

COMPUTING CONGRUENCES EFFICIENTLY

RALPH FREESE

ABSTRACT. For a fixed similarity type which is at least binary, we show there is a linear time algorithm for computing the principal congruence generated by a pair of elements.

Let \mathbf{A} be a finite algebra and let θ be a partition of \mathbf{A} . In this note we give a simple algorithm for computing the smallest congruence containing θ and discuss its efficiency. We begin with the case that θ only contains one nontrivial block and this block has only two elements. That is we are computing $\text{Cg}^{\mathbf{A}}(a, b)$.

Our algorithm, and probably any algorithm, will build up the partition θ starting with $[[a, b]]$, the partition with only one nontrivial block, and ending with $\theta = \text{Cg}^{\mathbf{A}}(a, b)$. As the algorithm proceeds we need to frequently do the following:

- (1) Test if $x \theta y$ for elements x and $y \in A$.
- (2) Combine two blocks of θ .

We would like to represent θ in such a way that each of the above operations could be done as quickly as possible, ideally in constant time.

In the 1960's and 70's this problem was worked on by computer scientists; see [2] and the references therein. They experimented with different representations but found the simple idea of representing each block as a tree (so the partition is a forest) did best. (A *tree* is an ordered set (finite here) with a greatest element known as the *root* such that each interval is a chain. The elements of a tree are sometimes called *nodes* and each element other than the root has a unique upper cover called its *parent*. A *forest* then is the disjoint union of the trees.) Finding the parent of a node can be done in constant time and finding the root in time proportional to the depth of the node. We can test if $x \theta y$ by computing the root of each and seeing if they are equal.

If r and s are roots of distinct blocks, we can join the blocks simply by setting the parent of s to be r . This can be done in constant time. However, joining two blocks can increase the depth of the tree and thus slow down subsequent root findings.

There are two tricks to speed this up:

Weight Rule: When joining, join the small block onto the big block; that is, when joining the block with root r to the block with root s , if the tree with root r has more elements than the one with root s , make r the parent of s ; otherwise make s the parent of r .

Collapsing Rule:: After finding the root r of an element x set the parent of x to r .

Using the Weight Rule guarantees that the depth of (the forest representing) θ never exceeds $\log_2 n$ assuming the initial depth of θ is 1 (every partition has a representation of depth at most 1), since the depth of a node is increased by at most 1 whenever two blocks are joined and, if it is increased, the block it is in after the join must be at least twice as large. Thus by using the Weight Rule alone we can guarantee that the time required for each find operation is $O(\log_2 n)$.

1. PARTITION ALGORITHMS

Let $A = \{0, 1, \dots, n - 1\}$. Since we only need quick access to a node's parent and the size of each block, we can represent θ as a vector V with $V[i]$ the parent of i unless i has no parent (and so is a root), in which case $V[i]$ is negative the size of the block with i . In this scheme the least partition would be represented by the vector $\langle -1, -1, \dots, -1 \rangle$ and the greatest partition could be represented in many ways including the vector $\langle -n, 0, \dots, 0 \rangle$ and $\langle -n, 0, 1, 2, \dots, n - 2 \rangle$. Of course i is a root if and only if $V[i] < 0$.

Algorithm 1 gives a simple recursive procedure for finding the root of any element i and implementing the collapsing rule.¹ Note that $i \theta j$ if and only if $root(i) = root(j)$. As mentioned before the running time for $root$ is proportional to the depth of i in its tree.

```

1 procedure root( $i, V$ )
2    $j := V[i]$ 
3   if  $j < 0$  then return( $i$ )
4   else
5      $r := root(j, V)$ 
6      $V[i] := r$ 
7     return( $r$ )
8   endif
9 endprocedure

```

Algorithm 1: Finding the root and compressing V

Our algorithm to calculate $Cg(a, b)$ will call on a procedure *join-blocks*(r, s, V) which will join the block with root r and the block with root s . Since r is the root of its block, the size of this block is $n_r = -V[r]$ and similarly the size of the block with root s is $n_s = -V[s]$. If $n_r \geq n_s$ the procedure will modify V by setting $V[s] = r$ and $V[r] = -(n_r + n_s)$ and make a similar modification if $n_r < n_s$. This procedure has constant running time and implements the Weight Rule.

¹The simple pseudo computer language used to present our algorithms is described in the first section of Chapter 11 of [1]. It is designed to be easily read by mathematicians. The statement $j := V[i]$, for example, means the variable j is assigned the value $V[i]$.

2. COMPUTING $\text{Cg}(a, b)$

Using our *root* and *join-blocks* procedures we obtain a near linear time algorithm to compute $\text{Cg}^{\mathbf{A}}(a, b)$ for a finite algebra \mathbf{A} . Initially we will take \mathbf{A} to be a unary algebra $\langle A, S \rangle$ where S is a set of k unary operations. We assume $A = \{0, 1, \dots, n-1\}$. We define the following parameters:

$$\begin{aligned} n &= |A| \\ k &= |S| \\ h &= \text{the height of } \text{Cg}(a, b) \text{ in the partition lattice} \\ &= n - \text{the number of blocks of } \text{Cg}(a, b) \end{aligned}$$

and we assume $n \geq 2$ and $k \geq 1$.

Algorithm 2 shows how this is done. We begin by initializing V to correspond to the partition identifying only a and b . We also maintain a list P of pairs of elements of A . Initially P contains only the pair (a, b) .

The first two lines are comments (lines beginning with `%` are comments) describing how to initialize V and P . After the **while**-loop completes V is (the representation of) $\text{Cg}(a, b)$.

```

1 % Initialize  $P$  to be a list containing one element: the pair  $(a, b)$ .
2 % Initialize  $V[b] = a$ ,  $V[a] = -2$ , and  $V[x] = -1$  otherwise.
3 while  $P \neq \emptyset$  do
4     % Let  $(x, y)$  be the first element of  $P$ :
5      $x := \text{first}(\text{first}(P))$ 
6      $y := \text{second}(\text{first}(P))$ 
7     remove  $(x, y)$  from  $P$ 
8     for  $f \in S$  do
9          $r := \text{root}(f(x), V)$ 
10         $s := \text{root}(f(y), V)$ 
11        if  $r \neq s$  then
12            join-blocks( $r, s, V$ )
13            add  $(r, s)$  onto  $P$ 
14        endif
15    endfor
16 endwhile
17 return( $V$ )

```

Algorithm 2: Computing $\text{Cg}(a, b)$

First we prove the correctness of the algorithm.

Theorem 1. *When Algorithm 2 completes, V will represent $\text{Cg}(a, b)$.*

Proof: First note that if we do not use the Collapsing Rule, that is, we omit line 6 from Algorithm 1, the final V may change but the partition it represents will not. In fact it is easy to see that the elements of A which are

roots in V at any point in Algorithm 2 will be the same whether or not line 6 is included in Algorithm 1. So in proving the correctness we may assume that the Collapsing Rule is not used. Let V_1 denote V when the algorithm completes and let θ_1 denote the partition it represents.

Clearly $a \theta_1 b$ and $\theta_1 \leq \text{Cg}(a, b)$. Thus we only need to show

$$(1) \quad \text{if } u \theta_1 v \text{ then } f(u) \theta_1 f(v) \text{ for } f \in S.$$

So suppose that $u \theta_1 v$.

Clearly if at some point in the procedure (u, v) or (v, u) is on P , the **for**-loop starting at line 8 will guarantee that (1) holds.

Suppose that $u \leq v$ in the tree of V_1 in which they lie. We show by induction on the length of the interval $[u, v]$ in this tree that (1) holds. If $u = v$ this is clear. We already mentioned that if $(u, v) = (a, b)$ (or (b, a)) then (1) holds. So at the beginning of the algorithm u and v are in distinct blocks and at some point *join-blocks*(r, s, V) is called where r is the root of u and s is the root of v just before this call. The only new comparabilities introduced by *join-blocks*(r, s, V) are that all of the elements of one of the blocks are less than the root of the other. Also no other part of Algorithm 2 introduces any new comparabilities. Thus $u < v$ implies that $v = s$ and that after doing *join-blocks*(r, s, V), r is a child of s . Since no comparabilities are lost when running Algorithm 2 if line 6 is omitted from Algorithm 1, we have $u \leq r < v$ in V_1 . So by induction $f(u) \theta_1 f(r)$ and since (r, v) or (v, r) gets added to P , $f(r) \theta_1 f(v)$, proving (1) in this case.

Finally suppose $u \theta_1 v$ are arbitrary. Let r be their common root. Then $u \leq r$ and $v \leq r$ and (1) follows from the special case above.

3. TIME ANALYSIS

Recall that h denotes the rank of $\text{Cg}(a, b)$ in the partition lattice. Of course $1 \leq h \leq n - 1$.

First note that the number of pairs that are on P at some point in the algorithm is h . This is because line 13 is executed only when line 12 is. But the *join-blocks* procedure of line 12 will increase the rank of V by one.

Thus each line of the algorithm will be executed at most $h \cdot k$ times, where $k = |S|$. The time required to initialize V is of course proportional to n . All of these lines except lines 9 and 10 take a constant amount of time. Thus the total time is at most $c(hk + n)$ plus the total time spent executing lines 9 and 10, for some constant c .

An analysis of the time spent on finding the roots was done by Tarjan in [2]. He defined the following variant of Ackermann's function on nonnegative integers as follows:

$$A(i, x) = \begin{cases} 2x & \text{if } i = 0 \\ 0 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ A(i - 1, A(i, x - 1)) & \text{if } i \geq 1 \text{ and } x \geq 2 \end{cases}$$

and defined $\alpha(m, n)$ by

$$(2) \quad \alpha(m, n) = \min\{z \geq 1 : A(z, 4\lceil m/n \rceil) > \log_2 n\}$$

He proved that the time spent in finding the roots in Algorithm 2 is at most $m\alpha(m, n)$, where $m = \max\{n, 2hk\}$. Summarizing this:

Theorem 2. *The time required to execute Algorithm 2 is $O(m\alpha(m, n))$, where $m = \max\{n, 2hk\}$. In particular it is $O(2kn\alpha(m, n))$.*

While $\alpha(m, n)$ is not bounded, it nevertheless grows *extremely* slowly. For example $\alpha(m, n) \leq 3$ unless n exceeds

$$2^{2^{2^{\dots^2}}}$$

with 65536 2's. Moreover, if $k \geq n$, the time is $O(kn)$, as we now show.

Lemma 3. *If $1 \leq h < n$, then*

$$(3) \quad h\alpha(2hn, n) \leq 2n.$$

Proof: To establish this we first show

$$(4) \quad A(s, 3t) \geq st \quad \text{for all } s \text{ and } t.$$

This is obvious if either $s = 0$ or $t = 0$. It is shown in [2] that A is increasing in both of its arguments and that it is strictly increasing in its second argument and $A(s, 2) = 4$. We show (4) for $t = 1$ by induction on s :

$$A(s, 3) = A(s - 1, A(s, 2)) = A(s - 1, 4) > A(s - 1, 3) \geq s - 1$$

$A(1, 3t) = 2^{3t} \geq t$ so (4) holds for $s = 1$. If $s > 1$ and $t > 1$

$$\begin{aligned} A(s, 3t) &= A(s - 1, A(s - 1, A(s - 1, A(s, 3t - 3)))) \\ &\geq A(1, s(t - 1)) = 2^{s(t-1)} \geq st \end{aligned}$$

Now (3) is equivalent to

$$A(\lfloor 2n/h \rfloor, 8h) > \log_2 n$$

But $h\lfloor 2n/h \rfloor > n$ because $h < n$, so this follows from (4).

Theorem 4. *Let $\mathbf{A} = \langle A, S \rangle$ be a unary algebra and let $k = |S|$. If $k \geq n$, the time needed by Algorithm 2 to compute $\text{Cg}^{\mathbf{A}}(a, b)$ is $O(kn)$.*

Proof: Since $k \geq n$, $m = 2kh$. Using Theorem 2 and Lemma 3 and the fact that $\alpha(m, n)$ is decreasing in its first variable, we see the time is $O(m\alpha(m, n)) = O(2kh\alpha(2kh, n)) \leq O(2kh\alpha(2hn, n)) \leq O(4kn) = O(kn)$.

4. NONUNARY ALGEBRAS

Let \mathbf{A} be an algebra with n elements and let k_i be the number of basic operations of arity i . Let r be the maximum arity so that $k_i = 0$ for all $i > r$. We say the similarity type is *nonunary* if $k_i > 0$ for some $i > 1$.

The *input size* or *table size* of \mathbf{A} , denoted $\|\mathbf{A}\|$, is

$$\|\mathbf{A}\| = \sum_{i=0}^r k_i n^i$$

Let S denote the set of unary polynomials of \mathbf{A} of the form

$$p(x) = f(a_1, \dots, a_{j-1}, x, a_{j+1}, \dots, a_i),$$

where f is a basic operation of arity i and $1 \leq j \leq i$. Of course the congruences of \mathbf{A} are the same as those of $\langle A, S \rangle$. Note

$$(5) \quad k = |S| = \sum_{i=0}^N i k_i n^{i-1}$$

It is easy to see that $kn/\|\mathbf{A}\| \leq r$ and, assuming n is at least 2, $n^r \geq 2^r$.

Corollary 5. *If \mathbf{A} is a nonunary algebra then $\text{Cg}^{\mathbf{A}}(a, b)$ can be computed in time $O(r\|\mathbf{A}\|)$. In particular, it can be computed in time $O(\|\mathbf{A}\| \log_2 \|\mathbf{A}\|)$.*

Corollary 6. *There is a linear time algorithm for computing $\text{Cg}^{\mathbf{A}}(a, b)$ for the class of algebras of a fixed, finite, nonunary similarity type.*

5. STARTING WITH AN ARBITRARY PARTITION

Let θ be an arbitrary partition on A . We can change Algorithm 2 to compute the congruence generated by θ simply by changing the initialization of P and V . For each block of θ we pick a first element and pair it with all the other elements, putting these pairs on P . For example, if the block is $[a, b, c, d]$ we would put (a, b) , (a, c) , and (a, d) on P . We would initialize V so that it represents θ . It will still be true that the number of pairs that are ever on P during the execution of the algorithm is h , the rank of the congruence generated by θ and so the time required for this algorithm is that given in Theorem 2.

REFERENCES

- [1] Ralph Freese, Jaroslav Ježek, and J. B. Nation, *Free lattices*, Amer. Math. Soc., Providence, 1995, Mathematical Surveys and Monographs, vol. 42.
- [2] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach. **22** (1975), 215–225.

(Ralph Freese) DEPARTMENT OF MATHEMATICS, UNIVERSITY OF HAWAII, HONOLULU, HAWAII, 96822 USA

E-mail address, Ralph Freese: ralph@math.hawaii.edu