# An Improved Equivalence Algorithm

Bernard A. Galler and Michael J. Fisher
*University of Michigan, Ann Arbor, Michigan*

An algorithm for assigning storage on the basis of EQUIV-
ALENCE, DIMENSION and COMMON declarations is pre-
sented. The algorithm is based on a tree structure, and has
reduced computation time by 40 percent over a previously
published algorithm by identifying all equivalence classes
with one scan of the EQUIVALENCE declarations. The method
is applicable in any problem in which it is necessary to identify
equivalence classes, given the element pairs defining the
equivalence relation.

An algorithm for the assignment of storage on the basis
of the EQUIVALENCE declaration found in such lan-
guages as Fortran and MAD was presented in [1]. The
algorithm given here, which uses a tree structure, is a
considerable improvement over the previous one, and
the two algorithms furnish a clear-cut example of the
benefits which can be realized through the use of such
methods. (Comparison tests have shown that the new
method reduces the execution time of the algorithm by as
much as 40 percent.) The notation and statement of the
problem have been made as similar to that of [1] as possible
to facilitate comparison, and is reviewed here for com-
pleteness.

Figure 1 shows a general equivalence algorithm, suitable
for identification of equivalence classes in any context.
Figures 2 and 3 use this same algorithm for the address
assignment problem considered in [1], retaining additional
information ($D$, $d$, $d_0$, $R$, $H$ and $H'$) during the con-
struction of the trees to facilitate the address assignment
at the end.

The problem may then be stated as follows: In some
algebraic (or any other) languages, one may write EQUIV-
ALENCE declarations of the form:

$$\text{EQUIVALENCE } (X, Y, Z_1), (Z, W_5), (U, V) \quad (1)$$

where the entries consist of names of variables, subscripted
array names or unsubscripted array names (which are
assumed to represent the element of the array which has
subscript zero). Some of the variables or arrays which
occur here may have already been assigned to specific
locations in storage; others have not yet been assigned.
The entries are grouped by means of parentheses, the
groups being separated by commas. For example, state-
ment (1) would assign $X$, $Y$ and $Z_1$ to the same location,
then $Z(\equiv Z_0)$ and $W_5$ to another location, and $U$ and $V$
to yet another location (unless either $U$ or $V$ is made
equivalent to one of the other variables or arrays by some

---

other EQUIVALENCE declaration). We must exhibit
an algorithm which will result in a storage assignment for
each variable and array occurring in any EQUIVALENCE
statement.

Of course, the groups may be highly linked, such as in the
following statement.

$$\text{EQUIVALENCE } (X, Y_2), (Q, J, K), (Y_3, Z_1),$$
$$(U, V), (Y, Q), (U_3, M_{10}, N) \quad (2)$$

We shall use this example to illustrate the algorithm
presented here. Assume that $K$ has been assigned to loca-
tion 100 by some other declaration and that the dimensions
of $Y$, $Z$, $M$ and $U$ are 10, 4, 12 and 5, respectively. (In
other words, since the zero subscript is allowed here,
the highest subscripts occurring for $Y$, $Z$, $M$ and $U$ are
9, 3, 11 and 4, respectively.) There is no loss in generality
if we assume (and we do) that every other variable is
also an array of dimension 1.

The algorithm has as input a collection of $n$ groups
of subscripted array names. We shall call the groups
$G_1, \cdots, G_n$, and for the group $G_i$, we shall label the
$m_i$ array names $g_{i1}, g_{i2}, \cdots, g_{im_i}$. Associated with each
array name $g_{ij}$ will be its subscript $s(g_{ij})$ and its dimen-
sion $d(g_{ij})$. It will be convenient to use five auxiliary
vectors, called the $E$, $R$, $S$, $H$, and $H'$ vectors, respectively.
These vectors must be large enough to hold all distinct
array names appearing in the EQUIVALENCE state-
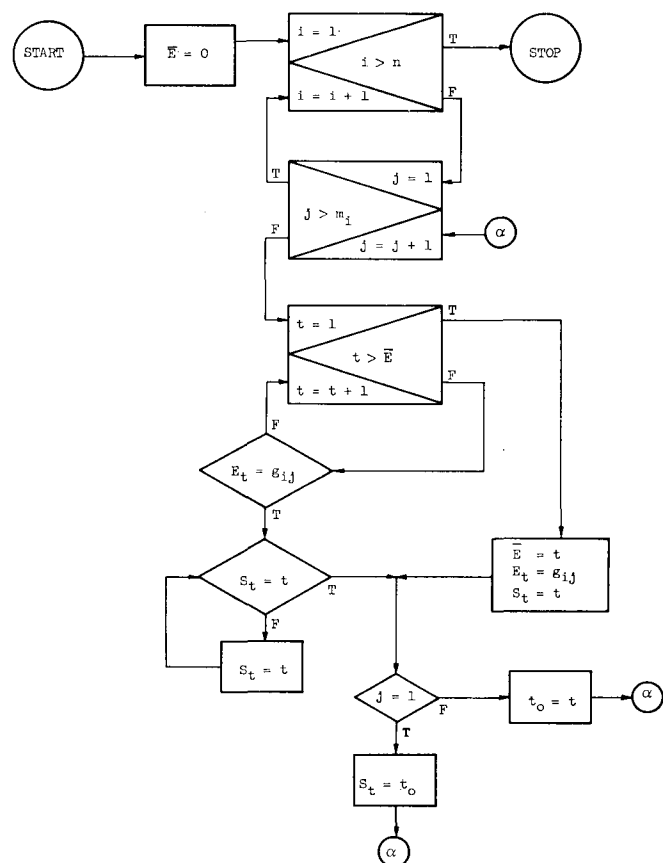ments. The number of entries in the E-vector will be



Fig. 1

denoted by $\bar{E}$, and the address assigned to an array B will be denoted a(B). We will assume that for any array B not yet assigned an address, a(B) = 0. The first available address will be denoted $a_0$, and arrays will be assumed to be stored "backwards," as in FORTRAN II and MAD. A similar algorithm would work if arrays were stored "forward."

For the example (2) above, we shall let $a_0$ = 2000. In this example, we have $G_1$ = {X, Y}, $G_2$ = {Q, J, K}, $G_3$ = {Y, Z}, $G_4$ = {U, V}, $G_5$ = {Y, Q}, $G_6$ = {U, M, N}, with $s(g_{11})$ = 0, $s(g_{12})$ = 2, $\cdots$, $s(g_{62})$ = 10, $s(g_{63})$ = 0. Also, $d(g_{11})$ = 1, $d(g_{12})$ = 10, $\cdots$, $d(g_{62})$ = 12, $d(g_{63})$ = 1. The desired address assignment would then be as follows, remembering that a(K) = 100 by some earlier declaration.

| X | 98 |
|---|---|
| Y | 100 |
| Q | 100 |
| J | 100 |
| K | 100 |
| Z | 98 |
| U | 2004 |
| V | 2004 |
| M | 2011 |
| N | 2001 |

After this assignment we should have $a_0$ = 2012 as the next available address.

Each equivalence class will be represented by a tree, each node of the tree denoting an array in the equivalence class. It will be convenient, in discussing the algorithm, to introduce the *signed distance* between successive nodes of a tree. Since the equivalence declaration might specify, for example, that A(2) and B(5) are to be equivalent, the base elements A(0) and B(0) will actually be assigned storage locations whose addresses differ by 3. We shall call the (signed) distance from one node to another the number that must be added to the address of the first one to obtain the address of the second. In this example the distance from A to B is 3, while the distance from B to A is −3, since arrays are stored backwards. It follows that one may compute the distance between any two nodes of a tree representing an equivalence class by adding algebraically the distances along any path connecting the two nodes. We shall use this fact later.

In assigning addresses to elements of an equivalence class, there are two possibilities: (1) the entire class is tied to a previous address specification for some element of the class, or (2) the class is to be assigned to the next available region in storage, and the total span of the region must be determined. In this algorithm each tree has a root which is one of the elements of the class. (Any element may be chosen to be the root, and we shall simply

choose the first one encountered.) The span for the class will be associated with the root by remembering H and H′, the distance needed above and below the root element, respectively. The sum H + H′ is thus the span for the class.

The tree structure is given by a successor vector S, in which each node is linked to the node just above it in the tree. It will be seen that most nodes are linked directly to the root, so that traversing of paths is reduced considerably.

The table representing the tree is constructed in a single pass over the groups $G_i$. Each element of each group is examined in turn. If it does not yet appear in the table, it is entered as the root of a new tree consisting of a single node. Then if the element is part of an equivalence class already identified, this new tree must be combined with the tree already under construction. If, on the other hand, it is the first element of a new group, then this step is unnecessary, since it is legitimately the root of a new tree.

If the new element is found to be in the table already, then we are either adding to a previously constructed tree (if the new element begins a group), or we must again
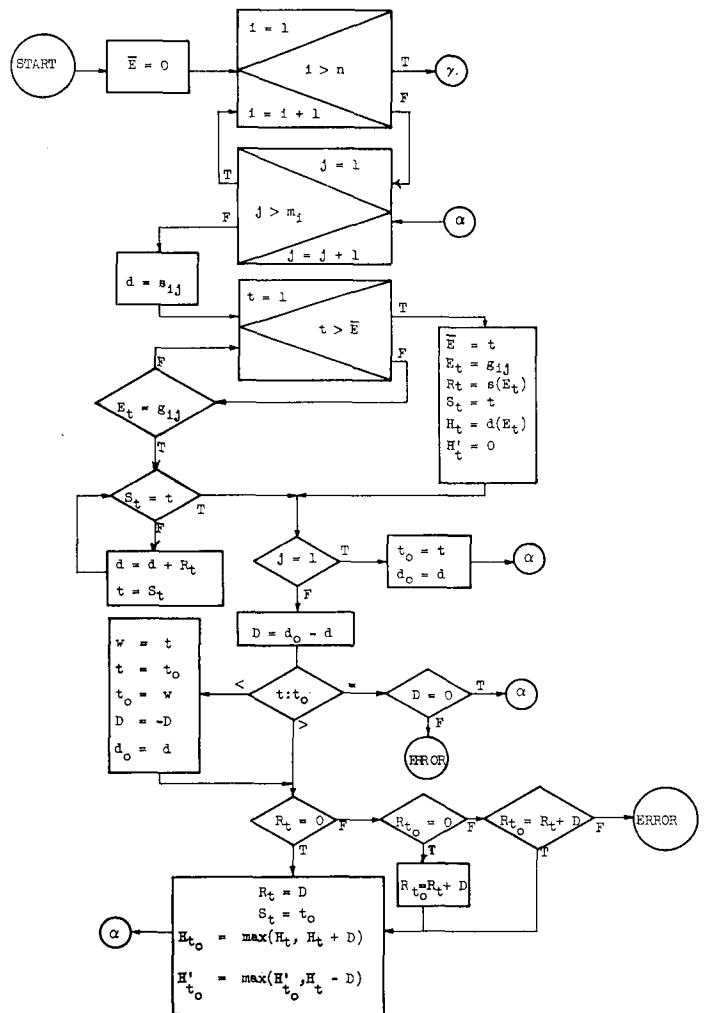


FIG. 2

combine two trees, e.g., the one currently under construction and the one indicated by the earlier appearance in the table. For example, at one stage in constructing the table for (2) above, there would be one tree $T_X$ containing X, Y and Z, with root X, another tree $T_Q$ containing Q, J and K, with root Q, and another tree $T_U$ containing U and V, with root U. Then the group (Y, Q) is encountered. Since Y is in the table, $T_X$ is the "current tree." However, when Q is found in the table, it becomes apparent that $T_X$ and $T_Q$ are to be combined.

When two trees are to be combined, either may be linked to the other. The assigning of addresses is simplified, however, if the higher-numbered root (i.e. the root appearing later in the table) is linked to the lower-numbered root, so that the root of any tree is always the first element encountered in a scan of the table. This makes the assignment of addresses possible using one pass over the table (as in Figure 3).

Assuming that a decision has been made to attach one tree to another, the distance between the root and the node to be added (the old root of the other tree) must be computed. Thus, suppose F and G are nodes in trees $T_H$ and $T_L$, with roots H and L, respectively. Suppose, also, that because F and G are declared equivalent, $T_L$ is to be attached to $T_H$. The distance from L to H may be computed as the algebraic sum of the distances from L to G, G to F (which may be zero or nonzero, depending on their subscripts in the equivalence declaration), and F to H.

In the example discussed above, where $T_X$ is to be combined with $T_Q$, we would add Q to $T_X$, and the distance from Q to X is the sum of the distances from Q to Y (zero, in this case), and Y to X.

Any address which is assigned to a tree because some element in the tree has a previous address assignment, is stored with the root. If such an address is present when two trees are linked together, the appropriate new address is computed and stored with the new root. If both old roots have previous assignments, the declared equiva-

lence is either redundant or contradictory. The former is the case only if the difference between the addresses equals the computed distance.

Another kind of redundancy or contradiction is possible, also, when two elements declared equivalent are already on the same tree. This is recognized when the two roots are the same at the time one attempts to link the trees involved. If the roots are the same, the distance is computed as usual, and a nonzero distance indicates a contradiction, while a zero distance indicates a redundancy.
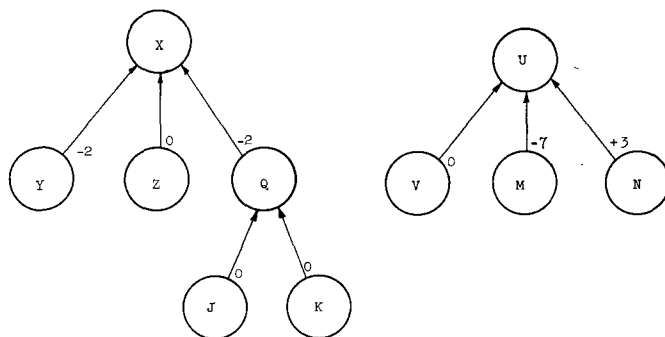


| Line | E | R | S | H | H' | A |
|---|---|---|---|---|---|---|
| 1 | X | 98 | 1 | 8 | 2 | 98 |
| 2 | Y | 100 | 1 | 10 | 0 | 100 |
| 3 | Q | 100 | 1 | 1 | 0 | 100 |
| 4 | J | 100 | 3 | 1 | 0 | 100 |
| 5 | K | 100 | 3 | 1 | 0 | 100 |
| 6 | Z | 98 | 1 | 4 | 0 | 98 |
| 7 | U | 2004 | 7 | 5 | 7 | 2004 |
| 8 | V | 2004 | 7 | 1 | 0 | 2004 |
| 9 | M | 2011 | 7 | 12 | 0 | 2011 |
| 10 | N | 2001 | 7 | 1 | 0 | 2001 |

FIG. 4

The actual assignment of addresses is accomplished in a single pass over the table. When a root is encountered, it is assigned an address if a previous specification had occurred for some element in the tree. Such a specification implies an address assignment for the root, and this in turn determines an address for each other element in the equivalence class (depending on its distance from the root). If not, the span information, which is stored with the root, allows the reservation of a block of storage of appropriate size for the entire equivalence class. Whenever a node which is not a root is encountered in the table, its address is computed by a single subtraction of the distance to the next higher node from the address assigned to that node. Since that node appears earlier in the table, its address must have been assigned already. Figure 4 shows the tree structure and the relevant table entries after addresses have been assigned on the basis of (2) above.

REFERENCE

1. ARDEN, B. W., GALLER, B. A., AND GRAHAM, R. M. An algorithm for equivalence declarations. *Comm. ACM* 4, 7 (July 1961), 310–314.
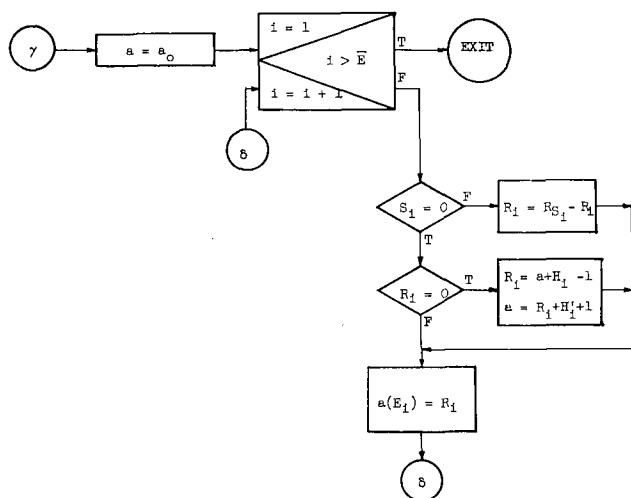
FIG. 3