Variations on the Common Subexpression Problem

PETER J. DOWNEY

The University of Arizona, Tucson, Arizona

RAVI SETHI

Bell Laboratories, Murray Hill, New Jersey

AND

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT Let G be a directed graph such that for each vertex v in G, the successors of v are ordered Let C be any equivalence relation on the vertices of G. The *congruence closure* C^* of C is the finest equivalence relation containing C and such that any two vertices having corresponding successors equivalent under C^* are themselves equivalent under C^* Efficient algorithms are described for computing congruence closures in the general case and in the following two special cases. (i) G under C^* is acyclic, and (ii) G is acyclic and C identifies a single pair of vertices. The use of these algorithms to test expression equivalence (a problem central to program verification) and to test losslessness of joins in relational databases is described

KEY WORDS AND PHRASES common subexpression, congruence closure, decision procedure, expression equivalence, graph algorithm, lossless join, relational database, theory of equality, unification, uniform word problem

CR CATEGORIES 4 12, 4.33, 4 34, 5.24, 5 25, 5 32

1. *Introduction*

The congruence closure problem is an abstraction of the following expression equivalence problem, sometimes called the uniform word problem for finitely presented algebras: Determine whether an equality $t_1 = t_2$ logically follows from a set of equalities $S = \{s_{11} = s_{12}, s_{21} = s_{22}, \ldots, s_{k1} = s_{k2}\}$, where the s's and t's are expressions constructed from uninterpreted constants and operation symbols. This problem arises in decision procedures for formal theories [6, 11, 13, 14], where it is necessary to determine sets of expressions that are equivalent. A special case of this problem occurs in compiling [4], where it is called the common subexpression problem.

Let $G = (V, E)$ be a directed graph such that for each vertex v in G, the successors of v are ordered. Let C be any equivalence relation on v. The *congruence closure* C^* of C is the finest equivalence relation on V that contains C and satisfies the following property for all vertices v and w:

(1) Let v and w have successors $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_l$, respectively. If $k = l \geq 1$ and $(v_i, w_i) \in C^*$ for $1 \leq i \leq k$, then $(v, w) \in C^*$. In other words, if the corresponding

The research of the first author was supported in part by the National Science Foundation under Grant MCS 75-22557. The research of the third author was supported in part by the National Science Foundation under Grant MCS 75-22870 A02, the Office of Naval Research under Contract N00014-76-C-0688, and a Guggenheim Fellowship Reproduction in whole or in part is permitted for any purpose of the United States government

Authors' present addresses: P J. Downey, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, R. Sethi and R. E Tarjan, Bell Laboratories, Murray Hill, NJ 07974

successors of v and w are equivalent under C^*, then v and w are themselves equivalent under C^*.

In many applications of congruence closure G is acyclic, but we make no such assumption here. We present an algorithm that requires $O(m \log m \log n/\log k)$ time and $O(km)$ space in the worst case, where n is the number of vertices in G, m is the number of edges, and k is a parameter that can be chosen arbitrarily.[1] The algorithm can also be implemented to run in $O(m \log m)$ time and $O(m)$ space on the average. We describe this algorithm in Section 2. In Section 3 we present an algorithm with a worst-case running time of $O(m)$ for the special case when G is acyclic under the equivalence relation C^*. We also present an algorithm for the special case when G is acyclic and C identifies a single pair of vertices. In Section 4 we discuss two problems related to congruence closure. In Section 5 we outline uses of our algorithms in decision procedures for formal theories, and in Section 6 we apply one of our algorithms to the problem of testing a relational database for the "lossless join" property [1].

2. *A Fast Algorithm for Congruence Closure*

2.1 A SIMPLE ALGORITHM. Here is a simple algorithm for computing congruence closure. The algorithm is a straightforward extension of the "value number" method of Cocke and Schwartz [4] for finding common subexpressions.

Initialization Let $C_0 = C$ and $i = 0$.

General step Number the equivalence classes in C_i consecutively from 1 Assign to each vertex v the number $\alpha(v)$ of the equivalence class containing v For each vertex v construct a *signature* $s(v) = (\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_k))$, where $v_1, v_2, \ldots, v_k$ are the successors of v The signature of each vertex is a sequence of integers in the range $1 \le i \le n$

Group the vertices into classes of vertices having equal signatures Let C_{i+1} be the finest equivalence relation on V such that two vertices equivalent under C_i or having the same signature are equivalent under C_{i+1}

If $C_{i+1} = C_i$, let $C^* = C_{i+1}$. Otherwise replace i by $i + 1$ and repeat the general step.

It is easy to verify the correctness of this algorithm. Since every execution of the general step except the last must merge at least two equivalence classes, the number of executions of the general step is at most n. The bottleneck in the algorithm is the grouping together of vertices having equal signatures. Using a lexicographic sort as described in [2], this step requires $O(m)$ time, and the entire algorithm requires $O(mn)$ time in the worst case. Alternatively, using a hash table [10], this step requires $O(m)$ time on the average, and the entire algorithm requires $O(mn)$ time on the average.

This simple algorithm does too much work. In each succeeding execution of the general step, it is only necessary to consider vertices having at least one successor whose equivalence class has changed during the previous execution of the general step. Using this observation in combination with a "modify the smaller half" idea, we can obtain a faster algorithm. First, however, we show that we can restrict our attention to graphs with outdegree bounded by 2.

2.2 REDUCTION TO OUTDEGREE 2. Let $G = (V, E)$ be an arbitrary directed graph and let C be an equivalence relation on V. Construct $G_1 = (V_1, E_1)$ and C_1 as follows (see Figure 1).

$$V_1 = V \cup \{x_j \mid 2 \le j \le n\} \cup \{w_i(v) \mid v \in V \text{ and } 2 \le i \le d(v)\},$$

where $d(v)$ is the outdegree of v. Let

$$\begin{aligned} E_1 = {} & \{(v, v_1) \mid v \in V \text{ and } v_1 \text{ is the first successor of } v\} \\ & \cup \{(v, w_2(v)) \mid v \in V \text{ and } d(v) \ge 2\} \\ & \cup \{(w_i(v), v_i) \mid v \in V, 2 \le i \le d(v), \text{ and } v_i \text{ is the } i\text{th successor of } V\} \\ & \cup \{(w_i(v), w_{i+1}(v)) \mid v \in V \text{ and } i < d(v)\} \\ & \cup \{(w_{d(v)}(v), x_{d(v)}) \mid v \in V \text{ and } d(v) \ge 2\}. \end{aligned}$$

[1] Throughout this paper we assume that n is $O(m)$ If $n > 2m$, G contains at least one isolated vertex. Isolated vertices can be deleted from G without affecting the problem in any substantial way.

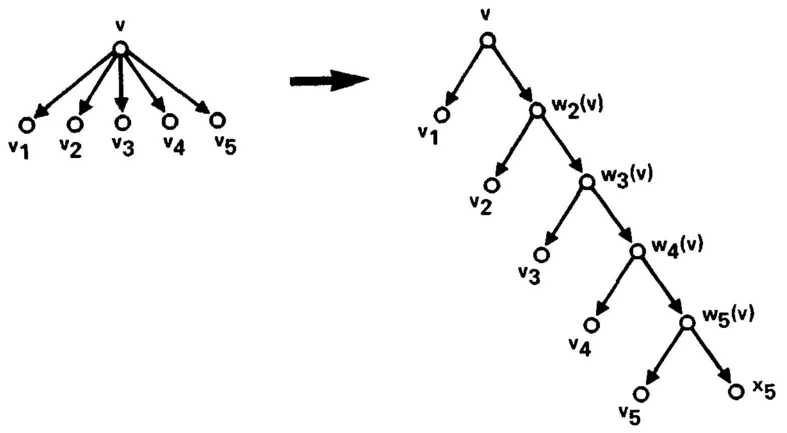

FIG 1 Reduction to outdegree 2

For a vertex $v \in V$ with $d(v) \geq 2$, the order of the successors of v in G is v_1, $w_2(v)$. For a vertex $w_i(v)$ with $i < d(v)$, the order of successors is v_i, $w_{i+1}(v)$. For a vertex $w_{d(v)}(v)$, the order is $v_{d(v)}$, $x_{d(v)}$. Let $C_1 = C \cup \{(v, v) \mid v \in V_1 - V\}$.

It is easy to show that C_1^* restricted to V is C^*. Thus to compute C^* it is sufficient to construct G_1 and C_1, compute C_1^*, and compute C_1^* restricted to V. The first and third steps take $O(m)$ time, and G_1 contains $O(m)$ vertices and edges. Note that the first successor of every vertex in G_1 is a vertex in G.

In the rest of Section 2 we consider a graph G_1 in which all vertices have outdegree at most 2.

2.3 DATA STRUCTURES. In Section 2.4 we describe a fast algorithm for computing the congruence closure of C_1 or, in general, of any equivalence relation defined on a graph with no vertex of outdegree exceeding 2. The algorithm uses two data structures and in addition manipulates two sets. One data structure represents the equivalence classes defined by the current equivalence relation. Each equivalence class has a *name* which is an integer from one to $| V_1 |$. Associated with each equivalence class is a list of the vertices that have at least one successor in the class. (A vertex with two successors in the equivalence class may appear twice in the list.) The algorithm performs three operations on equivalence classes:

find(v): Return the name of the equivalence class containing v.
list(e): Return the list of vertices with at least one successor in equivalence class e.
union(e_1, e_2): Combine e_1 and e_2 into a single equivalence class, named e_1.

The algorithm initializes this data structure to represent the equivalence classes of the given equivalence relation C_1.

The algorithm uses a second data structure, called a *signature table*, to store vertices and their signatures. Each signature is either a single integer or an ordered pair of integers (i, j) in the range $1 \leq i \leq n$, $1 \leq j \leq | V_1 |$. (Vertices with no successors do not appear in the signature table.) The algorithm performs three operations on the signature table.

enter(v): Store v with its current signature in the signature table.
delete(v): Delete v from the signature table if it is present.
query(v): If some vertex w in the signature table has the same signature as v, then return w; otherwise return Λ.

Initially the signature table is empty. The algorithm maintains the signature table so that any given signature appears at most once.

In addition to these data structures, the algorithm manipulates two sets: *pending*, a list of vertices to be entered in the signature table; and *combine*, a list of pairs of vertices whose equivalence classes are to be combined.

2.4 A FAST ALGORITHM. Informally, the fast algorithm is like the simple one proposed previously, with the following differences. Vertices and their current signatures are stored in the signature table. Each time the signature of a vertex changes because the equivalence class of one of its successors changes, the vertex is reentered in the signature table. If some other vertex has the same signature, the equivalence classes of the vertices are combined. When two equivalence classes are combined, a "modify the smaller half" strategy is used: of the two old classes, the name of the one with more predecessors is given to the new class. Thus the only signatures that change when two classes are combined are those of vertices with a successor in the old class with fewer predecessors. The algorithm appears below in Algol-like notation.

Fast Congruence Closure Algorithm

```
pending = $\{v \in V_1 \mid d(v) \geq 1\}$,
while pending $\neq \emptyset$ do
    combine = $\emptyset$;
    for each $v \in$ pending do
        if query($v$) = $\Lambda$ then enter($v$)
        else add ($v$, query($v$)) to combine fi od;
    pending := $\emptyset$,
    for each ($v$, $w$) $\in$ combine do
        if find($v$) $\neq$ find($w$) then
            if |list(find($v$))| < |list(find($w$))| then
                comment renaming the equivalence class with fewer predecessors causes fewer signatures to
                    be reentered,
                for each $u \in$ list(find($v$)) do
                    delete($u$), add $u$ to pending od;
                union(find($w$), find($v$))
            else
                for each $u \in$ list(find($w$)) do
                    delete($u$), add $u$ to pending od;
                union(find($v$), find($w$)) fi fi od od;
```

2.5 RUNNING TIME. In order to estimate the running time of the algorithm, we need to bound the time taken to perform the operations on each data structure. We show that all operations on the equivalence classes and on the sets *pending* and *combine* require $O(m \log m)$ time. Moreover, there are at most $O(m \log m)$ operations on the signature table. Thus the time to perform the signature table operations gives a bound on the total running time of the algorithm.

We begin by bounding the number of additions to *pending* during the course of the algorithm. There are at most $|V_1|$ initial additions to *pending*. Each vertex appears in at most two predecessor lists, so the total size of all predecessor lists is at most $2|V_1|$. Each time two predecessor lists are merged because the corresponding equivalence classes are combined, only the vertices on the smaller list are added to *pending*. In other words, each time a vertex is added to *pending* (other than initially), the length of one of the predecessor lists containing it at least doubles. It follows that there are at most $|V_1| + 2|V_1| \log 2|V_1|$ (see footnote 2) additions to *pending*.

Next we bound the time of the operations on equivalence classes. There are at most $|V_1| - 1$ *union* operations, since each *union* operation reduces the number of equivalence classes by one, and there are at most $|V_1|$ equivalence classes originally. The number of *list* operations is bounded by a constant times the number of *union* operations and is thus $O(|V_1|) = O(m)$. The number of *find* operations is bounded by a constant times the number of additions to *combine*, which in turn is bounded by the number of additions to *pending*. Thus the number of *find* operations is $O(m \log m)$.

[2] All logarithms in this paper are base 2

TABLE I. Time and Space Bounds for the Fast Congruence Closure Algorithm

	Degree-2 graphs		General graphs	
Implementation	Time	Space	Time	Space
Balanced binary tree	$O(n(\log n)^2)$	$O(n)$	$O(m(\log m)^2)$	$O(m)$
$n \times O(m)$ array	$O(n \log n)$	$O(n^2)$	$O(m \log m)$	$O(mn)$
Hash table	$O(n \log n)$ average	$O(n)$	$O(m \log m)$ average	$O(m)$
Trie	$O(n(\log n)^2/\log k)$	$O(kn)$	$O(m \log m \log n/\log k)$	$O(km)$

To implement the equivalence class operations, we use the fast set union algorithm analyzed in [20]. In addition, we store with each equivalence class a circularly linked list of its predecessors and an integer specifying the size of this list. With this representation the time for a union operation is $O(1)$. The time for a *list* operation is also $O(1)$; we return a pointer to the list of predecessors. Thus the total time for *union* and *list* operations is $O(m)$. The $O(m \log m)$ *find* operations require $O(m \log m)$ time [20].[3]

Finally, we examine the time taken for signature table operations. The number of these operations is bounded by a constant times the number of additions to *pending* and is thus $O(m \log m)$. In order to represent the signature table, we divide it into two parts, one for signatures that are single integers and one for signatures that are ordered pairs. To store the first part of the table, we use an array of size n, with one position for each possible signature. Each operation on this part of the table takes $O(1)$ time.

The second part of the table requires storage of ordered pairs (i, j) in the range $1 \le i \le n$, $1 \le j \le |V_1|$. To store this part of the table, we can use a standard data structure such as a balanced binary tree [10], an $n \times |V_1|$ array, or a hash table [10]. With a binary tree each table operation requires $O(\log m)$ time, but the storage necessary is only $O(m)$. In this case we get a worst-case time bound of $O(m(\log m)^2)$ and a space bound of $O(m)$ for the entire congruence closure algorithm. With an $n \times |V_1|$ array each table operation requires $O(1)$ time, but $O(mn)$ storage is necessary; we get an $O(m \log m)$ worst-case running time and an $O(mn)$ storage requirement. With a hash table each operation requires $O(1)$ time on the average and the storage is $O(m)$, giving an $O(m \log m)$ average running time with $O(m)$ storage in the worst case.

We can also use the trie method of [19] to maintain the ordered pairs. In this case, for any value of k chosen in advance we need $O(\log n/\log k)$ time per table operation using $O(km)$ storage. Thus we get a worst-case running time of $O(m \log m \log n/\log k)$ with $O(km)$ storage. This method dominates both the binary tree method (choosing $k = n^{\Omega(1)}$) and the array method (choosing $k = O(1)$).

Table I summarizes the total time and space requirements of the fast congruence closure algorithm with each of these implementations, both for graphs of outdegree bounded by 2 and for general graphs.

3. *The Acyclic Case*

Let $G = (V, E)$ be a directed graph and let R be an equivalence relation on G. A *cycle* in *G under R* is a sequence of vertices $v_0, v_1, v_2, \ldots, v_{2k} = v_0$, such that $(v_{2i-2}, v_{2i-1}) \in E$ and $(v_{2i-1}, v_{2i}) \in R$ for $1 \le i \le k$. Informally, G has a cycle under R if the graph formed from G by contracting the equivalence classes of R has a cycle. If G does not have a cycle under R, then G itself must be acyclic.

3.1 Acyclic Congruence Closure.

Let $G = (V, E)$ be a directed graph and let C be an equivalence relation on G. In this section we consider the problem of computing C^* for the special case when G under C^* is acyclic. (Note that G under C^* may contain cycles

[3] An alternative way to represent the equivalence classes is to use the "simple" set union method described in [2, Sec. 4.6] This method also gives an $O(m \log m)$-time bound for the equivalence class operations However, the method is never faster and sometimes much slower than the method we advocate, and in addition uses more storage

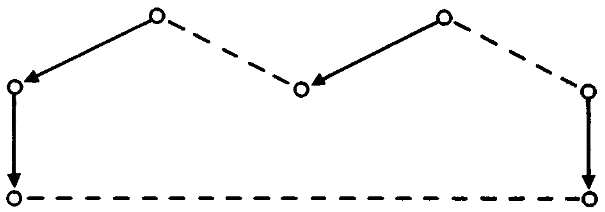

FIG 2 Edges of G are represented by solid arrows The three equivalence classes of C are given by the dotted lines Although G under C is acyclic, G under C^* has a cycle.

even if G under C does not; see Figure 2. In Section 3.3 we give a sufficient condition for G under C^* to be acyclic.)

If G under C^* is acyclic, then the equivalence classes $e_1, e_2, \ldots, e_k$ of C^* can be ordered topologically, that is, ordered so that no edge leads from a vertex in e_i to a vertex in e_j if $j \leq i$ [9]. Our algorithm identifies the classes of C^* in reverse topological order. Once equivalence classes $e_i, e_{i+1}, \ldots, e_k$ are known, any vertex all of whose successors appear in $e_i, e_{i+1}, \ldots, e_k$ has its signature permanently fixed. Informally, our algorithm is as follows. The algorithm labels an equivalence class (and all vertices in it) *fixed* when the permanent signatures of all vertices in the class are known; after a class is fixed it will never be merged with another class.

Acyclic Congruence Closure Algorithm

Initialization Let all equivalence classes of C be unfixed and let all vertices in V be unfixed and ungrouped.

General Step

1 Find an unfixed equivalence class e such that every vertex in e has only fixed successors.
2 Fix all vertices in e, and fix e by assigning e a number
3 Let s be the set of ungrouped vertices with one or more successors, all of which are fixed.
4 Group the vertices of s into groups having equal signatures
5 Combine the equivalence classes corresponding to vertices in s having equal signatures

Repeat the general step until either all vertices become fixed or no unfixed equivalence class has all its successors fixed (in the latter case G under C^* contains a cycle)

3.2 CORRECTNESS AND IMPLEMENTATION. The correctness of this algorithm depends on the fact that at the end of a given execution of the general step, all vertices with only fixed successors have been grouped by signature, and the corresponding equivalence classes have been combined. No vertex with at least one unfixed successor at the end of a general step can ever have the same signature as a vertex with no unfixed successors at the end of the step. Thus the equivalence class selected in step 1 of the next general step is a class of C^* and can properly be fixed.

Implementing this algorithm so that it runs in $O(m)$ time requires some subtlety. In the remainder of this section we discuss the important points. The appendix contains a detailed Algol-like description of the algorithm.

Successor count. For each vertex we maintain a count of the unfixed successors of the vertex. Each time a vertex is fixed, the count of each of its predecessors is decreased by one. When the count of a vertex becomes zero, it is placed in s. The total amount of time for executing step 3 and maintaining successor counts is $O(m)$.

Vertex partition. So that the grouping of vertices with equal signatures in step 4 may be efficient, we maintain a partition of the vertices with at least one unfixed successor. This partition has the following property. If v and w are two vertices with successors $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_l$, respectively, then v and w are in the same part of the partition if and only if for all i such that either v_i or w_i is fixed, both v_i and w_i are fixed and both are in the same equivalence class.

Initially, the partition has one part containing all vertices with at least one successor. When an equivalence class e becomes fixed and numbered, we update the partition in the following way. For each value of i such that the ith successor of some vertex is in e, we determine the set q of *all* vertices whose ith successor is in e. For each part p of the partition whose intersection with q is nonempty we divide p into two parts: $p \cap q$ and $p - q$. This updating requires time proportional to the number of edges entering e if we represent the partition as a linked structure (see [16, 17]). Furthermore, it is possible with this representation to determine the part containing a given vertex in constant time.

When step 4 is about to be executed, two vertices in s have the same signature if and only if they are in the same part of the partition. Thus we can execute step 4 by computing $p \cap s$ for each part p of the partition containing at least one vertex in s and replacing p in the partition by $p - s$. This step requires $O(|s|)$ time [16, 17]. The total time for all executions of step 4 and all updating of the partition is $O(m)$.

Representation of equivalence classes. What remains is to represent the equivalence classes so that steps 1, 2, and 5 are fast. One approach is to use the set union method described in [20] and used in Section 2. For each equivalence class we must additionally maintain a list of the vertices in it and a count of the unfixed ones. The time bound resulting for the entire algorithm is $O(n\alpha(n, n) + m)$, where $\alpha(n, n)$ is a functional inverse of Ackermann's function [20].

However, we can do somewhat better by using the main idea in Paterson and Wegman's unification algorithm [15]. We maintain a graph G' that consists of G and some additional undirected edges that join vertices in the same equivalence class. We initialize the undirected edges of G' so that they form a forest, each tree of which defines an equivalence class of the initial equivalence relation C. For each group $\{v_1, v_2, \ldots, v_k\}$ of vertices having the same signature, we combine equivalence classes in step 5 by adding undirected edges $\{v_1, v_2\}$, $\{v_1, v_3\}$, $\{v_1, v_4\}$, $\{v_1, v_5\}$, $\ldots$, $\{v_1, v_k\}$ to the graph. Then all executions of step 5 require a total of $O(n)$ time. The connected components of G' defined by the undirected edges are the current equivalence classes.

Finding equivalence classes to be fixed. We use a depth-first search to carry out all executions of step 1. The search begins at some vertex v_1. If v_1 has no unfixed successors, we traverse some undirected edge incident to v_1. We continue in this way, scanning the equivalence class containing v_1, until we find a vertex w_1 in the class that has an unfixed successor v_2. We then postpone scanning the equivalence class containing v_1 and begin scanning the equivalence class containing v_2. We continue the search in this way, producing a stack of partially scanned equivalence classes of which the top class is currently active. When we completely scan an equivalence class, we remove it from the stack and execute steps 2–5. If the search traverses an undirected edge leading to a postponed class, or a directed edge leading from the current class either to the current class or to a postponed class, the search stops. In this case G under C^* contains a cycle.

It is useful to discuss a few more details of this process. When we reach an unscanned vertex v, we assign v a *level* equal to the number of partially scanned equivalence classes, and we place v on a stack. Assigning levels makes it easy to test whether a newly traversed edge produces a cycle. On the stack, vertices of the same level are consecutive. When the search backs up along an edge leading from a lower to a higher level, the equivalence class at the higher level has been completely scanned and all its vertices are on top of the stack. We delete all these vertices from the stack, retrieving the equivalence class e for the next execution of steps 2–5.

An Algol-like version of the entire algorithm appears in the appendix. The time required by the depth-first search is $O(m)$, and thus the total time required is $O(m)$. For further discussion of various parts of the algorithm, see [15–17].

3.3 A Sufficient Condition for Acyclicity. There is at least one important special case in which we can guarantee that G under C^* is acyclic and thus that the algorithm of this section will work. Suppose G is acyclic. For each vertex v in G define the *level* of v as the length of the longest path in G starting at v. That is, $level(v) = \max(\{0\} \cup \{level(w) + 1 \mid (v, w) \in E\})$.

THEOREM 1. *Suppose that G is acyclic and that any two vertices equivalent under C have the same level. Then G under C^* is acyclic.*

SKETCH OF PROOF. Consider the simple algorithm of Section 2 for computing congruence closure. An easy proof by induction shows that for any i, two vertices equivalent under C_i have the same level. Thus any two vertices equivalent under C^* have the same level. Since every edge of G leads from a smaller to a larger level, it follows that G under C^* is acyclic. □

COROLLARY 1. *If G is acyclic and C is the identity relation, then G under C^* is acyclic.*

3.4 THE CASE OF A SINGLE EQUIVALENCE. There is an interesting special case to which the algorithm of this section does not apply, but that still does not require the full generality of the algorithm in Section 2. Suppose that G is acyclic, C identifies a single pair of vertices x, y, and we wish to compute C^* even if G under C^* contains cycles. The following one-pass version of the fast algorithm of Section 2 solves this problem. We assume that the graph is preprocessed as in Section 2 to reduce the maximum indegree to 2.

The algorithm processes each vertex once, assigning a number to each vertex so that two vertices are equivalent under C^* if and only if they receive the same number. To decide how to assign numbers, the algorithm maintains a signature table as in Section 2. In order to avoid having to renumber vertices, the algorithm processes vertices in a special order.

Congruence Closure Algorithm for a Single Equivalence

```
Step 1  Assume without loss of generality that $x$ has level no greater than $y$ Arrange the vertices of $G$ in reverse
        topological order so that the following two conditions hold (i) $y$ follows $x$ in the ordering; (ii) if $z$ is any
        vertex such that there is no path from $x$ or $y$ to $z$, then $z$ follows both $x$ and $y$
Step 2  Process the vertices in the order given by step 1 To process a vertex $v$, proceed as follows·
        (a) Compute the signature of $v$
        (b) Look up $v$ in the signature table If some vertex $w$ has the same signature as $v$, assign $v$ the same
            number as $w$ Otherwise, if $v \neq y$, assign $v$ a previously unassigned number; if $v = y$, assign $v$ the same
            number as $x$
```

The following lemma expresses the crucial point that guarantees the correctness of this algorithm.

LEMMA 1. *When vertex y is processed, its signature does not appear in the signature table.*

PROOF. An easy induction as in Theorem 1 shows that up to and including the time y is processed, two vertices have the same signature only if they have the same level. Suppose a vertex z with the same signature as y is found in the signature table when y is processed. Then y and z must have the same level. However, since z was processed before y, there must be a path from either x or y to z. This is a contradiction. □

It follows from Lemma 1 that the algorithm assigns two vertices the same number if and only if they have the same signature or they are x and y. This means that the numbering given by the algorithm corresponds to C^*, and no renumbering is necessary.

The time required by the algorithm is dominated by the time spent accessing the signature table; the method requires $O(m)$ time plus $O(m)$ table operations. If a trie is used to represent the table, $O(m \log n/\log k)$ time and $O(km)$ space are required in the worst case. If a hash table is used, $O(m)$ average time and $O(n)$ worst-case space are required.

4. *Related Problems*

In this section we examine two problems related to congruence closure.

4.1 SYMMETRIC CONGRUENCE CLOSURE. Let $G = (V, E)$ be a directed graph and let C be an equivalence relation on V. The *symmetric congruence closure* C^s of C is the finest equivalence relation on V containing C and satisfying the following property: If v and w are vertices with successors $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_l$, respectively, $k = l \geq 1$, and there is a permutation π on $\{1, 2, \ldots, k\}$ such that $(v_i, w_{\pi(i)}) \in C^s$ for $1 \leq i \leq k$, then $(v, w) \in C^s$. In other words, we regard the successors of each vertex in G as unordered. Finding symmetric congruence closures is required in order to solve the expression equivalence problem (which we discuss in the next section) when all operations are commutative. The acyclic symmetric congruence closure problem is also a generalization of the problem of testing isomorphism of rooted trees [2].

The algorithms of Sections 2 and 3 can easily be modified to find symmetric congruence closures without degrading the resource bounds. The main idea is that when computing the signature of a vertex, we first sort its successors by the numbers of their equivalence classes. In the acyclic congruence closure algorithm of Section 3, this sorting takes place implicitly; see [16, 17]. The only difficulty with this approach is that the transformation from general graphs to outdegree-2 graphs given in Section 2 does not preserve symmetric congruence closures; however in most applications the graphs have outdegree 2 and this is not a problem. We can easily generalize the algorithms further to allow some vertices of G to have ordered successors and some to have unordered successors.

4.2 UNIFICATION. Let $G = (V, E)$ be a directed graph and let C be an equivalence relation on V. The *unifier* C^u of C is the finest equivalence relation on V containing C and satisfying

(i) (consistency) if $(v, w) \in C^u$, then $d(v) = d(w)$ or at least one of $d(v)$, $d(w)$ is zero; and
(ii) if v and w are vertices with successors $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_l$, respectively, $k = l \geq 1$, and $(v, w) \in C^u$, then $(v_i, w_i) \in C^u$ for $1 \leq i \leq k$.

Because of the consistency condition, a unifier may not exist. The problem of computing unifiers is a directional dual of the congruence closure problem; it arises in testing equivalence of finite automata [8] and in determining a most general set of substitutions to make two expressions equal [15]. Hopcroft and Karp [8] have given an $O(m\alpha(m, n))$-time algorithm for the general problem based on the fast set union method of [20], and Paterson and Wegman [15] have given an $O(m)$-time algorithm for the case when G under C^u is acyclic. We have adapted some of Paterson and Wegman's ideas for our acyclic congruence closure algorithm. The unification problem seems somewhat easier than the congruence closure problem, perhaps because in the former, two vertices must be equivalent if *some* pair of corresponding predecessors are equivalent, whereas in the latter, two vertices must be equivalent if *all* pairs of corresponding successors are equivalent.

5. *Testing Expression Equivalence*

The most important application of congruence closure is in solving the following expression equivalence problem, sometimes called the *uniform word problem for finitely presented algebras*: Determine whether an equality $t_1 = t_2$ logically follows from a set of equalities $S = \{s_{11} = s_{12}, s_{21} = s_{22}, \ldots, s_{k1} = s_{k2}\}$, where the s's and t's are expressions constructed from uninterpreted constant and operation symbols. This problem is central to program verification and has been studied by a number of authors [5, 11, 13, 14, 18], all of whom have suggested algorithms based implicitly on finding a congruence closure.

The idea is as follows. We construct a graph G with one vertex for each variable, each operation symbol, and each subexpression occurring in the equalities. Corresponding to each variable x is a vertex $v(x)$ of outdegree 0; corresponding to each operation symbol θ is a vertex $v(\theta)$ of outdegree 0; corresponding to each subexpression $r = \theta(r_1, r_2, \ldots, r_j)$ is a vertex $v(r)$ with successors $v(\theta)$, $v(r_1)$, $v(r_2)$, $\ldots$, $v(r_j)$ (in order). Constructing this graph requires time linear in the total size of the set of equalities if the variables and operation symbols are all encoded by reasonably small integers (see [13]).

As the equivalence relation C we use the finest equivalence relation containing $(v(s_{i1}), v(s_{i2}))$ for $1 \le i \le k$. Then we compute the congruence closure C^* of G under C. The equality $t_1 = t_2$ follows logically from $S = \{s_{11} = s_{12}, s_{21} = s_{22}, \ldots, s_{k1} = s_{k2}\}$ if and only if $v(t_1)$ and $v(t_2)$ are equivalent under C^* (for a proof see [5, 13]). Thus we can solve the expression equivalence problem in time linear in the size of the set of equalities plus the time necessary to compute a single congruence closure. The algorithm easily generalizes to the problem of determining which equalities among a set $T = \{t_{11} = t_{12}, t_{21} = t_{22}, \ldots, t_{l1} = t_{l2}\}$ logically follow from a set $S = \{s_{11} = s_{12}, s_{21} = s_{22}, \ldots, s_{k1} = s_{k2}\}$.

The graph G representing a set of expressions is acyclic, but G under C^* need not be (see Figure 2). However, if the problem we wish to solve is that of identifying identical subexpressions, then S is the empty set, and G under C^* is acyclic by Corollary 1. The acyclic congruence closure algorithm of Section 3 can thus be used to find common subexpressions in linear time. (The value number method of Cocke and Schwartz [4] requires linear time on the average to find common subexpressions.)

Downey and Sethi [6] have considered a version of the expression equivalence problem that arises in verifying a restricted class of array assignment programs. In their application, S contains only a single equality, and the congruence closure algorithm for a single equivalence described in Section 3 can be used. Oppen [14] has constructed a decision algorithm for the theory of recursively defined data structures which combines acyclic congruence closure with acyclic unification.

As described in Section 4, we can easily modify the congruence closure algorithms to allow commutativity of operations. If θ is commutative, we construct G so that an expression $\theta(r_1, r_2)$ is represented by a vertex $v(\theta(r_1, r_2))$ with two ordered successors, $v(\theta)$ and a new vertex x; x has two unordered successors, $v(r_1)$ and $v(r_2)$.

Associativity is much harder to handle than commutativity. Given a set of equalities $S = \{s_{11} = s_{12}, s_{21} = s_{22}, \ldots, s_{k1} = s_{k2}\}$ of expressions constructed from constants and a single binary associative operation $\cdot$, the problem of deciding whether another equality $t_1 = t_2$ follows from S is called the *uniform word problem for semigroups*; it is undecidable [21]. Even if S is fixed and only $t_1 = t_2$ is allowed to vary, the problem is still undecidable. If $\cdot$ is commutative, we obtain the uniform word problem for commutative semigroups, which is complete in exponential space [3]. Even the following problem is NP-complete [7]: Given a set of expressions E constructed from constants and a single commutative, associative operation, determine the minimum number of operations needed to evaluate all expressions in E.

6. *Testing for the Lossless Join Property*

The problem of expression equivalence arises in disguised form in the study of relational databases. Aho, Beeri, and Ullman [1] have defined an important property of such databases called the "lossless join" property. They present an $O(n^4)$-time algorithm for testing this property, where n is the size of the input; a more careful implementation by Liu and Demers [12] has an $O(n^3)$ bound. The algorithm works by equating symbols in a certain tableau. In this section we describe the application of congruence closure to this problem; the fast algorithm of Section 2 gives us an $O(n^2 \log n)$-time test for losslessness.

We assume that the reader is familiar with [1]; we present the Aho–Beeri–Ullman algorithm without commenting upon its correctness or significance. The algorithm is given as input a set $A = \{1, 2, \ldots, h\}$ of *attributes*; a collection $R_1, R_2, \ldots, R_m$ of *relation schemes*, each a nonempty subset of A; and a collection $r_1, r_2, \ldots, r_l$ of *functional dependency rules*, each rule r_i of the form $X_i \to Y_i$, where X_i and Y_i are nonempty subsets of A. The algorithm begins by defining an $m \times h$ matrix T as follows:

$$T(i, j) = \begin{cases} a_j & \text{if } j \text{ is in } R_i, \\ b_{ij} & \text{otherwise,} \end{cases}$$

where all the a's and b's are distinct symbols.

The algorithm proceeds by declaring symbols of $T(i, j)$ equivalent, using the following rule:

(D) If $X_p \rightarrow Y_p$ is a functional dependency rule and i, j are two rows of T such that $T(i, q)$ is equivalent to $T(j, q)$ for all $q \in X_p$, then declare $T(i, r)$ and $T(j, r)$ equivalent for all $r \in Y_p$.

The algorithm applies (D) until it is no longer applicable; that is, it computes the finest equivalence relation on the symbols of T that satisfies (D). Let us call this equivalence relation the *dependency closure* D^* of T. Aho, Beeri, and Ullman show that the relational database scheme defined by $R_1, R_2, \ldots, R_m$ with functional dependencies $r_1, r_2, \ldots, r_l$ has the lossless join property if and only if some row of T has all its symbols equivalent under D^* to a's. (Note that (D) only declares symbols in the same column of T equivalent; thus each b_{ij} in T has a unique a, namely, a_j, with which it may become equivalent.)

We can transform the problem of computing a dependency closure into that of computing a congruence closure on an appropriate graph. First it is convenient to apply a trick suggested by Ullman to modify the functional dependency rules so that each has only a single attribute on its right-hand side. If $X_i \rightarrow Y_i$ is a functional dependency rule such that Y_i contains at least two attributes, we introduce a new attribute x and replace $X_i \rightarrow Y_i$ by the rule $X_i \rightarrow x$ and the set of rules $\{x \rightarrow y \mid y \in Y_i\}$. Applying this construction to all the original rules increases the number of attributes by at most l and increases the total length of the functional dependency rules by at most $2l$. Furthermore, if D^* is the dependency closure of the tableau T corresponding to the original scheme, and D_1^* is the dependency closure of the tableau T_1 corresponding to the new scheme, two symbols of T are equivalent under D^* if and only if they are equivalent under D_1^*. (We leave the routine proof of this fact as an exercise.)

Now suppose that each functional dependency rule is of the form $X_i \rightarrow y_i$, where y_i is a single attribute. We construct a graph $G = (V, E)$ as follows. The vertex set V consists of the symbols in T plus one additional symbol c_{ip} for each pair consisting of a row i of T and a dependency rule $X_p \rightarrow y_p$ and one additional symbol f_p for each dependency rule $X_p \rightarrow y_p$. For each vertex c_{ip}, the edge set E contains the edges (c_{ip}, f_p), $(c_{ip}, T(i, j_1))$, $(c_{ip}, T(i, j_2))$, $\ldots, (c_{ip}, T(i, j_r))$, where $X_p = \{j_1, j_2, \ldots, j_r\}$ with $j_1 < j_2 < \cdots < j$. We define an equivalence relation C on V by declaring c_{ip} and $T(i, y_p)$ equivalent for $1 \leq i \leq m$, $1 \leq p \leq l$. It is not hard to prove that if C^* is the congruence closure of C on G, then D^* is the restriction of C^* to the symbols in T.

Thus we can employ the fast algorithm of Section 2 to compute dependency closures. The graph G contains at most $m(h + 2l)$ vertices and at most $m(t + 2l)$ edges, where t is the total length of all the functional dependency rules. (These estimates take into account the blow-up caused by transforming the dependency rules so that all the right-hand sides have a single attribute.) Thus G is of size $O(n^2)$, where n is the total length of the relation schemes and dependency rules. The fast algorithm of Section 2 requires $O(n^2(\log n)^2/\log k)$ time and $O(kn^2)$ space to compute the dependency closure, where k is an arbitrary parameter. Note that these bounds are likely to be very pessimistic in practice.

Appendix

This appendix contains an Algol-like description of the fast acyclic congruence closure algorithm discussed in Section 3. The purpose of this description is to divide the algorithm into elementary steps, each of which can be executed in constant time on a random access or pointer machine, and to provide enough details so that implementing the algorithm is a routine exercise. We omit all variable declarations except those of procedure parameters.

The main procedure is as follows:

```
procedure acyclic congruence closure (set V, E, C);
    begin
        initialize;
        for $v \in V$ do if level($v$) = 0 then search($v$, true) fi od
    end acyclic congruence closure;
```

C represents the initial equivalence relation as a set of equivalence classes.

```
procedure initialize;
    begin
        stack := U := $\varnothing$;
        current level := 0;
        current class := $n + 1$;
        partition := $\{V\}$;
        for $i := 1$ until $n$ do $r(i) := \varnothing$ od;
        for $v \in V$ do level($v$) := 0; $d(v) := \{w \mid (v, w) \in E\}$ od;
        for $e \in C$ do combine($e$) od
    end initialize;
```

Variable U is a set of undirected edges defining the current equivalence classes. The other variables occurring in *initialize* will be discussed as they are used. Procedure *combine* takes as input a set of vertices and adds enough edges to U to connect all vertices in the set.

```
procedure combine (set $z$);
    begin
        let $x \in z$;
        for $y \in z - \{x\}$ do add $\{x, y\}$ to $U$ od
    end combine;
```

Procedure *search* carries out the depth-first search that finds equivalence classes, thereby executing step 1 of the algorithm. Array *level* serves two purposes. Before a vertex v is reached by the search, $level(v) = 0$. After v is reached but before v is fixed in an equivalence class, $level(v)$ gives the number of partially explored classes up to and including the one containing v. After v is fixed in an equivalence class, $level(v)$ gives the number of this class. The classes are numbered backward from n; a given number in the range $1 \leq i \leq n$ can at a given time be a level number or a class number but not both.

Procedure *search* is called with parameter *new level* equal to **true** if parameter v is the first vertex of a new level; otherwise *new level* is **false**. Variable *stack* contains the reached but not fixed vertices. Procedure *search* calls procedure *abort* (which we shall not specify further) when it finds a cycle.

```
procedure search (vertex $v$; logical new level);
    begin
        if new level then current level := current level + 1 fi;
        level($v$) := current level;
        push $v$ on stack;
        for $w$ such that $(v, w) \in E$ do
            if level($w$) = 0 then search($w$, true)
            else if level($w$) $\leq$ current level then abort fi fi od;
        for $w$ such that $\{v, w\} \in U$ do
            if level($w$) = 0 then search($w$, false)
            else if level($w$) $<$ current level then abort fi fi od;
        if new level then process next class fi
    end search;
```

Procedure *search* is an implementation of the heart of the Paterson–Wegman unification algorithm [15]. By reversing the direction of edges in E and modifying *process next class* appropriately, we can use *search* to find unifiers instead of congruence closures.

```
procedure process next class;
    begin
        construct next class;
        construct refiners;
        refine partition;
        group by signature;
        for $g \in$ groups do combine($g$) od
    end process next class;
```

Procedure *construct next class* carries out step 2 of the algorithm.

```
procedure construct next class;
    begin
        current class := current class − 1;
        e := ∅;
        while level (top of stack) = current level do
            add top of stack to e,
            level (top of stack) := current class,
            pop stack od,
        current level := current level − 1
    end construct next class,
```

Variable *partition* contains the unfixed vertices grouped by the part of the signature so far fixed. It consists of a singly linked list of headers, one header for each partial signature. The header heads a doubly linked list of vertices with that partial signature. Each vertex contains a pointer to its header. Thus in constant time it is possible to add a vertex to a list, delete a vertex from a list, find the header of a vertex, or insert a new header after a given one. See [16] for more discussion of the details of this representation.

Procedures *construct refiners* and *refine partition* update the partition using the newly fixed equivalence class *e*. Procedure *construct refiners* also carries out step 3 of the algorithm.

```
procedure construct refiners,
    begin
        s := indices := ∅,
        for $(v, w) \in E$ such that $w \in e$ do
            $d(v) := d(v) - 1$;
            if $d(v) = 0$ then add $v$ to $s$ fi,
            let $i$ be such that $w$ is the $i$th successor of $v$,
            if $r(i) = \emptyset$ then add $i$ to indices fi,
            add $v$ to $r(i)$ od,
    end construct refiners;
```

Procedure *refine partition* makes use of a procedure *strip*, which, given a set *z* as input, replaces each set $p \in$ *partition* by $p - z$. The output of *strip* is the set $\{p \cap z \mid p \in$ *partition*$\}$.

```
procedure refine partition,
    for $i \in$ indices do
        partition := strip(r(i)) ∪ partition,
        r(i) := ∅ od,
    end refine partition,
```

Procedure *group by signature* carries out step 4 of the algorithm. It too uses *strip*. The last line of *process next class* carries out step 5 of the algorithm.

```
procedure group by signature,
    groups := strip(s),
```

Here is **procedure** *strip*. For further discussion concerning the workings of *strip*, see [16].

```
set procedure strip(set z),
    begin
        split list := ∅;
        for $v \in z$ do
            let $p \in$ partition be such that $v \in p$,
            if $p$ is not split then
                mark $p$ split,
                add $p$ to *split list*,
                $p' := \varnothing$ **fi**,
            delete $v$ from $p$,
            add $v$ to $p'$ **od**,
        *strip* $= \varnothing$;
        **for** $p \in$ *split list* **do**
            **if** $p = \varnothing$ **then** delete $p$ from *partition*
                    **else** mark $p$ unsplit **fi**,
            add $p'$ to *strip* **od**
    **end** *strip*,
```

This completes our presentation of the fast acyclic congruence closure algorithm. All the elementary steps in the algorithm require constant time. The total time number of such steps is $O(m)$. Thus the algorithm runs in $O(m)$ time. The space required is also $O(m)$.

ACKNOWLEDGMENTS. We wish to thank Al Aho, John Bruno, David Johnson, Don Johnson, Scott Johnson, Greg Nelson, Derek Oppen, and Jeff Ullman for helpful discussions. We also thank Peter Honeyman for pointing out an error in our original formulation of the lossless join algorithm in Section 6.

REFERENCES

1 AHO, A V, BEERI, C, AND ULLMAN, J.D The theory of joins in relational databases. Proc 18th Ann. IEEE Symp on Foundations of Computer Science, Providence, R I, 1977, pp 107–113.

2 AHO, A V, HOPCROFT, J.E, AND ULLMAN, J.D *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass, 1974

3 CARDOZA, E, LIPTON, R., AND MEYER, A R Exponential space complete problems for Petri nets and commutative semigroups Proc 8th Ann ACM Symp. on Theory of Computing, 1976, Hershey, Pa., pp. 50–54

4. COCKE, J, AND SCHWARTZ, J T. *Programming Languages and Their Compilers. Preliminary Notes*, Second Revised Version Courant Institute of Mathematical Sciences, New York, 1970

5 DOWNEY, P J., SAMET, H, AND SETHI, R Off-line and on-line algorithms for deducing equalities Conf Record 5th Ann ACM Symp on Principles of Programming Languages, Tucson, Ariz., 1978, pp. 158–170

6 DOWNEY, P J, AND SETHI, R Assignment commands with array references *J ACM 25*, 4 (Oct. 1978), 652–666

7 GAREY, M R, AND JOHNSON, D S. *Computers and Intractibility· A Guide to the Theory of NP-completeness* Freeman, San Francisco, 1979

8 HOPCROFT, J E, AND KARP, R.M An algorithm for testing the equivalence of finite automata. Tech. Rep. 71-114, Computer Science Dept, Cornell Univ, Ithaca, N.Y, 1971

9 KNUTH, D E *The Art of Computer Programming, Vol. 1. Fundamental Algorithms.* Addison-Wesley, Reading, Mass, 1968

10 KNUTH, D E *The Art of Computer Programming, Vol. 3 Sorting and Searching* Addison-Wesley, Reading, Mass, 1973

11 KOZEN, D Complexity of finitely presented algebras. Proc. 9th Annual ACM Symp. on Theory of Computing, Boulder, Colo, 1977, pp 164–177

12 LIU, L, AND DEMERS, A An efficient algorithm for testing losslessness of joins in relational databases Tech Rep 78-351, Computer Science Dept, Cornell Univ, Ithaca, N Y, 1978

13 NELSON, G, AND OPPEN, D C Fast decision procedures based on congruence closure. *J. ACM 27*, 2 (April 1980), 356–364

14 OPPEN, D C Reasoning about recursively defined data structures *J ACM 27*, 3 (July 1980), 403–411

15 PATERSON, M S, AND WEGMAN, M N Linear unification *J Comp. Syst Sci. 16* (1978), 158–167.

16. ROSE, D J, TARJAN, R E, AND LUEKER, G S Algorithmic aspects of vertex elimination on graphs. *SIAM J Comput 5* (1976), 266–283

17 SETHI, R Scheduling graphs on two processors *SIAM J Comput 5* (1976), 73–82

18 SHOSTAK, R E An algorithm for reasoning about equality *Commun ACM 21*, 7 (July 1978), 583–585.

19 TARJAN, R E, AND YAO, A C Storing a sparse table *Commun ACM 22*, 11 (Nov. 1979), 606–611

20 TARJAN, R E Efficiency of a good but not linear set union algorithm *J. ACM 22*, 2 (April 1975), 215–225

21 TARSKI, A, MOSTOWSKI, A AND ROBINSON, R M *Undecidable Theories* North-Holland, Amsterdam, 1953

RECEIVED MAY 1976, REVISED OCTOBER 1979, ACCEPTED OCTOBER 1980