MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 256 April 1972

EFFICIENCY OF EQUIVALENCE ALGORITHMS

Michael J. Fischer

This paper was first presented at the Symposium on Complexity of Computer Computations, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, on March 22, 1972.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

EFFICIENCY OF EQUIVALENCE ALGORITHMS[†]

Michael J. Fischer

Massachusetts Institute of Technology
Cambridge, Massachusetts

1. INTRODUCTION

The equivalence problem is to determine the finest partition on a set that is consistent with a sequence of assertions of the form "$x \equiv y$". A strategy for doing this on a computer processes the assertions serially, maintaining always in storage a representation of the partition defined by the assertions so far encountered. To process the command "$x \equiv y$", the equivalence classes of x and y are determined. If they are the same, nothing further is done; otherwise the two classes are merged together.

Galler and Fischer (1964A) give an algorithm for solving this problem based on tree structures, and it also appears in Knuth (1968A). The items in each equivalence class are arranged in a tree, and each item except for the root contains a pointer to its father. The root contains a flag indicating that it is a root, and it may also contain other information relevant to the equivalence class as a whole.

Two operations are involved in processing a command "$x \equiv y$": first we must *find* the classes containing x and y, and then these classes are (possibly) *merged* together. The find is accomplished

[†]Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362 -0003.

by successively following the father links up the path from the given node until the root is encountered. To merge two trees together, the root of one is attached to the root of the other, and the former node is marked to indicate that it is no longer a root in the new data structure.

The time required to accomplish a find depends on the length of the path from the given node to the root of its tree, while the time to process a merge (given the roots of the two trees involved) is a constant. For definiteness, we let the cost of a merge be unity and the cost of a find be the number of nodes, including the endpoints, on the path from the given node to the root.

In this paper, we are interested in the way the cost of a sequence of instructions grows as a function of its length. Using the above algorithm, a sequence of n merge instructions can cause a tree to be built with a node v of depth n, so subsequent finds on that node will cost n+1 units each. The sequence consisting of the n merge instructions followed by n copies of a find(v) instruction will then cost n(n+2), and it is easy to see that $O(n^2)$ is an upper bound as well.

The above example suggests adding to the algorithm a "collapsing rule" which Knuth (1972A) attributes to Tritter. Every time a find instruction is executed, a second pass is made up the path from the given node to the root and each node on that path is attached directly to the root (except for the root itself). At worst this will only double the cost of the algorithm, and it may cause subsequent finds to be greatly speeded up. Indeed this turns out to be the case, for in Section 4 we show that the upper bound drops to $O(n^{3/2})$ using this heuristic.

Another heuristic, the "weighting rule", was studied by Hopcroft and Ullman (1971A) and previously known to several others. When performing a merge, an attempt is made to keep the trees balanced by always attaching the tree with the smaller number of nodes to the root of the tree with the larger number. To do this efficiently requires that extra storage be associated with each root in which to record the number of nodes in its tree. Hopcroft and Ullman show that with the weighting, a tree of n nodes can have height at most log n, and it follows that an instruction sequence of length $n>1$ can therefore have cost no greater than O(n log n). Moreover, in the absence of the collapsing rule, it is easy to construct instruction sequences whose cost does grow as n log n.

Combining the collapsing rule with the weighting rule yields an algorithm superior to those using either heuristic alone. With only the collapsing rule, we exhibit in Section 3 sequences whose cost grows proportionally to n log n, where n is the length of the sequence, and as we remarked above, a similar lower bound holds for just the weighting rule alone. Combining both heuristics, we derive in Section 4 an O(n log log n) upper bound. Hopcroft and Ullman (1971A) claim that the upper bound is actually linear. However, we have a counterexample to one of their earlier lemmas, and although this difficulty can be overcome, we are unable to follow the final part of their argument.

2. THE ALGORITHMS

An *equivalence program* over the set E is any sequence of instructions of the form find(a) where a is an element of E, or merge(A,B,C) where A, B and C are names of equivalence classes. (Cf. Hopcroft and Ullman (1971A).) Find(a) returns the name of the equivalence class of which a is a member, and merge(A,B,C) combines classes A and B into a single new class C.

We now consider two algorithms which can be used to implement equivalence programs. We first need some notation.

A *forest* F is a set of oriented (unordered) trees over some set V(F) of nodes. If v is a node, then *depth*[F](v) is the length of the path in F from v to a root, and *height*[F](v) is the maximum length of a path in F from v to a leaf. The depth and height will be written simply *depth*(v) and *height*(v) when the forest F is understood. The height of a tree A, *height*(A), is the height of its root.

The algorithms are built from three kinds of instructions which operate on a forest F. If v is a node, then *find*(v) does the following:

1. If v is a root, or if father(v) is a root, then F is left unchanged.

2. Otherwise, let $v=v_0,v_1,\ldots,v_k$ be the (unique) path from v to the root v_k. Then F is modified by making v_k the father of each of the nodes $v_0,\ldots,v_{k-2}$.

The cost of find(v) is 1 + depth(v).

The instruction *U-merge*(u,v) has unit cost and is defined only when u and v are both roots. It causes the node u to become a direct descendant of v (and hence u is no longer a root).

For any node v, let *weight*(v) be the number of nodes in the subtree rooted by v (and including v itself). The instruction *W-merge*(u,v) also has unit cost and is defined only when u and v are both roots. If weight(u) $\leq$ weight(v), it behaves exactly like U-merge(u,v); otherwise, it causes the node v to become a direct descendant of u.

We define a *U-program* to be any sequence of instructions consisting solely of finds and U-merges. Similarly, a *W-program* is any sequence of finds and W-merges.

Let α be a U- (W-)program. Then $T(\alpha)$ is the total cost of executing the instructions of α in sequence, starting from an initial forest F_0 in which every node is a root. $T(\alpha)$ is undefined if any of the instructions in α is undefined.

3. A LOWER BOUND FOR THE COST OF THE UNWEIGHTED ALGORITHM

In this section, we show how to find, for each $n > 0$, a U-program α of length n such that $T(\alpha) > cn(\log n)$ for some constant c independent of n.

We begin by defining inductively for each n a class S_n of trees:

(i) Any tree consisting of just a single node is an S_0 tree.

(ii) Let A and B be S_{n-1} trees, and assume that A and B have no nodes in common. Then the tree obtained by attaching the root of A to the root of B is an S_n tree.

Figure 3.1 illustrates the building of an S_n tree, and Figure 3.2 shows an S_4 tree.

Lemma 3.1. Let A be an S_n tree. Then A has 2^n nodes, height(A) = n, and A contains a unique node of depth n.

Proof. Trivial induction on n. □

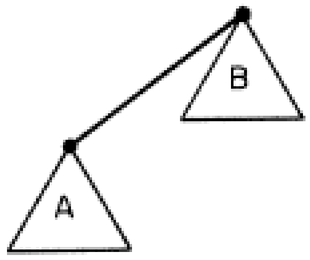

Figure 3.1. Definition of an S_n tree.

In light of the lemma, we define the *handle* of an S_n tree to be the unique node of depth n.

Two alternate characterizations of S_n trees are illustrated in Figure 3.3 and stated in:

Lemma 3.2. Let A be an S_n tree with handle v.

(a) There exist disjoint trees $A_0,\ldots,A_{n-1}$ not containing v with roots $a_0,\ldots,a_{n-1}$ respectively such that (1) A_i is an S_i tree, $0 \le i \le n-1$, and (2) A is the result of attaching v to a_0 and a_i to a_{i+1} for each i, $0 \le i < n-1$.

(b) There exist disjoint trees $A'_0,\ldots,A'_{n-1}$ with roots $a'_0,\ldots,a'_{n-1}$ respectively and a node u not in any A'_i such that (1) A'_i is an S_i tree, $0 \le i \le n-1$, and (2) A is the result of attaching a'_i to u for each i, $0 \le i \le n-1$. Moreover, v is the handle of A'_{n-1}.

Proof. Again the proof is a trivial induction on n and is omitted. □

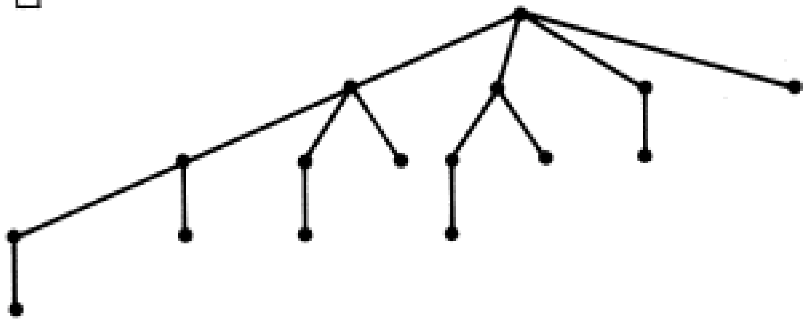

Figure 3.2. An S_4 tree.

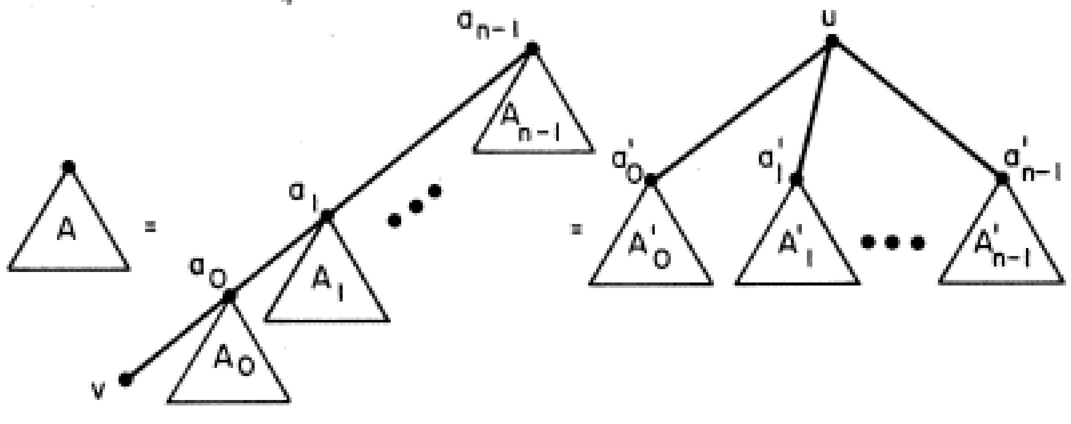

Figure 3.3. Decompositions of an S_n tree A.

The remarkable property of an S_n tree is that it is self-reproducing in the sense that if an S_n tree A is embedded in a larger tree B so that the root of A has depth > 0 in B, then a find on the handle of A (which collapses the path above the handle) costs at least n+2 and the resulting tree still has an S_n tree embedded in it!

We now make these notions more precise.

Definition. Let A and B be trees. A one-one function η: $V(A) \to V(B)$ is an *embedding* of A in B if for all $u,v \in V(A)$, $u = \text{father}(v)$ iff $\eta(u) = \text{father}(\eta(v))$. η is *initial* (*proper*) if η maps (does not map) the root of A onto the root of B. We say that A is *initially* (*properly*) *embeddable* in B if there exists an initial (proper) embedding of A in B.

Lemma 3.3. Let A be an S_n tree with handle v, and assume η is a proper embedding of A in a tree P. Then A' is initially embeddable in the tree P', where A' is an S_n tree and P' results from the instruction find(η(v)) on P.

Proof. The trees described below are illustrated in Figure 3.4.

Let A be an S_n tree with handle v, and assume η is a proper embedding of A in P. By Lemma 3.2(a), we may assume that $v, a_0, \ldots, a_{n-1}$ is the path from v to the root of A, and $a_0, \ldots, a_{n-1}$ are the roots of disjoint subtrees $A_0, \ldots, A_{n-1}$ respectively, where each A_i is an S_i tree, $0 \leq i \leq n-1$.

For each i, $0 \leq i \leq n-1$, let P_i be the subtree of P consisting of the nodes in $\{\eta(u) \mid u \in V(A_i)\}$.

Let A' be the tree formed as in Lemma 3.2(b) by linking each of the nodes a_i to a new node a'. Then A' is an S_n tree.

Let P' result from the execution of the instruction find(η(v)) on P, and let ρ be the root of P'.

Finally, define a mapping η' from the nodes of A' to the nodes of P':

$$\eta'(u) = \begin{cases} \eta(u) \text{ if } u \in V(A_i) \text{ for some } i,\ 0 \leq i \leq n-1; \\ \rho \text{ if } u = a'. \end{cases}$$

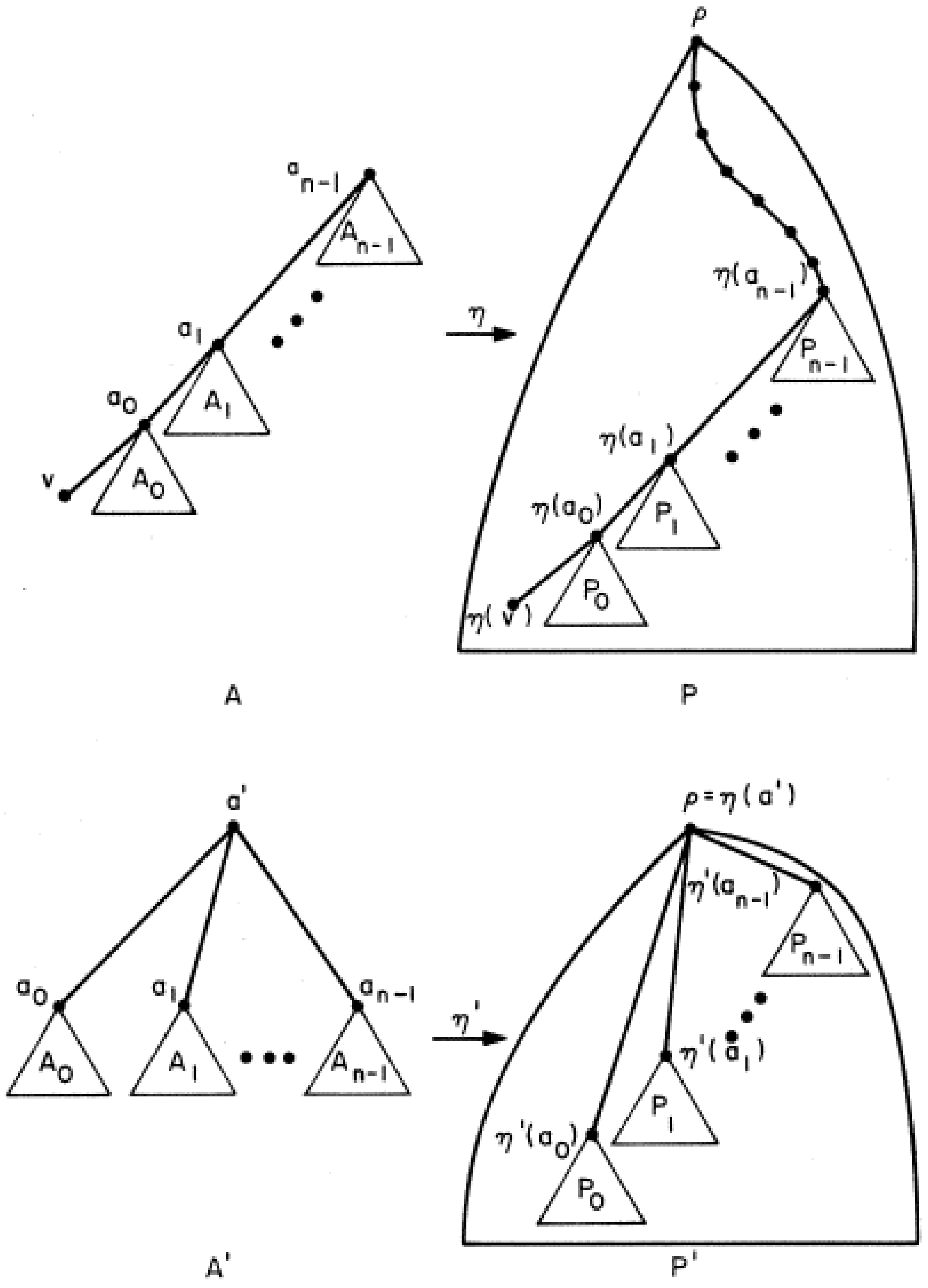

Figure 3.4. Trees in the proof of Lemma 3.3.

It remains to show that η' is an initial embedding of A' in P'

Let π be the path from $\eta(v)$ to the root of P. From the definition of embedding, each of the nodes $\eta(v)$, $\eta(a_0),\ldots,\eta(a_{n-1})$ appears on π, and no node in P_i except for $\eta(a_i)$ is in π, $0\leq i\leq n-1$.

As a consequence of the find, each of the nodes $\eta(a_i)$ is linked directly to the root ρ of P', and since the path π did not run through any nodes of P_i except for the root, P_i is a subtree of P' linked directly to ρ. It is easily verified that η' is an initial embedding of A' in P'. $\square$

We now construct a costly U-program. First build an S_k tree. Then alternately "push" it down by merging it to a new node, and perform a find on the handle. This find costs k+2 units and it leaves us with a new tree in which an S_k tree is initially embedded. Thus we can repeat the "merge, find" sequence as often as we wish, yielding an average instruction time that approaches $(k+3)/2$. Since we can do this for arbitrary k, the cost of U-programs cannot be linear in their length. In fact, we show:

Theorem 1. For any $n>0$, there exists a U-program α of length n such that $T(\alpha) > cn(\log n)$ for some constant c independent of n.

Proof. Let $a_1,a_2,\ldots$ be a sequence of distinct nodes, and let β be a program of 2^k-1 U-merges which builds an S_k tree out of the nodes $a_1,\ldots,a_{2^k}$. For each $i \geq 1$, let v_i be the handle and r_i the root of the tree that results from the sequence $\beta,\gamma_1,\ldots,\gamma_{i-1}$ and define γ_i = "U-merge($r_i,a_{2^k+i_k}$), find(v_i)". Let α be the sequence $\beta,\gamma_1,\ldots,\gamma_m$, where $m = 2^k-1$. Then $T(\alpha) = (2^k-1) + m(k+3)$, and the length of α is $n = 3m$, so

$$T(\alpha) = \frac{n}{3} + \frac{n(k+3)}{3} > cn(\log n) \qquad (3.1)$$

for some constant c.

For n not of the form $3(2^k-1)$, we form the next shorter sequence that is of that form and then extend it arbitrarily to get a sequence of length exactly n. This will have the effect only of changing the constant in (3.1). $\square$

4. UPPER BOUNDS

We get upper bounds on the two algorithms by considering a slight generalization of a find instruction. Find(u,v) behaves like a find(u) where we pretend that v is the root. More precisely, *find*(u,v) is defined only if v is an ancestor of u. If that is the case, let $u=u_0,u_1,\ldots,u_k=v$ be the path from u to v. Then find(u,v) causes each of the nodes $u_0,\ldots,u_{k-2}$ to be attached directly to v. Its cost is defined to be k+1. A sequence of generalized find and U- (W-)merge instructions is called a *generalized U- (W-)program*.

Notation. Let F be a forest and α a program. Then $F:\alpha$ is the forest that results from F by executing the instructions in α.

Lemma 4.1. Let u be any node in a forest F. Then there exists a node v in F such that F:find(u) = F:find(u,v) and the costs of executing find(u) and find(u,v) are the same.

Proof. Choose v to be the root of the tree containing u. □

Applying Lemma 4.1 in turn to each of the find instructions in a U- or W-program α gives the following:

Lemma 4.2. Let α be a U- (W-)program and F a forest. Then there exists a generalized U- (W-)program β such that $F:\alpha = F:\beta$ and $T(\alpha) = T(\beta)$.

Generalized programs are convenient to deal with because there is no loss of generality in restricting attention to programs in which all the merges precede all the finds.

Lemma 4.3. Let F be a forest containing the nodes p, q, u and v and let M be the instruction U-merge(p,q) (W-merge(p,q)). Let α_1 = "find(u,v), M" and α_2 = "M, find(u,v)". If α_1 is defined on F, then $F:\alpha_1 = F:\alpha_2$ and $T(\alpha_1) = T(\alpha_2)$.

Proof. The only possible effects of M are to change the father of p to be q, or to change the father of q to be p. Similarly, the only possible effects of the instruction find(u,v) are to change the fathers of the nodes on the path from u to v (but not including the last two such nodes). Since α_1 is defined, then v is an ancestor of u and both p and q are roots in F; hence the sets of father links changed by the two instructions are disjoint. Moreover, the choice of whether to link p to q or q to p in case M is a W-merge instruction depends only on the weights of p and q, and the weight of a root is not affected by a find instruction. Hence, neither instruction affects the action of the other, so $F:\alpha_1 = F:\alpha_2$ and $T(\alpha_1) = T(\alpha_2)$. $\square$

Lemma 4.3 enables one to convert a generalized program into an equivalent one in which all the merges precede all the finds.

Lemma 4.4. Let α be a generalized program, and let β result from α by moving all the merge instructions left in the sequence before all the finds, but preserving the order of the merges and the order of the finds. Then $F:\alpha = F:\beta$ and $T(\alpha) = T(\beta)$.

To bound the cost of a generalized U-program, we consider the effects of a U-merge and a generalized find instruction on the *total path length* of a forest F, defined to be

$$\sum_{v \in V(F)} \text{depth}(v).$$

Lemma 4.5. Let α be a sequence of n U-merge instructions and let $F = F_0:\alpha$. Then the total path length of $F \leq n^2$.

Proof. No node in F can have depth > n, and at most n nodes have non-zero depth. Hence, the total path length $\leq n^2$. $\square$

Lemma 4.6. A generalized find instruction of cost $\ell > 2$ reduces the total path length by at least $(\ell-2)^2/2$.

Proof. Let find(u,v) be an instruction of cost ℓ. Then there is a path $u=u_0,u_1,\ldots,u_{\ell-1}=v$ from u to v. For each i, $0 \leq i \leq \ell-3$, the find causes the depth of node u_i to become one plus the depth of v, so the reduction in total path length is at least

$$\sum_{i=0}^{\ell-3} (\text{depth}(u_i) - (1+\text{depth}(v))) = \sum_{i=0}^{\ell-3} (\ell-2-i) = \sum_{j=1}^{\ell-2} j \geq \frac{(\ell-2)^2}{2}. \square$$

Theorem 2. Let α be a U-program of length n. Then $T(\alpha) < cn^{3/2}$ for some constant c independent of n.

Proof. By Lemma 4.2, it suffices to bound a generalized U-program α instead, and by Lemma 4.4, we may assume that all the U-merges in α precede all the finds.

A program of length n clearly has at most n merge instructions and at most n find instructions. Let ℓ_i be the cost of the i^{th} find instruction if there is one and 0 if not. Clearly,

$$T(\alpha) \leq n + \sum_{i=1}^{n} \ell_i. \tag{4.1}$$

By Lemma 4.5, the forest after executing the merge instructions in α can have a total path length of at most n^2. Only the find instructions of cost greater than two affect the tree, so let $I = \{i \mid \ell_i > 2\}$. If $i \in I$, Lemma 4.6 asserts that the ith find instruction decreases the total path length by at least $(\ell_i - 2)^2/2$. The total path length at the end cannot be negative, so

$$n^2 \geq \frac{1}{2} \sum_{i \in I} (\ell_i - 2)^2 \geq \frac{1}{2} \sum_{i=1}^{n} (\ell_i - 2)^2 - 2n \tag{4.2}$$

or
$$6n^2 \geq \sum_{i=1}^{n} (\ell_i - 2)^2. \tag{4.3}$$

The maximum value for $\sum_{i=1}^{n} \ell_i$ is achieved when all the ℓ_i's are equal, for if they are not all the same, replacing each by the mean ℓ can only cause $\sum_{i=1}^{n} (\ell_i - 2)^2$ to decrease. Hence, from (4.1) and (4.3) we get

$$T(\alpha) \leq n + n\ell \tag{4.4}$$

where ℓ is subject to the constraint that

$$6n^2 \geq n(\ell - 2)^2. \tag{4.5}$$

From (4.5),

$$\ell \leq 2 + \sqrt{6n} \tag{4.6}$$

and substituting into (4.4), we get

$$T(\alpha) \leq n + n(2 + \sqrt{6n}) < 6n^{3/2}. \quad \square \tag{4.7}$$

For the case of the weighted algorithm, we prove an upper bound of $O(n \log \log n)$ using a method similar to our proof of Theorem 2.

We say that a forest F is *buildable* if it can be obtained from F_0 by a sequence of W-merge instructions. Buildable forests have the important property that most nodes have low height.

Lemma 4.7 (Hopcroft and Ullman (1971A)). Let F be a buildable forest. If v is a node in F of height h, then weight(v) $\geq 2^h$.

Proof. The result follows readily by induction on h. We leave the details to the reader. □

Corollary. Let α be a sequence of W-merge instructions of length n and let $F = F_0:\alpha$. For any $h \geq 0$, F contains at most $n/2^h$ non-roots of height h.

Proof. F has exactly n non-roots, for each W-merge changes one root to a non-root. Suppose $u_1,\ldots,u_k$ are non-roots of height h. By the lemma, weight$(u_i) \geq 2^h$, and all the nodes counted in the weight of u_i are non-roots, $1 \leq i \leq k$. Hence,

$$n \geq \sum_{i=1}^{k} \text{weight}(u_i) \geq k \cdot 2^h, \tag{4.8}$$

so $k \leq n/2^h$.

Instead of looking at total path length, we consider a quantity Q(F,G) which depends on two forests F and G. Our interest is in the case where F is a buildable forest and G results from F by a sequence of generalized finds, although our definition applies whenever V(F) = V(G):

$$Q(F,G) = \sum_{v \in V(F)} \text{depth}[G](v) \cdot 2^{\text{height}[F](v)}. \quad \square \tag{4.9}$$

Lemma 4.8. Let α be a sequence of W-merge instructions of length $n \geq 1$ and let $F = F_0:\alpha$. Then $Q(F,F) \leq n(\log(n+1))^2$.

Proof. No tree in F can have more than n+1 nodes. By Lemma 4.7, a root can have height at most log(n+1), so no node has height or depth greater than log(n+1).

Let $N = \{ v \in V(F) \mid \text{depth}[F](v) > 0 \}$ be the set of non-roots of F. From (4.9), we get

$$Q(F,F) \leq \log(n+1) \cdot \sum_{v \in N} 2^{\text{height}[F](v)}. \tag{4.10}$$

We now wish to bound $R(F) = \sum_{v \in N} 2^{\text{height}[F](v)}$. Since a root has height at most log(n+1), any node $v \in N$ has height at most $H = \log(n+1) - 1$, so summing over the heights of nodes,

$$R(F) = \sum_{h=0}^{\lfloor H \rfloor} (\#\text{ nodes in N of height h}) \cdot 2^h \tag{4.11}$$

By the corollary to Lemma 4.7, the number of nodes in N of height h is at most $n/2^h$, so

$$R(F) \leq \sum_{h=0}^{\lfloor H \rfloor} (\frac{n}{2^h}) \cdot 2^h \leq (H+1)n = n \cdot \log(n+1). \tag{4.12}$$

Substituting (4.12) into (4.10) gives the desired result. $\square$

Lemma 4.9. Let F be a buildable forest, ϕ a sequence of generalized finds, and let $G = F{:}\phi$. If u is a descendant of v in G and $u \neq v$, then height[F](u) < height[F](v).

Proof. It is easy to show by induction on the length of ϕ that if u is a descendant of v in G, then u is also a descendant of v in F. By the definition of height, it follows that height[F](u) < height[F](v). $\square$

Lemma 4.10. Let F be a buildable forest, ϕ a sequence of generalized finds, and let $G = F{:}\phi$. Assume find(u,v) is defined on G, has cost $\ell > 2$, and results in a forest G'. Then $Q(F,G) - Q(F,G') \geq 2^{\ell-3}$.

Proof. Let $u=u_0,\ldots,u_{\ell-1}=v$ be the path from u to v in G. By Lemma 4.9, the heights in F of the nodes in the path are monotone increasing, and since heights are integral, height[F]$(u_{\ell-3}) \geq \ell-3$. The instruction find(u,v) does not increase the depth of any node and it decreases the depth of $u_{\ell-3}$ by one, so

$Q(F,G) - Q(F,G') \geq 2^{\text{height}[F](u_{\ell-3})} \geq 2^{\ell-3}$. $\square$

Theorem 3. Let α be a W-program of length $n \geq 4$. Then $T(\alpha) < cn(\log\log n)$ for some constant c independent of n.

Proof. By Lemmas 4.2 and 4.4, it suffices to prove the theorem for a generalized W-program $\alpha = \mu\phi$ of length n, where μ is a sequence of W-merge instructions and ϕ is a sequence of generalized find instructions.

The lengths of μ and ϕ are clearly both at most n. Let ℓ_i be the cost of the ith find instruction if there is one and 0 if not. Then

$$T(\alpha) \leq n + \sum_{i=1}^{n} \ell_i. \tag{4.13}$$

Now, let $F = F_0 : \mu$. By Lemma 4.8,

$$Q(F,F) \leq n(\log(n+1))^2. \tag{4.14}$$

Only find instructions of cost greater than two affect the forest, so let $I = \{i \mid \ell_i > 2\}$ and let $G = F:\phi$. By repeated use of Lemma 4.10,

$$Q(F,F) - Q(F,G) \geq \sum_{i \in I} 2^{(\ell_i - 3)} \geq \left(\sum_{i=1}^{n} 2^{(\ell_i-3)}\right) - n. \tag{4.15}$$

Since $Q(F,G) \geq 0$, we conclude from (4.14) and (4.15) that

$$n(\log(n+1))^2 \geq \sum_{i=1}^{n} 2^{(\ell_i-3)} - n, \tag{4.16}$$

so $$2n(\log(n+1))^2 \geq \sum_{i=1}^{n} 2^{(\ell_i-3)}. \tag{4.17}$$

The maximum value for $\sum_{i=1}^{n} \ell_i$ is achieved when all the ℓ_i's are equal, for if they are not all the same, replacing each by the mean ℓ can only cause $\sum_{i=1}^{n} 2^{(\ell_i-3)}$ to decrease. Hence, from (4.13) and (4.17), we get

$$T(\alpha) \leq n + n\ell \tag{4.18}$$

where ℓ is subject to the constraint that

$$2n(\log(n+1))^2 \geq n \cdot 2^{(\ell-3)}. \tag{4.19}$$

Taking logarithms (to the base 2), we get

$$\ell \leq 3 + \log 2 + 2(\log\log(n+1)) \leq 6(\log\log(n+1)). \tag{4.20}$$

Substituting back into (4.18) yields

$$T(\alpha) \leq n + 6n(\log\log(n+1)) < 13n(\log\log n). \tag{4.21}$$

5. CONCLUSION

We have considered two heuristics, the collapsing rule and the weighting rule, which purportedly improve the basic tree-based equivalence algorithm. Our results, together with the remarks in the introduction, show that each heuristic does indeed improve the worst case behavior of the algorithm, and together they are better than either alone.

There is still a considerable gap between the lower and upper bounds we have been able to prove for the two algorithms employing the collapsing rule, and we are unable to show even that the weighted algorithm requires more than linear time. We leave as an open problem to construct any equivalence algorithm at all which can be proved to operate in linear time.

ACKNOWLEDGEMENT

The author wishes to express his appreciation to Albert Meyer for several enlightening discussions and to Patrick O'Neil for some ideas leading to a proof of Theorem 3. He is also grateful to John Hopcroft for pointing out an error in the original version of that proof.

<u>References</u> for "Efficiency of Equivalence Algorithms"
by Michael J. Fischer.

B.A. Galler and M.J. Fischer

(1964A) "An Improved Equivalence Algorithm," <u>Comm. ACM 7,5</u> (May 1964), 301-303.

J.E. Hopcroft and J.D. Ullman

(1971A) "A Linear List Merging Algorithm," <u>Technical Report TR 71 - 111</u>, Computer Science Department, Cornell University (November 1971).

D.E. Knuth

(1968A) <u>The Art of Computer Programming, Volume 1</u>, Addison-Wesley, Reading, Mass., 1968, 353-355.

(1972A) "Some Combinatorial Research Problems with a Computer-Science Flavor," notes by L. Guibas and D. Plaisted from an informal seminar, January 17, 1972.