider the nodes *c* and *e* in Figure 7, the bitwise OR of their codes is 01101, but there is no node with such a code. So we look for a node whose code *subsumes* this code. Nodes *g* and *h* both subsume 01101, since they are both upper bounds of nodes *c* and *e*. Of these node *g* is the

```

Algorithm Encode( $L : \text{lattice}$ )
    Form_Layers( $L : \text{lattice}$ );
    Pass1 ( $L : \text{lattice}$ );
    Pass2 ( $L : \text{lattice}$ );

```

Figure 8: *Algorithm Encode*

LUB since it is at a lower layer.

After encoding the codes may be stored in an array sorted lexically by the codes. A linear search on this array for the LUB code will take $\mathcal{O}(n)$ time. Alternatively the codes may be stored in a data structure which is the same as the original lattice. Suppose *lub.code* is the bitwise-OR of the codes whose LUB we are computing. We start at the maximal node and move to the child which *subsumes* *lub.code*. We keep moving similarly until we get a node whose children do not *subsume* *lub.code*. This node is the LUB. This operation would take $\mathcal{O}(h)$ time where h is the height of the lattice.

These modifications and a few more yield the algorithm in the following section.

5 Algorithm Encode

This section the procedures invoked by Algorithm Encode (figure 8) in detail.

Form_Layers divides the lattice into layers. The layering is done using a depth first search starting at the minimal node and going upwards. A layer is a set of incomparable nodes - a cochain, computed as the set of all the immediate parents that cannot be reached later. *Algorithm Encode* next executes *Pass1* and *Pass2*. We now describe them in detail.

In a lattice it is possible that some, but not all, of the children of the node reside in layers below the one immediately below the node's layer. However the proof of correctness of the algorithm is simplified by the notion that there exists a *continuous* path (*i.e.*, a path that does not jump across layers) between each ancestor-descendant pair. If an edge jumps across layers we

```

int : curr_code_len, unik[max_noof_layers]
/* The  $i^{th}$  element of the array unik[man_noof_layers] stores the bit-position
from which the pure nodes in the  $i^{th}$  layer are assigned codes */

procedure Pass1( $L$  : lattice)
1. int layer_no,  $i, d, prefix, virtual\_code$ 
   node  $n, m$ 
   global int : curr_code_len, unik[max_noof_layers]
2. curr_code_len  $\leftarrow 0$ 
3. For layer_no  $\leftarrow 1$  to max_layer do
4.     For each node,  $n$ , in the layer numbered layer_no do /* get the bitwise OR
        of all the children codes */
5.          $n.code \leftarrow zero(curr\_code\_len)$  /* initialize  $n.code$  to
curr_code_len long string of 0's */
6.         For each node  $c \in children(n)$  do
7.              $n.code \leftarrow n.code \text{ OR } c.code$ 
8.         endfor
9.         If outdegree( $n$ ) = 1 then
10.            If indegree(child( $n$ )) = 1 then
11.                 $i \leftarrow left\_most\_1(child(n))$  /* bit position of
the left most 1 in child's code */
12.                 $n.code \leftarrow 2^{i-1} \text{ OR } child(n).code$ 
13.            endif
14.        endif
15.    endfor
16.    If layer_no  $\neq$  max_layer do
17.        curr_code_len  $\leftarrow curr\_code\_len + compute\_prefix\_len(layer\_no)$ 
18.         $i \leftarrow curr\_code\_len$  /* the bit position from which coding will
start at this layer; first encode the impure nodes */
19.        For each node,  $n$ , in the layer numbered layer_no, do
20.            If indegree( $n$ ) > 1 then
21.                 $n.code \leftarrow 2^i \text{ OR } n.code$ 
22.                 $n.code\_len \leftarrow curr\_code\_len$ 
23.                 $i \leftarrow i - 1$ 
24.            endif
25.        endfor /* impure nodes encoding finished */
26.        unik[layer_no]  $\leftarrow i$  /* the bit position from which the
pure nodes in the layer numbered layer_no are assigned codes */
27.        For each node,  $m$ , in the layer numbered layer_no + 1 do
28.            name_children( $m, layer\_no$ )
29.        endfor
30.    endif
31.endfor

```

Figure 9: Pass1

will assume that *virtual* nodes are inserted in each intermediate layer along the edge from the child to the parent. Each of these *virtual* nodes can be seen as having the code of the child. This notion will only be required to prove correctness of the algorithm.

In *Pass1* the algorithm starts at layer 1 (the minimal node resides in layer 0) and proceeds to the topmost layer, *i.e.*, the layer in which the maximal node exists. Procedure *zero(i)* returns a bit pattern of *i* 0's used to initialize the code. Every node in the current layer first gets the bit-wise OR of the codes of the children⁴(lines 6-8 in Figure 9).

Note that if the outdegree of a node is one and the indegree of its only child is also one then the algorithm of the previous section will assign identical codes to these two nodes. Lines 9 to 14 in Figure 9 take care of this contingency. The call to *left_most_1(child(n))* returns the bit position at which a '1' was introduced when *child(n)* was encoded which is the left most 1 in its code let this be *i*. Now line 17 introduces a 1 at position $i - 1$ in *n.code*. This amounts to inserting a virtual child of *n* (see below a description of the manner in which sibling nodes are prefixed by *name_children*).

The rest of *Pass1* (lines 16-30 in Figure 9) is performed for all the layers except the topmost layer. First the length of the prefix to be attached is calculated by *compute_prefix_len* and *curr_code_len* is incremented (lines 17-18 in Figure 9). We will discuss the procedure *compute_prefix_len* after discussing *Pass1*).

Next (lines 16-19 in Figure 9) the nodes with outdegree greater than one are taken and each one is given a distinct prefix. This ensures that a 1 at this bit position can be introduced only by this node (see Lemma 1). When all such nodes have been prefixed the bit-position from which the prefixing of the *pure* nodes can start is stored in the array *unik* (line 31 in Figure 9). Thus the prefixes greater than $2^{unik[layer_no]}$ are used to uniquely prefix nodes with indegree greater than one. The prefixes less than or equal to $2^{unik[layer_no]}$ are used to prefix the nodes with indegree equal to one. Now (lines 27-29 in Figure 9) the nodes at the next higher layer are taken one after another (note that each such node has at least one child in *current_layer*) and their children are prefixed, or named, by *name_children*. This procedure gives distinct prefixes to the codes of the children of *parent* that are not yet prefixed, *i.e.*, the nodes with indegree equal to one. The idea is to make

⁴the nodes in the lower layers to which a node is connected

```

procedure name_children(parent : node, layer : int)
int j
global int curr_code_len, unik[max_noof_layers]
node n
If parent.layer_no  $\neq$  layer + 1 then return
j  $\leftarrow$  unik[layer]
For each node n  $\in$  children(parent) do
    If indegree(n) = 1 then
        n.code  $\leftarrow$   $2^j$  OR n.code
        n.code_len  $\leftarrow$  curr_code_len
        j  $\leftarrow$  j - 1

```

Figure 10: procedure *name_children*

the sibling codes distinct. Note that the *parent* node is in *layer_no* + 1; thus some, but not all of its children may be at deeper layers than *layer_no*. Thus this procedure can be invoked by a child at the deeper layer. We wish to start encoding the siblings when the sibling at the highest layer calls *name_children*. Hence the check at the first line of the procedure.

Now we are in a position to discuss the computation of prefix length in *Pass1*. This task is carried out by the procedure *compute_prefix_len*. It uses the procedure *max_pure_siblings*, which we will discuss first.

The children with indegree equal to one are called *pure* children. Procedure *noof_pure_children*(*n* : *node*) returns the number of *pure* children of a node *n*. Procedure *max_pure_children* uses this procedure to compute the cardinality of the largest set of *pure* sibling nodes at the given layer.

Procedure *compute_prefix_len* first initializes *len* to the number of nodes with indegree greater than one in the layer (*noof_indgr_gt_1*). Next the maximum number of *pure* sibling nodes are determined. This will turn out to be one if there is only one pure node in the layer. Thus *name_children* would prefix the pure node with a 1 in the right most bit position allowed in this layer, say *i*. Now if the pure node's parent has outdegree one then, while

```

procedure max_pure_siblings(j : layer)
int noof_siblings
node m
noof_siblings  $\leftarrow$  0
For each node, m, in layer j + 1 do
    If noof_pure_children(m) > noof_siblings
        then noof_siblings  $\leftarrow$  noof_pure_children(m)
return(noof_siblings)

```

Figure 11: procedure *max_pure_siblings*

encoding the parent Algorithm Encode would (lines 9-14 in Figure 9) try to place a 1 at bit position $i - 1$. Placing a 1 at this bit position may incorrectly relate the parent node to a node at *layer_no* - 1. Hence a check is made to see if *max_pure_siblings* indeed returns 1. If it does then a check is made to see if the parent has outdegree 1. If it does then *len* is incremented by 2 otherwise it is incremented by the value returned by *man_pure_siblings*.

Pass2 is extremely simple. It starts at the layer just below the topmost layer and goes down to the bottom most layer. For each node the *d*(ifference) between the parent's code length and the node's code length is calculated. If the indegree of the node is greater than one then the code is prefixed by *d* zeroes (*zero*(*i* : *int*) returns a string of *i* zeroes) otherwise the code is prefixed by the first *d* bits of the parent (*first*(*d*, *q*) returns the leftmost *d* bits of *q.code*). This completes the discussion of *Algorithm Encode*. In the next section we analyze its time complexity.

5.1 Analysis of Algorithm Encode

Let n be the total number of nodes in the lattice. Let n_i denote the number of nodes in layer i . Further let e be the total number of edges in the lattice and e_i be the number of edges originating from the nodes in layer i . Note that $\sum_i n_i = n$ and $\sum_i e_i = e$. We will assume that the nodes are stored in

```

procedure compute_prefix_len(i : layer)
int len
len  $\leftarrow$  noof_indgr_gt_1(i) /* each impure node must be prefixed uniquely */
If (max_pure_siblings(i) = 1)
    then /* only 1 pure node say n in layer i */
        If outdegree(parent(n)) = 1 then
            len  $\leftarrow$  len + 2
        else len  $\leftarrow$  len + 1
    else len  $\leftarrow$  len + max_pure_siblings(i)
return(len)

```

Figure 12: procedure *compute_prefix_len*

```

procedure Pass2(L: lattice)
For layer_no  $\leftarrow$  max_layer - 1 downto 0 do
    For each node, n, in layer_no do
        d  $\leftarrow$  parent.code_len - n.code_len
        If indegree(n) > 1 then
            prefix  $\leftarrow$  zero(d)
        else
            prefix  $\leftarrow$  first(d, parent.code)
        n.code  $\leftarrow$  concatenate(prefix, n.code)

```

Figure 13: *Pass2*

an array sorted according to layers. The ordering inside a particular layer does not matter. This ordering can be performed in $\mathcal{O}(n + e)$ time from a graph represented using adjacency lists.

We will first analyze *Pass1*. Consider the steps performed by *Pass1* at layer i . First each node gets the bitwise-OR of all its children codes (lines 4 to 15 in Figure 9). This involves exactly e_i steps. After this *compute_prefix_len* is called (line 17 in Figure 9).

Procedure *compute_prefix_len* (Figure 12) first determines the number of *impure* nodes in layer i . This takes one run through the layer and thus takes $\mathcal{O}(n_i)$ time. It then calls *max_pure_siblings*. Procedure *max_pure_siblings* (Figure 11) checks every node which is a child of a node in layer $i + 1$. This takes $\mathcal{O}(e_{i+1})$ time. Thus *compute_prefix_len* takes $\mathcal{O}(n_i + e_{i+1})$ in layer i .

Next in *Pass1* (lines 19 to 25) the impure nodes are encoded. This takes one run through the layer and thus takes $\mathcal{O}(n_i)$. After this (lines 27 to 29) each node in layer $i + 1$ is taken and procedure *name_children* is called each time. Procedure *name_children* (Figure 10) works on all the children of the argument node. Thus in lines 27 to 29 all the children of the nodes in layer i are encountered, this takes $\mathcal{O}(e_{i+1})$ time.

Thus *Pass1* takes $e_i + \mathcal{O}(n_i + e_{i+1}) + \mathcal{O}(n_i) + \mathcal{O}(e_{i+1})$ time. This simplifies to $\mathcal{O}(n_i + e_{i+1})$ time for each layer i . Summing over all layers we get the time complexity of *Pass1* to be $\mathcal{O}(n + e)$.

In *Pass2* every node is visited exactly once so it takes $\Theta(n)$ time. Thus the time complexity of *Algorithm Encode* is $\mathcal{O}(n + e)$. For sparse lattices, $e = \mathcal{O}(n)$. Thus the algorithm is linear in the number of nodes. The experimental results (Figure 16) tally with this analytical result.

This concludes the analysis of *Algorithm Encode*. In the next section we prove its correctness.

6 Correctness

In this section we prove the correctness of our algorithm and show how the encoding leads to the reduction of LUB computation to an OR operation on the codes.

Lemma 1 : If an *impure* node (indegree greater than one) receives a prefix in *Pass 1* such that the i^{th} bit becomes 1, then only its ancestors and descendants can have 1 at the i^{th} position in their final codes.

Proof : In Pass 1 the *impure* nodes are taken separately and assigned distinct prefixes (lines 20-24 in Figure 9). A node with indegree greater than one is the only node in the layer which has a 1 at the unique position. Hence only nodes related to it can get the 1 at that position. The ancestors get it in Pass 1 and the descendants in Pass 2.

□

Lemma 2 : Consider two nodes a, b , such that $a.layer_no < b.layer_no$ but b is not an ancestor of a . Let a' be an ancestor of a in $b.layer_no$. Let $a = a_1, a_2, \dots, a_n = a'$ be a path from a to a' . Finally let all a_i 's and b have an indegree 1. If a' and b are non-sibling nodes then $b.code$ does not subsume $a.code$.

Proof : The proof is by contradiction. Let us assume that $b.code$ subsumes $a.code$ i.e.,

$$\forall i((2^i \& a.code) = 2^i \implies (2^i \& b.code) = 2^i)$$

we will refer to this as the *subsumption supposition*.

Let the LUB of nodes a' and b be node p . Let $a' = a'_1, a'_2, \dots, a'_{n-1}, a'_n = p$, be a path from a' to p . Every a'_i has an indegree = 1. Further no a'_i , except $a'_n = p$, is ancestor of node b . If it was then a'_i would be the LUB of a' and b .

Let $b = b_1, b_2, \dots, b_n = p$ be a path from node b to node p . Consider nodes a'_{n-1} and b_{n-1} . They are children of p , hence they are siblings. They were given distinct prefixes in Pass 1 by name_children (line 26-27), such that $2^i \& a'_{n-1} = 2^i$ (1 in the i^{th} bit position). Also $2^i \& b_{n-1} = 0$. The prefixes of a'_{n-1} and b_{n-1} will be passed down to node a' and b respectively. The prefix of a' will be passed down to a during pass 2. Hence node a has 1 at a position where node b has a 0, contradicting the *subsumption supposition*.

□

We claim that (we will subsequently prove this) Algorithm Encode encodes in a way such that node a *subsumes* node b iff node a is an ancestor of node b (i.e., if there is a '1' at i^{th} position in a node n 's code, then there is a '1' at the i^{th} position of each of its ancestor's code). Thus the OR operation on two codes yields a code that has 1's in these identifying positions. All the nodes whose codes subsume this code are upper bounds of the two initial

nodes. Algorithm Encode imposes a lexical ordering on related nodes according to layers (if $a.\text{layer_no} < b.\text{layer_no}$ and a is related to b then $a.\text{code} \prec b.\text{code}$, where $\prec$ refers to lexical ordering with $0 \prec 1$). Hence the lexically least code of these upper bounds refers to the code of the LUB.

Lemma 3 : Algorithm Encode encodes the nodes such that if two nodes a and b are unrelated (*i.e.*, they do not form an ancestor-descendant pair) then

$$\exists i \mid ((2^i \& a.\text{code}) = 2^i) \text{ and } ((2^i \& b.\text{code}) = 0)$$

Proof : The proof is by contradiction. Let us assume without loss of generality that $a.\text{layer_no} \leq b.\text{layer_no}$. So let us suppose that $b.\text{code}$ subsumes $a.\text{code}$, *i.e.*,

$$\forall i ((2^i \& a.\text{code}) = 2^i \implies (2^i \& b.\text{code}) = 2^i)$$

as before we will refer to this as the *subsumption supposition*.

Can $\text{indegree}(a) > 1$? If yes, then the prefix assigned to node a in Pass 1 was unique (line 21-24). No node which is unrelated to node a can have a 1 in that position. But we have supposed that there exists such a node namely node b . This contradicts the *subsumption supposition*. Hence $\text{indegree}(a)=1$.

Let a' be the ancestor of a in layer $b.\text{layer_no}$. Let $a = a_1, a_2, \dots, a_n = a'$ be a path from a to a' . All a_i 's have $\text{indegree} = 1$, for the following reason. $a_2 = \text{parent}(a)$ has $\text{indegree} = 1$. If it has indegree greater than 1 then it would have been given a unique prefix (*i.e.*, 1 at say the i^{th} bit-position) in Pass 1. Node b is unrelated to node a so it is unrelated to $\text{parent}(a)$ too. So node b can't have a 1 at the i^{th} bit-position. Node a gets 1 in the i^{th} position in Pass2. This contradicts the *subsumption supposition*. Hence $\text{indegree}(\text{parent}(a)) = 1$. Proceeding similarly we have that all nodes along the path from node a to node a' have $\text{indegree} = 1$.

Now consider 2 separate cases - $\text{indegree}(b) > 1$ and $\text{indegree}(b)=1$.

Case 1 : $\text{indegree}(b) > 1$

In Pass1 (lines 20-22 in Figure 9) node b was assigned a distinct prefix, such that the i^{th} bit became 1. a' was also assigned a prefix such that the j^{th} bit became 1. $i \neq j$. b has a 0 at the j^{th} position. But a' has 1 at the j^{th} position and since a is related to a' in Pass2 the prefix of a' will be passed

down to a thus a has 1 at the j^{th} position. (Note that all the nodes along the path from a' to a have indegree 1, so the prefix will not be gobbled up half-way through). This contradicts the *subsumption supposition*. Hence this case is proved.

Case 2 : $\text{indegree}(b) = 1$.

2.a) If a' and b are non-sibling nodes then by Lemma 2 this case is proved.

2.b) If a' and b are sibling nodes then *name_children* assigned them distinct codes in Pass1. a' has 1 at the i^{th} position where node b has 0. Since a is related to a' in Pass 2 the prefix of a' will be passed down to a , thus a also has a 1 at position i . This is a contradiction to *subsumption supposition*. This case is proved.

□

It is clear that the code of a node is subsumed by each of its ancestor's code. Combining this with Lemma 3 we can say that :

Theorem 1 : Algorithm Encode encodes in such a way that only a node's ancestors subsume its code.

In this section we have proved the correctness of Algorithm Encode. We will now proceed to make a few additions to the above algorithm so that we can get the GLB as well.

6.1 GLB Computation

In this section we will discuss procedure *glb_info* which can be called from *Pass1* so that the *same* set of codes yield the GLB as well.

Consider two nodes a and b . If $a.\text{code}$ subsumes $b.\text{code}$ then by Theorem 1 a is an ancestor of b , therefore node b is the GLB of nodes a and b . Similarly if $b.\text{code}$ subsumes $a.\text{code}$ then a is the GLB. This checking takes $\mathcal{O}(1)$ time.

Now suppose that neither a nor b subsume the other then by Theorem 1 nodes a and b do not form an ancestor-descendant pair. Since we are dealing with a lattice the GLB of a and b definitely exists and now it must be an *impure* node. Thus the GLB must have been uniquely prefixed in Pass 1 (the *minimal* node is the only exception to this). Suppose the i^{th} bit became 1 due to the unique prefixing. By Lemma 1 we can say that both a and b have 1 at the i^{th} bit-position.

Let *impure_1s* store the positions at which a 1 is introduced while encoding the *impure* nodes. Thus bit i of *impure_1s* is 1 iff a 1 was introduced at

```

/* node  $n$  is an impure node which introduces a 1 at bit position  $i$  during
Pass1. procedure glb_info stores this information */

```

```

procedure glb_info( $n : \text{node}, i : \text{integer}$ )
global int: impure_ls
global array of nodes : glb[]
    impure_ls  $\leftarrow 2^i \text{ OR } \text{impure\_ls}$ 
    glb[ $i$ ]  $\leftarrow n$ 

```

Figure 14: procedure *glb_info*

bit position i while encoding an *impure* node of the lattice. Moreover let *glb* be an array which stores the *impure* node names or indices, such that *glb*[i] gives the *impure* node whose encoding led to the introduction of 1 at the i^{th} bit-position. We perform these operations in *procedure glb_info*. It is called from *Pass1* when an *impure* node is encountered (line 20 in Figure 9). Note that these operations do not increase the time complexity of Algorithm Encode since *procedure glb_info* takes $\mathcal{O}(1)$ time.

Finally to get the GLB of nodes a and b we first take the bitwise AND of $a.\text{code}$, $b.\text{code}$ and *impure*, let this be $c.\text{code}$. $c.\text{code}$ has 1's at those bit-positions at which 1's were introduced by their common *impure* descendants and ancestors. A node at a higher layer introduces a 1 towards the left of the 1 introduced by a node at a lower layer. We now use *left_most_1* to find out the bit position of the left most 1 in $c.\text{code}$, let this be i . Now *glb*[i] yields the topmost common ancestor or descendant of a and b . We note that *Algorithm Encode* imposes a lexical ordering on related nodes according to layers. So if $a.\text{code} \prec i.\text{code}$ or $b.\text{code} \prec i.\text{code}$ then we don't have the GLB because surely node i resides at a layer higher than that of a or b . So we take the next leftmost bit-position in *impure* and check again until $a.\text{code} \prec i.\text{code}$ and $b.\text{code} \prec i.\text{code}$. If we fail to find such a node or if $c.\text{code}$ has 0's at all the bit positions then the minimal node is the GLB. This search takes $\mathcal{O}(n_{\text{impure}})$ time, where n_{impure} is the number of impure nodes in the lattice. Thus the GLB computation takes $\mathcal{O}(n_{\text{impure}})$ time. We have already noted

Figure 15: Number of nodes vs. Avg. code length

that in most cases in practice n_{impure} is small thus GLB computation is very fast.

Throughout the discussion we dealt with a lattice, so every pair of nodes had a *distinct* GLB and LUB. However this restriction can be relaxed. The algorithm works in exactly the same way on a structure in which a pair of nodes have more than one LUB or GLB. The proof of the algorithm for this structure proceeds along exactly the same lines.

7 Implementation

The algorithm was implemented in C. It was tested on randomly generated posets - A tree was built with each node having a random degree and then edges were randomly added between unconnected nodes. The number of edges added were varied. As expected the code length was minimum when the number of new edges were less. In the Figure 15 the three curves corre-

Figure 16: Number of nodes vs. avg. computation time

spond to different percentages, p , of the total number of nodes with outdegree greater than one. Each curve represents the average number of bits required to encode a lattice with corresponding number of nodes at the specified percent of nodes with outdegree greater than one. It may be noted here that when the total number of nodes and number of nodes with outdegree greater than one were specified, the code length remained remarkably stable for the different lattices produced.

Next we show the time required to compute the codes in Figure 16. The time of computation didn't vary appreciably with the percentage of the nodes with outdegree greater than one, so only one curve has been drawn. It shows the computation time when 9 percent of the nodes had outdegree greater than one.

8 Conclusion

We have presented a simple algorithm for encoding a tree for LUB computation. Then the algorithm was further evolved so that it could be applied to a lattice. This required dividing lattice into layers and finally making further changes in the algorithm itself to take care of the differences between a tree and a lattice. The main difference is that a lattice can have nodes with indegree greater than one, while a tree cannot. We proceeded to analyze and prove correctness of the algorithm formally and then present the experimental results. We noted that the same encoding also yielded the GLB by essentially applying the bitwise-and operation on the codes. Our schemes can be generalized to non-unique GLB's and LUB's. These techniques are can be applied for efficient computation of lattice operations, which are becoming more and more important in programming languages supporting object inheritance.

9 Acknowledgments

Thanks are due to Patrick Lincoln for sending us a small poset and describing their method of poset generation. We owe special thanks to Chilikuri Mohan and V.S. Subrahmanian for thoroughly reading the paper and giving valuable suggestions.

References

- [1] Hassan Aït Kaci, Robert Boyer, Patrick Lincoln and Roger Nasr, "Efficient Implementation of Lattice Operations", *ACM Trans. on Prog. Lang. and Sys*, Vol. 11, No. 1, January 1989, Pages 115-146.
- [2] Aït Kaci, H. "An algebraic-semantic approach to the effective resolution of types equations." *Theor. Comput. Sci.* 45 (1986), Pages 185-215.
- [3] M. Kifer and V.S. Subrahmanian. (1990) "Theory of Generalized Annotated Logic Programming and its Applications", to appear in: *Journal of Logic Programming*. Preliminary version in: *Proc. 1989 North Ameri-*

can Conf. on Logic Programming, (eds. E. Lusk and R. Overbeek), MIT Press.

- [4] J. P Tremblay and R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*, McGraw-Hill, 1987.