ON THE EFFICIENCY OF A GOOD BUT NOT LINEAR SET UNION ALGORITHM

Robert Endre Tarjan

TR 72-148

November 1972

Computer Science Dept.
Cornell University
Ithaca, N.Y. 14850

ON THE EFFICIENCY OF A GOOD BUT NOT LINEAR SET UNION ALGORITHM

Robert Endre Tarjan
Computer Science Department
Cornell University
Ithaca, New York

Abstract:

Consider two types of instructions for manipulating disjoint sets. FIND(x) computes the name of the (unique) set containing element x. UNION(A,B,C) combines sets A and B into a new set named C. We examine a known algorithm for implementing sequences of these instructions. We show that if f(n) is the maximum time required by any sequence of n instructions,

$$k_1 n \alpha(n) \leq f(n) \leq k_2 n \log^*(n)$$

for some constants k_1 and k_2, where

$$\log^*(n) = \min\{i \mid \log^i(n) \leq 1\}$$

and $\alpha(n)$ is a recursively defined function which satisfies $\alpha(n) \to \infty$ as $n \to \infty$. Thus the set union algorithm is $O(n \log^*(n))$ but not $O(n)$.

Keywords and phrases: algorithm, complexity, equivalence, partition, set union, tree

CR Categories: 4.12, 5.25, 5.32

ON THE EFFICIENCY OF A GOOD BUT NOT LINEAR SET UNION ALGORITHM

Robert Endre Tarjan
Computer Science Department
Cornell University
Ithaca, New York

<u>Introduction</u>:

Suppose we want to use two types of instructions for manipulating disjoint sets. FIND(x) computes the name of the unique set containing element x. UNION(A,B,C) combines sets A and B into a new set named C. Initially we are given n elements, each in a singleton set. We then wish to carry out O(n) instructions of the two types. This problem arises in many contexts, including the handling of EQUIVALENCE statements in FORTRAN[1,3,4,5,6].

We may use a very simple algorithm to solve the problem. Each set is represented as a tree.† Each node in the tree represents an element in the set, and the root of the tree represents the entire set as well as some element in the set. Each tree node is represented in a computer by a cell containing two items: the element corresponding to the node, and either the name of the set

†For the purposes of this paper, a <u>tree</u> T is a directed graph with a unique vertex r, the <u>root</u> of T, such that (i) no edge (w,r) exists in T, and (ii) if $v \neq r$,,then there is a unique edge (w,v) in T. If (w,v) is an edge of T, w is called the <u>father</u> of v, and v is called a <u>son</u> of w. If there is a path from w to v in T, then w is an <u>ancestor</u> of v and v is a <u>descendant</u> of w. If vertex v has no sons, then v is a <u>leaf</u> of T. If T is a tree and v is a vertex in T, the <u>depth</u> of v is the length of the path from the root of T to v. The <u>height</u> of v is the length of the longest path from v to a leaf of T. The <u>depth</u> of the tree T is the maximum of the depths of its vertices.

(if the node is the root of the tree) or a pointer to the father of the node in the tree. Initially each singleton set is represented by a tree with one vertex.

To carry out FIND(x), we locate the cell containing x; then we follow pointers to the root of the corresponding tree to get the name of the set. In addition, we may collapse the tree:

Collapsing Rule: After a FIND, make all nodes reached during the FIND operation sons of the root of the tree.

Figure 1 illustrates a FIND operation with collapsing. Collapsing at most multiplies the time a FIND takes by a constant factor and may save time in later finds.

To carry out UNION(A,B,C), we locate the roots named A and B, make one a son of the other, and name the new root C, after deleting the old names. We may arbitrarily pick A or B as the new root, or we may apply the following rule:

Weighting Rule: If set A contains more elements than set B, make B a son of A. Otherwise make A a son of B. (In order to implement this rule, we must attach a third item to each node, namely the number of its descendants.)

Suppose we carry out O(n) FIND's and UNION's. Each UNION requires a fixed finite time. Each FIND requires time proportional to the length of the path from the node representing the element to the root of the corresponding tree. To simplify the analysis slightly, let us assume that we carry out exactly n FIND's, one on each element, and exactly $n-1$ UNION's. Let $f(n)$ be the maximum time required by any such sequence of instructions. If neither the weighting nor the collapsing rules are used, it is easy to show that:

$$(1) \qquad k_1 n^2 \leq f(n) \leq k_2 n^2$$

for suitable constants k_1 and k_2 [2]. If only the weighting rule is used, it is similarly easy to show that:

(2) $k_1 n \log n \leq f(n) \leq k_2 n \log n$

for some constants k_1 and k_2 [2]. If only the collapsing rule is used, we also have

(3) $k_1 n \log n \leq f(n) \leq k_2 n \log n$

for some constants k_1 and k_2. The first part of the inequality is due to Fisher [2]; the second part was proved recently by Mike Paterson.

If we use both the weighting rule and the collapsing rule, the algorithm becomes much harder to analyze. Hopcroft and Ullman have proved that in this case

(4) $f(n) \leq k_2 n \log^* n$

for a suitable constant k_2, where $\log^* n = \min\{i \mid \log^i n \leq 1\}$. Here we give a new proof of Hopcroft and Ullman's result. We also show that if the algorithm uses both weighting and collapsing,

(5) $k_1 n \alpha(n) \leq f(n)$

for some constant k_1, where $\alpha(n)$ is a very slowly growing, recursively defined function such that $\alpha(n) \to \infty$ as $n \to \infty$. Thus $f(n)$ is not $O(n)$.

An Upper Bound

It is useful to think about the set union algorithm in the following way: suppose we perform all n-1 UNION's first. We then have a single tree with n vertices. Each of the orginal FIND's now is a "partial" find in the new tree: to carry out

FIND(x) we follow the path in the tree from x to the closest ancestor of x corresponding to a UNION which appears before FIND(x) in the orginal sequence of operations. In this interpretation of the problem, we are interested in bounding the total length of n partial finds performed on a tree generated by n-1 set unions.

Let T be a tree containing n vertices numbered 1 through n which has been created by n-1 weighted set unions. Let d_i be the number of descendants of vertex i. Let C(T), the <u>cost</u> of tree T, be defined by:

$$(6) \quad C(T) = \sum_{i=1}^{n} c(d_i)$$

where:

$$(7) \quad c(d_i) = \frac{d_i}{(\log 4d_i)^2}$$

It is easy to show:

$$(8) \quad c(x+y) \leq c(x) + c(y) .$$

Suppose trees T_1 and T_2, with a vertices and b vertices respectively, are constructed using the weighting rule. Let $a \leq b$, and let T be the tree formed by making the root of T_1 a son of the root of T_2. Then:

$$(9) \quad C(T) = C(T_1) + C(T_2) + c(a+b) - c(b) \text{ (see Figure 2)}$$

and

$$(10) \quad C(T) \leq C(T_1) + C(T_2) + c(a) \text{ by (8)}.$$

Let $\bar{C}(n)$ be the maximum cost of a tree with n vertices constructed by applying the weighting rule. Then

$$(11)\quad \bar{C}(1) = 1/4$$

and if

$$(12)\quad C(k) \le \frac{5}{4}k - \frac{k}{\log 2k} \quad \text{for all positive integers } k<n$$

then applying (10),

$$(13)\quad \bar{C}(n) \le \max\{\bar{C}(a) + \bar{C}(b) + c(a) \mid a\le b \text{ and } a+b = n\}$$

That is,

$$(14)\quad \bar{C}(n) \le \max\{\frac{5}{4}a - \frac{a}{\log 2a} + \frac{5}{4}b - \frac{b}{\log 2b} + \frac{a}{(\log 4a)^2} \mid a\le b \text{ and } a+b = n\} ,$$

and

$$(15)\quad C(n) \le \frac{5}{4}n - \min\{a(\frac{1}{\log 2a} - \frac{1}{(\log 4a)^2}) + \frac{b}{\log 2b} \mid a\le b \text{ and } a+b = n\}$$

Since

$$(16)\quad \frac{1}{\log x-1} - \frac{1}{(\log x)^2} \ge \frac{1}{\log x} \quad \text{if } x \ge 4 ,$$

we have from (15),

$$(17)\quad \bar{C}(n) \le \frac{5}{4}n - \min\{\frac{a}{\log 4a} + \frac{b}{\log 2b} \mid a\le b \text{ and } a+b = n\}$$

and

$$(18)\quad C(n) \le \frac{5}{4}n - (\frac{a}{\log 2n} + \frac{b}{\log 2n}) \le \frac{5}{4}n - \frac{n}{\log 2n} .$$

By induction (11), (12), and (18) give:

(19) $\bar{C}(n) \le \frac{5}{4}n$ for all integers n.

Now suppose we apply a partial find of length k to a tree T. Assume without loss of generality that the find starts at vertex 1, and causes vertices 1,2,...,k-1 to become sons of vertex k. Let T' be the tree after this find is performed, and let d_i' be the number of descendants of vertex i in T'. Then:

(20) $$\begin{aligned} d_1' &= d_1 \\ d_i' &= d_i - d_{i-1} \qquad \text{for } 2 \le i \le k-1 \\ d_k' &= d_k \quad . \end{aligned}$$

Thus

(21) $$C(T) - C(T') = \sum_{i=2}^{k-1} \frac{d_i}{(\log 4d_i)^2} - \frac{d_i - d_{i-1}}{(\log 4(d_i - d_{i-1}))^2} \ .$$

Now suppose

(22) $$d_i \le 2^{\sqrt{d_{i-1}}}$$

and

(23) $$d_{i-1}^2 \le 2^{\sqrt{d_{i-1}} - 3} \ .$$

It is true that

(24) $\log(1-x) \ge -2x$ if $0 \le x \le 1/4$.

Then

$$(25)\quad \frac{d_i}{(\log 4d_i)^2} - \frac{d_i - d_{i-1}}{(\log 4(d_i - d_{i-1}))^2}$$

$$\geq \frac{2^{\sqrt{d_{i-1}}}}{(\log 4\cdot 2^{\sqrt{d_{i-1}}})^2} - \frac{2^{\sqrt{d_{i-1}}} - d_{i-1}}{(\log 4(2^{\sqrt{d_{i-1}}} - d_{i-1}))^2} \text{ by (22)}$$

$$\geq \frac{1}{2} + 2^{\sqrt{d_{i-1}}}\left(\log\left(1 - \frac{d_{i-1}}{2^{\sqrt{d_{i-1}}}}\right)\right)^2$$

$$\geq \frac{1}{2} - \frac{2^{\sqrt{d_{i-1}}}\cdot 2d_{i-1}^2}{2^{d_{i-1}}} \text{ by (23) and (24)}$$

$$\geq \frac{1}{4} \text{ by (23) .}$$

Choose a positive M such that $d_{i-1} \geq M \Rightarrow (23)$. The d_i's are strictly increasing, and $d_1 \geq 1$. Thus $i > M = d_{i-1} \geq M$. Let m be the number of pairs (d_{i-1}, d_i), $2 \leq i \leq k-1$, such that $d_{i-1} < M$ or (22) does not hold. Since $d_1 \geq 1$, $d_k \leq n$, and the d_i are strictly increasing, the number of pairs which violate (22) is no more than 3 log*n, where

$$(26)\quad \log^*(n) = \min\{i \mid \log^i(n) \leq 1\} .$$

Thus

$$(27)\quad m \leq M + 3\log^* n.$$

Now every pair (d_i, d_{i-1}) which satisfies $d_{i-1} \geq M$ and

$d_i \le 2^{\sqrt{d_{i-1}}}$ causes the tree cost to decrease by at least 1/4 when a partial find which traverses edge (d_i, d_{i-1}) is performed, by (21) and (25). But the initial tree cost is bounded by $\frac{5}{4}n$ and must remain positive after all the finds are performed. If a find is of length k, there are at least k - 2 - (M + 3 log*n) pairs (d_i, d_{i-1}) which satisfy $d_{i-1} \ge M$ and $d_i \le 2^{\sqrt{a_{i-1}}}$, by (27). Thus, if s is the sum of the lengths of all the finds,

$$(27) \quad s \le 5n + nM + 2n + 3n \log^* n \ .$$

This gives:

Lemma 1: Let f(n) be the maximum time taken by a sequence of n finds with collapsing and n-1 unions with weighting. Then for some constant k_2,

$$(28) \quad f(n) \le k_2 n \log^* n \ .$$

A Lower Bound:

To get a lower bound on f(n) , we shall show that for any fixed length L, there is some n(L) such that there are trees with n(L) or more vertices on which we may perform a partial find of length L on each of a fixed fraction p of the vertices. It follows that $p\, L\, n(L) \le f(n(L))$, for all L; and the function n(L) will give us a lower bound on the running time of the algorithm.

We need to know a nice set of trees which may be built up using weighted unions. Let T_0 be a tree with one vertex. Let T_1 be formed by successively merging two T_0 trees into another T_0 tree using the weighting rule. T_1 is a tree with three vertices, a root and two sons of the root. In general, let T_{k+1} be formed by merging 2^{k+1} T_k trees into another T_k tree. Let K_k be the tree, all of whose leaves have depth k, such that the number of sons of any vertex of height $i \geq 1$ is 2^i.

The tree T_k consists of a K_k tree plus some extra vertices. The fraction of vertices which are extra in any T_k tree is bounded by $\sum_{i=2}^{\infty} \frac{1}{2^i+1} \leq 1/2$. Thus:

<u>Lemma 2:</u> By applying weighted unions we may construct, for any k, a tree T_k, at least half of whose vertices form a K_k tree.

Now we can concentrate on K_k trees. We need a few facts about them. Let ℓ_k be the number of leaves in a K_k tree. Then $\ell_0=1$ and $\ell_{k+1} = 2^{k+1}\ell_k$. Let n_k be the number of nodes in a K_k tree. Then $n_0=1$ and $n_{k+1} = 2^{k+1}n_k+1$. Thus:

<u>Lemma 3:</u> (29) $\ell_k = 2^{k(k+1)/2}$

(30) $n_k \leq 2\ell_k - 1$

<u>Proof</u>: By induction.

Now we wish to show that for any fixed length L, there is a depth A such that we may perform a partial find of length L on every leaf of a K_A tree. In order to prove this result, we must prove something more general. Suppose we replace each vertex of height H in a K_A tree by some tree with S leaves different from the root. We show that for fixed L, S, and H, there is some A such that we may perform a find of length L on each of the added leaves. To show this we need one property of trees constructed by successive finds: once a vertex v becomes a non-descendant of vertex w, v remains a non-descendant of vertex w, since a find may never add new descendants to a vertex.

Lemma 4: For any length L, for any spread S, for any height H, there is a depth A(L,S,H) such that if $a \geq A(L,S,H)$ and we replace each vertex of height H in a K_a tree by a subtree with more than one vertex and S leaves, then we may perform a find of length L on each leaf of the added subtrees. The substituted subtrees need not be the same.

Proof: By double induction on L and S.

(31) $A(0,S,H) = A(1,S,H) = H$ if $S \geq 1$, $H \geq 0$

since every find of length 0 or length 1 does not change the position of any vertex.

(32) $A(L,1,H) = A(L-1,2^{H+1},H+1)$ if $L \geq 2$, $H \geq 0$

If we have a subtree with one leaf replacing each vertex of height H in a K_a tree, then the fathers of all the added leaves are distinct. Looking only at the paths from the fathers of the added leaves to the vertices of height H+1 in the K_a tree, we have a subtree with 2^{H+1} leaves replacing each vertex of height H+1 in the K_a tree (since vertices of height H+1 in a K_a tree have 2^{H+1} sons). If $A(L-1,2^{H+1},H+1)$ is defined then we can perform finds of length L-1 on all the fathers of added leaves if $a \geq (L-1,2^{H+1},H+1)$. But we could just as well perform finds of length L on all the added leaves themselves. Thus (2) holds, and $A(L,1,H)$ is defined if $A(L-1,s,h)$ is defined for all s and h.

$$(33)\ A(L,S,H) = A(L-1,2^{A(L,S-1,H)[A(L,S-1,H)+1]/2},\ A(L,S-1,H)) \text{ if } L \geq 2,\ S \geq 2,\ H \geq 0 .$$

To perform finds on each of the S leaves in each attached subtree, we ignore any one of the leaves in each subtree and perform finds of length L on the remaining leaves. This we can do in any K_a tree with $a \geq A(L,S-1,H)$. Consider the remaining original leaves after these finds have been performed. Their fathers have not changed. Furthermore the fathers of all these leaves are different, and each is a descendant of a vertex of height $A(L,S-1,H)$. Considering only paths from the fathers of these remaining leaves to vertices of height $A(L,S-1,H)$, we have a subtree with no more than

$2^{A(L,S-1,H)[A(L,S-1,H)+1]/2}$ leaves replacing each vertex of height $A(L,S-1,H)$. If $A(L-1,2^{A(L,S-1,H)[A(L,S-1,H)+1]/2},$ $A(L,S-1,H))$ is defined then we can perform finds of length L-1 on each of these fathers if

$$a \geq A(L-1,2^{A(L,S-1,H)[A(L,S-1,H)+1]/2},A(L,S-1,H)) .$$

But then we could just as well perform finds of length L on their sons, the remaining leaves. Thus (3) holds, and $A(L,S,H)$ is defined if $A(L-1,s,h)$ is defined for all s and h. This concludes the proof.

Corollary 5: If $f(n)$ is the worst-case running time of the equivalence algorithm,

$$(34)\quad 2^{A(L,2,1)(A(L,2,1)-1)/2}\, L \leq f\left(4\cdot 2^{A(L,2,1)[A(L,2,1)+1]/2}\right)$$

Proof: The result follows from Lemmas 2, 3, and 4, since a K_a tree with each node of height 1 replaced by a tree containing 2 leaves is just a K_a tree.

Lemma 6: Let $\alpha(n) = \max\{L \mid 4\cdot 2^{A(L,2,1)[A(L,2,1)+1]/2} \leq n\}$. Then:

$$(35)\quad k_1\, n\, \alpha(n) \leq f(n) \quad \text{for some constant } k_1 .$$

Proof: The result follows from Corollary 5 and the fact that for any n, we can calculate $L = \alpha(n)$ and construct $K_{A(L,2,1)}$ trees out of at least one fourth of the vertices, and then perform finds of length L on all their leaves.

Conclusion:

We have analyzed a known algorithm for solving the set union problem, demonstrating that if $f(n)$ is the worst-case running time required by $O(n)$ finds and unions, then

$$k_1\, n\, \alpha(n) \leq f(n) \leq k_2\, n \log^* n$$

for suitable constants k_1 and k_2, where $\alpha(n) \to \infty$ as $n \to \infty$. The function $\alpha(n)$ is defined by

$$\alpha(n) = \max\{L \mid 4\cdot 2^{A(L,2,1)[A(L,2,1)+1]/2} \leq n\},$$

where $A(L,2,1)$ grows very rapidly. In fact, it is possible to show that $A(L,2,1)$ grows faster than any primitive recursive function. Thus the bounds on $f(n)$ are not as tight as we would like. The author conjectures that by using an argument similar to the one given here to get a lower bound on $f(n)$, it is possible to show that $f(n) \leq k_2 n\,\beta(n)$, where $\beta(n)$ grows very slowly, much slower than $\log^* n$. If this is true then the set union algorithm has a worst-case running time which is non-linear but grows only a little faster than n. This is probably the first example of a simple algorithm with such a complicated running time.

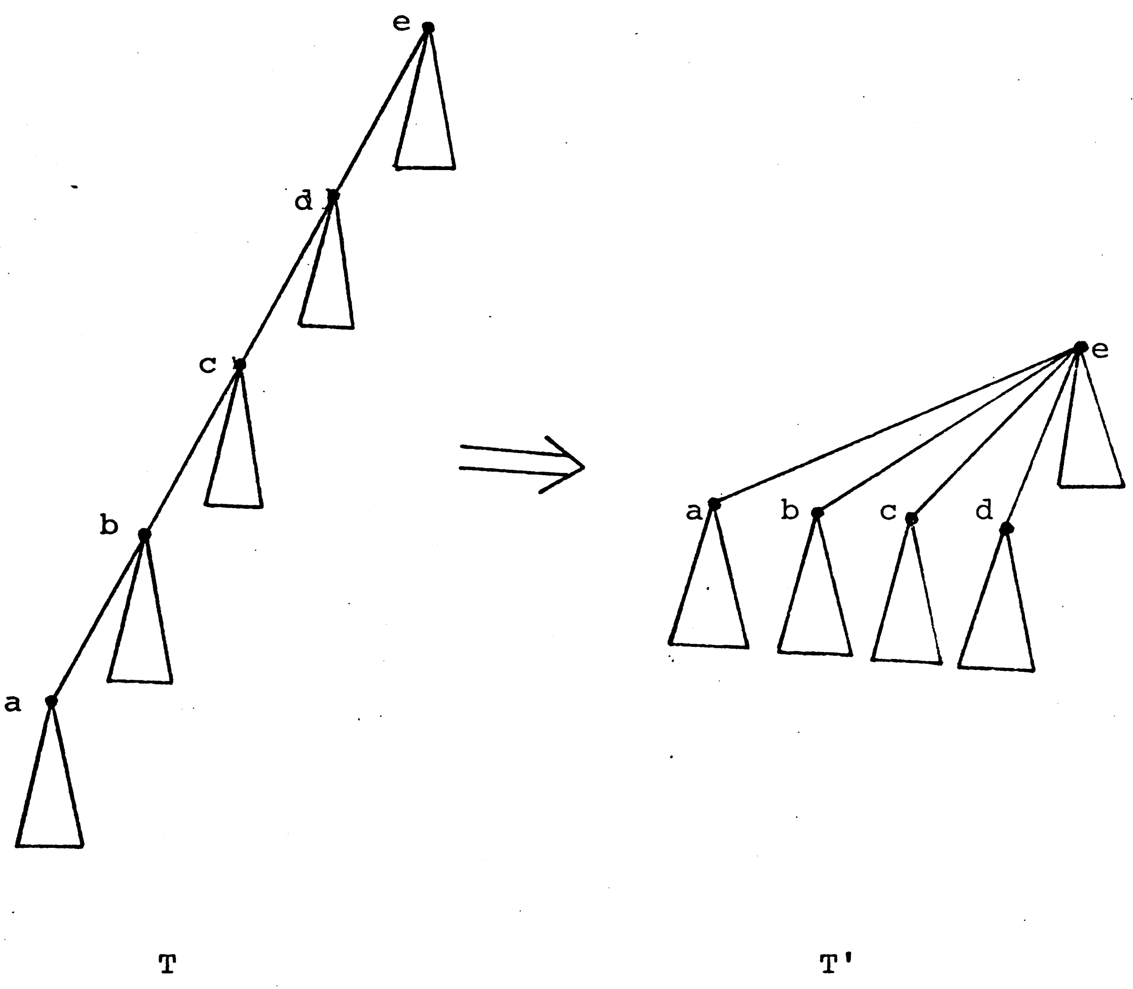

Figure 1: A FIND on element a, with collapsing. Triangles denote subtrees. Collapsing converts tree T into tree T'.

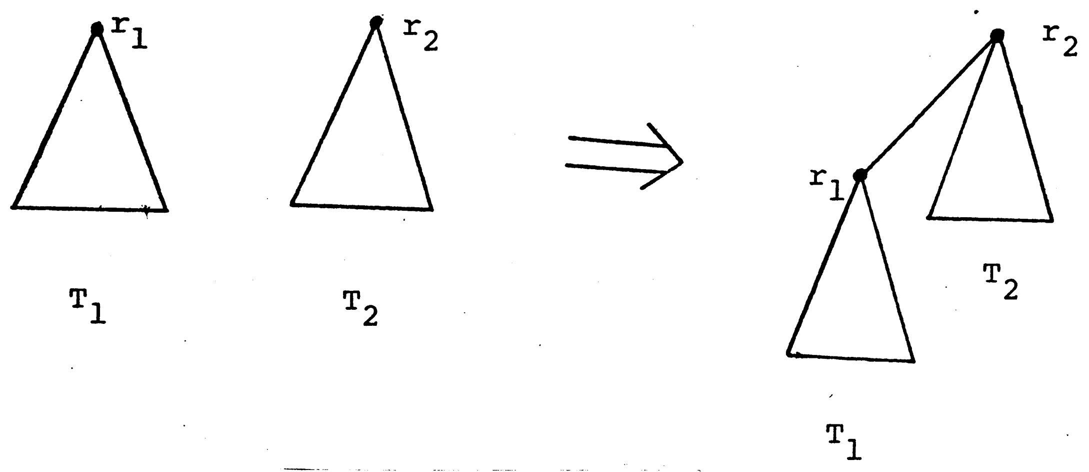

Figure 2: Merging two trees. Root r_1 of T_1 has a descendants; root r_2 of T_2 has b descendants. Root of new tree has a+b descendants.

REFERENCES

[1] B. W. Arden, B. A. Galler, and R. M. Graham, "An Algorithm for Equivalence Declarations", CACM, Vol. 4, No. 7, (July 1961), pp. 310-314.

[2] M. J. Fischer, "Efficiency of Equivalence Algorithms", in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York, 1972, pp. 153-168.

[3] B. A. Galler and M. J. Fischer, "An Improved Equivalence Algorithm", CACM, Vol. 7, No. 5, (May 1964), pp. 301-303.

[4] J. Hopcroft and J. D. Ullman, "A Linear List Merging Algorithm", Technical Report 71-111, Computer Science Department, Cornell University, Ithaca, New York, 1971.

[5] J. Hopcroft and J. D. Ullman, "Applications of a Fast List Merging Algorithm", unpublished.

[6] D. E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Massachusetts, 1969, pp. 353-355.