SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

0097-5397/82/1101-0013 $01.00/0

A POLYNOMIAL TIME ALGORITHM FOR DECIDING THE EQUIVALENCE PROBLEM FOR 2-TAPE DETERMINISTIC FINITE STATE ACCEPTORS*

E. P. FRIEDMAN† AND S. A. GREIBACH‡

Abstract. The equivalence problem for the class of one-way deterministic 2-tape finite state acceptors is the problem of deciding "$L(M_1) = L(M_2)$", where M_1 and M_2 are machines in this class. A new algorithm for deciding equivalence is provided, having time complexity proportional to $p(n)$, where p is a polynomial and n is the size of the machines. This improves upon the best previously known upper bound having order 2^{cn^6}, where c is a constant [C. Beeri, Theoret. Comput. Sci., 3 (1976), pp. 305–320].

Key words. deterministic, equivalence problem, finite state acceptors, multitape automata, polynomial time, 2-tape acceptors

1. Introduction. Consider the class of deterministic finite state machines with t tapes, each with its own read-only head moving from left to right. The equivalence problem for machines in this class has remained open for $t \geqq 3$ since it was originally posed [6]. Rabin and Scott [6] showed that equivalence is decidable for $t = 1$, and this problem was later proven to have time complexity of order $nG(n)$, where n is the size of the machines and $G(n)$ is a function which grows extremely slowly [1], [5]. Although the equivalence problem remains decidable for nondeterministic single-tape machines [6], the problem becomes undecidable for classes of nondeterministic machines with more than one tape [4], [7]. Hereafter we restrict our attention to deterministic devices.

Equivalence for the case $t = 2$ was first shown to be decidable by Bird [3]. Later, Valiant [8], [9] provided a different algorithm for determining equivalence of 2-tape acceptors, and his algorithm was modified by Beeri [2], yielding an upper bound on the time complexity of 2^{cn^6}, where c is some constant and n is the size of the machines. In this paper we provide another algorithm for determining equivalence of 2-tape acceptors, but with the time complexity reduced from an exponential bound to a polynomial one.

In § 2, we establish notation and define 2-tape deterministic finite state acceptors formally. In § 3, we provide an exponential time algorithm for determining equivalence between two states of a single 2-tape acceptor. In § 4 we build upon the ideas in this algorithm to obtain a polynomial time algorithm for deciding equivalence. The remarks in § 5 indicate how this algorithm can be improved to have time complexity of order n^4, where n is the size of the machines. All complexity bounds are given for execution on a RAM under the uniform cost criterion (cf. [1] for discussion of this concept).

2. Preliminaries. A *2-tape deterministic finite-state acceptor* (abbreviated 2-dfsa) is denoted by $M = (K_1, K_2, \Sigma, \delta, q_0, F)$, where K_1 and K_2 are disjoint finite sets of *states*, with K_1 controlling tape 1 and K_2 controlling tape 2, Σ is a finite set of *input symbols*, q_0 in $K_1 \cup K_2$ is the *initial state*, F a subset of $K_1 \cup K_2$ is the set of *accepting* or *final states*, and $\delta : ((K_1 \cup K_2) \times \Sigma) \to (K_1 \cup K_2)$ is the *transition function*.

* Received by the editors August 16, 1979, and in final form February 5, 1981. This research was supported in part by the National Science Foundation under Grant MCS-78-04725.

† Radar Systems Group, Hughes Aircraft Company, Bldg. R1, Mail Station B218, P.O. Box 92426, Los Angeles, California 90009 and Computer Science Department, University of California, Los Angeles, California 90024. This work was completed while this author was at the Software Engineering Laboratory, Radar Systems Group, Hughes Aircraft Company, El Segundo, California.

‡ Computer Science Department, University of California, Los Angeles, California 90024.

Let e denote the *empty word*, $|x|$, the *length of word* x, and $|(u, v)| = |u| + |v|$, the *length of pair* (u, v). We say that x is a *suffix* of y if $y = zx$ for some z; it is a *proper suffix* if $x \neq y$. We let $|S|$ denote the cardinality of finite set S.

We write $\delta(p, a) = q$ as $p\text{—}(a, e)\rightarrow q$ when p is in K_1 and as $p\text{—}(e, a)\rightarrow q$ when p is in K_2, and call it a *1-step computation*. We let $p\text{—}(e, e)\rightarrow p$ trivially for any p in $K_1 \cup K_2$; for any p, q, r in $K_1 \cup K_2$ and u, v, x, y in Σ^*, if $p\text{—}(u, v)\rightarrow q$ and $q\text{—}(x, y)\rightarrow r$, we write $p\text{—}(ux, vy)\rightarrow r$, and call it a *computation*; this computation is said to have *length* $|(ux, vy)|$. A state q is *accessible from a state* p if $p\text{—}(u, v)\rightarrow q$ for some u, v in Σ^*, and *accessible* if it is accessible from the initial state q_0.

The *language accepted from state* p is

$$L(p) = \{(u, v) \mid p\text{—}(u, v)\rightarrow q \text{ for some } q \text{ in } F\}$$

and the language accepted by M is $L(M) = L(q_0)$. A pair (u, v) is accepted by M if it is in $L(M)$; otherwise, it is rejected.

The *left quotient* of $L(p)$ by pair (x, y), denoted $(x, y)\backslash L(p)$, is the set

$$\{(u, v) \mid p\text{—}(xu, yv)\rightarrow r \text{ for some } r \text{ in } F\}.$$

Two states p and q are *equivalent* if $L(p) = L(q)$. Two machines are equivalent if their initial states are equivalent. If $p \neq q$ and (u, v) is in $(L(p) - L(q)) \cup (L(q) - L(p))$, then (u, v) *distinguishes* p and q. If L and L' are subsets of $\Sigma^* \times \Sigma^*$, $L \neq L'$, and (u, v) in in $(L - L') \cup (L' - L)$, then (u, v) *distinguishes* L and L'.

The *state equivalence problem for 2-dfsa's* is the problem of deciding "$L(p) = L(q)$" where p and q are states in a 2-dfsa. The *equivalence problem for 2-dfsa's* is the problem of deciding "$L(M_1) = L(M_2)$" where M_1 and M_2 are 2-dfsa's.

3. The simulating machine. In this section we develop the main ideas to be used in our algorithm for deciding equivalence of 2-dfsa's. The construction is illustrated by first describing a less efficient mechanism for deciding whether "$L(p) = L(q)$", for p and q two states in an arbitrary 2-dfsa M. In § 4 we build upon this construction to obtain the desired polynomial time algorithm for deciding equivalence. But first, our strategy is to simulate the two computations of M by a single 2-dfsa whose finite tate control encodes the information needed to carry out this simulation. This simulator rejects all input tape pairs if and only if $L(p) = L(q)$. As expected, our method uses some of the ideas first developed by Valiant [8] and later modified by Beeri [2].

Let $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ be any 2-dfsa and p, q be any two states in $K_1 \cup K_2$. We would like to construct a simulating machine $S(M, p, q)$ to simulate at the same time two computations of M on the same input tape pair, one from state p and another from state q, and accept if and only if $L(p) \neq L(q)$. This will provide the required algorithm because emptiness is decidable for 2-dfsa [6].

Consider the operation of a "naive" simulator on input tape pair (xu, yv). If

$$p\text{—}(x, yv)\rightarrow r$$

and

$$q\text{—}(xu, y)\rightarrow s$$

with $|u| = |v|$ (both computations have read the same number of input symbols, namely $|x| + |yv| = |xu| + |y|$), then the naive simulator might record this in its finite state control as

$$[(r, u, e), (s, e, v)].$$

State $[(r, u, e), (s, e, v)]$ means that the simulated computation continuing from state r, hereafter called the LEFT computation, must simulate reading the string of stored symbols u that the other computation, called the RIGHT computation, has "read ahead" on tape 1; similarly, the RIGHT computation must simulate processing the string v on tape 2 that the LEFT computation has read ahead and stored. Therefore, this state represents testing whether $(u, e)\backslash L(r) = (e, v)\backslash L(s)$. But observe that the two computations being simulated may operate on the tapes in quite different manners. For example, the LEFT computation might search down tape 1 until it reads some designated symbol before starting to read tape 2, whereas the RIGHT one might search tape 2 first. Thus, the naive simulator discussed above will not work in general, since there is no a priori bound on the length of the strings u or v that it would need to store in state $[(r, u, e), (s, e, v)]$.

The first modification that we discuss is called the "semi-naive" simulator. This simulator keeps the two computations in synchronism on tape 1, but allows one of the computations to read ahead on tape 2 (and store the symbols read) while the other is waiting to read tape 1. For example, if

$$p\text{—}(x, yv) \to r$$

and

$$q\text{—}(x, y) \to s$$

for s in K_1, then the semi-naive simulator reads input (x, yv) and records this in its finite state control as

$$[(r, e), (s, v)].$$

State $[(r, e), (s, v)]$ means that the LEFT computation has read ahead on tape 2 and stored the string v of symbols read for the RIGHT computation to operate on later. Hence, this state represents testing whether $L(r) = (e, v)\backslash L(s)$. The LEFT computation keeps reading symbols from tape 2 and storing them until it needs to read tape 1. At that time, both the LEFT and RIGHT computations read a symbol from that tape. As before the RIGHT computation must perform actions on the stored string before processing real symbols from tape 2. We do not consider the problem of identifying when the RIGHT computation reads ahead on tape 2 instead of the LEFT one. This is not important to the discussion at hand. Even without these details, however, we can see that our semi-naive simulator is no better than the other; there is no bound on the length of v.

How do we get around this problem? Consider a computation of the semi-naive simulator that reaches a state

$$[(r, e), (s, v)]$$

for some r in K_2, s in K_1, after having previously reached a state

$$[(r, e), (t, w)]$$

for some t in K_1, and where either $t \neq s$ or $w \neq v$. The simulating machine $S(M, p, q)$ that we shall construct is able to recognize this situation. When in such a state $[(r, e), (s, v)]$, it no longer continues the simultaneous computations from states r and s, testing whether $L(r) = (e, v)\backslash L(s)$. Instead, it changes to a state of the form

$$[(t, w), (s, v)]$$

to test whether $(e, w)\backslash L(t) = (e, v)\backslash L(s)$. We call such an action a REPLACEMENT.

How can we justify this? Lemma 3.2(3) and Theorem 3.3 guarantee that such a state change is fail-safe. At the time such a change would occur, $S(M, p, q)$ is testing whether $L(r) = (e, v)\backslash L(s)$. If tape pair (x, y) distinguishes $L(r)$ and $(e, v)\backslash L(s)$, then either (x, y) distinguishes $L(r)$ and $(e, w)\backslash L(t)$, or (x, y) distinguishes $(e, v)\backslash L(s)$ and $(e, w)\backslash L(t)$. On the other hand, if $L(p) = L(q)$, then $(e, v)\backslash L(s) = L(r) = (e, w)\backslash L(t)$.

After such a REPLACEMENT has occurred, the LEFT computation must simulate reading stored string w before reading "real" input symbols on tape 2, and the RIGHT computation must simulate reading stored string v before reading "real" symbols on tape 2. Both computations keep reading real symbols from tape 1 in synchronism, as necessary. This process continues until one of the computations has read all of its stored symbols. Let us say that $S(M, p, q)$ gets into a state of the form

$$[(t', e), (s', v')]$$

for v' a suffix of v. Here, only the RIGHT computation still has stored information remaining, so now we can continue the operation as discussed before the REPLACEMENT.

But how do we handle the case where

$$[(t', w'), (s', e)]$$

occurs for w' a suffix of w, so that only the LEFT computation still has stored information? Since we are testing whether $(e, w')\backslash L(t') = (e, e)\backslash L(s')$, we can reverse the ordering of the computations and record this as

$$[(s', e), (t', w')].$$

So now, only the RIGHT computation has stored string w' it must preprocess, and the operation of the simulator can continue as before the REPLACEMENT.

There is one problem remaining with the implementation of the simulating machine $S(M, p, q)$. It must have some mechanism for remembering previously reached states in order to perform the REPLACEMENT action from state $[(r, e), (s, v)]$ to state $[(t, w), (s, v)]$. We shall see that this can be done with a finite amount of memory. We shall discuss the states as though they were composed of two segments. One segment, called CURRENT, has the form of the states we have been describing; e.g., $[(r, e), (s, v)]$ and $[(t, w), (s, v)]$. The other segment, called HISTORY, will contain the relevant past history of states visited by $S(M, p, q)$ which enables it to determine proper REPLACEMENTs when necessary. The actual structure of the HISTORY segment will be shown later.

The construction of simulating 2-dfsa $S(M, p, q)$ is sketched below. The operation follows the outline above until it recognizes that one of the computations would accept an input tape pair and the other would not accept this pair. When this occurs, $S(M, p, q)$ accepts, and this is the only situation in which $S(M, p, q)$ can accept an input tape pair. Thus, we can see that $L(p) = L(q)$ if and only if $L(S(M, p, q)) = \varnothing$.

Let $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ be a 2-dfsa. Let $K = K_1 \cup K_2$, and let p, q be any two states in K. Simulating machine $S(M, p, q)$ will have a special state ACCEPT, which is the only final state, that it enters when it knows that $L(p) \neq L(q)$. Once in state ACCEPT the machine remains there, reading symbols from tape 1.

Machine $S(M, p, q)$ also has a set of nonaccepting states having two segments [HISTORY, CURRENT]. We call these latter types of states SIMULATING states.

The CURRENT segment of a SIMULATING state is of the form

$$[(r, u), (s, v)]$$

for r, s in K, u, v in Σ^* with $0 \leqq |u|, |v| \leqq |K_2|$. We call u the LEFT CURRENT word and v the RIGHT CURRENT word.

The HISTORY segment of a SIMULATING state is an n-tuple, $n = |K_2|$, recording the relevant past history of the LEFT and RIGHT computations, which enables it to determine proper REPLACEMENTS. Each component of this n-tuple is associated with a state in K_2, say r; we call this the r component of the HISTORY segment and denote it by HISTORY(r).

The r component of HISTORY has the form either

NIL

or

(s, w)

for w in Σ^*, $0 \leqq |w| < |K_2|$, and s in K_1.

When HISTORY(r) is NIL, then simulating machine $S(M, p, q)$ has never been in a state with CURRENT segment of the form $[(r, e), (t, u)]$, for any state t in K_1 or string u. Otherwise, when the r component is (s, w), then machine $S(M, p, q)$ has previously been in a state with CURRENT segment $[(r, e), (s, w)]$. Thus, HISTORY(r) records a string that the LEFT computation in state r has read ahead of the RIGHT computation on tape 2; HISTORY also records the state in K_1 that the RIGHT computation was in at that time.

Machine $S(M, p, q)$ starts in the SIMULATING state with CURRENT segment $[(p, e), (q, e)]$, and HISTORY segment having NIL for each component. This corresponds to the initial situation where both computations are in synchronism, with the LEFT in state p, the RIGHT in state q, and no past HISTORY.

Now we shall consider the operation of $S(M, p, q)$ in a SIMULATING state. To facilitate the presentation, we classify the action of a SIMULATING state as being one of two basic types: READ and PREPARATION. When $S(M, p, q)$ performs a READ type action, it simulates a single step of the LEFT computation on an input symbol of the tape indicated; a step of the RIGHT computation is also simulated on this symbol when the state of the RIGHT computation reads the same tape as the LEFT and no stored symbols remain to be processed on that tape. Before $S(M, p, q)$ can perform another READ type action, it may make several PREPARATION type actions. A PREPARATION type action adjusts the CURRENT and HISTORY segments, performs REPLACEMENTs and other actions necessary to put the simulation in proper form for performing another READ type action. We relax our definition of a 2-dfsa to allow PREPARATION type actions of $S(M, p, q)$ not to read any symbols from the input tapes. This will not affect our results as the machine will remain deterministic. When a rule is not defined for a SIMULATING state, then it is because this state is not accessible; the transition from such a state is irrelevant.

I. PREPARATION *type actions.* Suppose $S(M, p, q)$ attempts to execute a PREPARATION type rule when it has CURRENT segment

$$[(r, u), (s, v)]$$

where $0 \leqq |u|, |v| \leqq |K_2|$. $S(M, p, q)$ examines the possibilities in the order indicated below. If the CURRENT segment does not meet the requirements of any of the actions, it performs a READ type action, as specified in II below. Otherwise, PREPARATION type actions are performed on input (e, e). After each such action,

$S(M, p, q)$ tries to execute another PREPARATION type action (except when it goes to ACCEPT after action 4).

(1) UNPACK LEFT

If r is in K_2 and $u = au'$ for some a in Σ, then the CURRENT segment is changed to

$$[(\delta(r, a), u'), (s, v)].$$

(2) UNPACK RIGHT

If s is in K_2 and $v = av'$ for some a in Σ, then the CURRENT segment is changed to

$$[(r, u), (\delta(s, a), v')].$$

(3) SWITCH

If r is in K_1 and either (a) $u \neq e = v$, or (b) s is in K_2 and $u = e = v$, then the CURRENT segment is changed to

$$[(s, e), (r, u)].$$

(4) ACCEPT

Go to ACCEPT in the following two situations where there is some w such that (w, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$:

(i) $w = e$ and (e, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$,

(ii) $w \neq e$, state r is in K_2, $u = e$, state s is in K_1, and (w, v) is in $L(s)$.

(5) HISTORY-RECORDING

If r is in K_2, s is in K_1, $u = e$, and HISTORY $(r) =$ NIL, then change this HISTORY component to (s, v).

(6) REPLACEMENT

If r is in K_2, s is in K_1, and HISTORY $(r) = (t, x) \neq (s, v)$, then the CURRENT segment is changed to

$$[(t, x), (s, v)].$$

II. READ *type actions.* Simulating machine $S(M, p, q)$ performs a READ type action when the CURRENT segment does not satisfy the requirements for any PREPARATION type action.

During a READ type action, $S(M, p, q)$ simulates a 1-step computation of both the LEFT and RIGHT computations on the same input tape whenever possible. Otherwise, it simulates a 1-step computation of just the LEFT computation. After any such action, $S(M, p, q)$ performs as many PREPARATION type actions as possible.

There are three cases.

(1) BOTH READ TAPE 1

If r and s are both in K_1, then $S(M, p, q)$ reads a symbol, say a, from tape 1 and changes the CURRENT segment to

$$[(\delta(r, a), u), (\delta(s, a), v)].$$

(2) BOTH READ TAPE 2

If r and s are both in K_2 and $u = v = e$, then $S(M, p, q)$ reads a symbol, say a, from tape 2 and changes the CURRENT segment to

$$[(\delta(r, a), e), (\delta(s, a), e)].$$

(3) READ-AHEAD

If r is in K_2, s is in K_1, and $u = e$, then $S(M, p, q)$ reads a symbol, say a, from tape 2 and changes the CURRENT segment to

$$[(\delta(r, a), e), (s, va)].$$

This concludes the construction of $S(M, p, q)$. The next theorem shows that $S(M, p, q)$ is well defined. In particular, there exists a bound on the lengths of the strings stored in the CURRENT and HISTORY segments of a SIMULATION state. Because machine $S(M, p, q)$ is constructed only to illustrate the ideas behind the algorithm of the next section, we omit the proof.

THEOREM 3.1. *Let S be an accessible* SIMULATING *state of $S(M, p, q)$ with* CURRENT *segment $[(r, u), (s, v)]$. The following statements hold.*

(1) $|u| < |K_2|$.

(2) $|v| \leqq |K_2|$.

(3) *For each t in K_2, if* HISTORY (t) *has the form (t', w), then*

$$|w| < |K_2|.$$

The next lemma gives important properties of $S(M, p, q)$ needed to establish our desired result, that $L(S(M, p, q)) = \varnothing$ if and only if $L(p) = L(q)$. Each part verifies the validity of a rule of the simulating machine. The proof is straightforward and is therefore omitted.

LEMMA 3.2.

(1) UNPACKING TAPE 1

If r is in K_1 and r—$(a, e)\rightarrow r'$ for a in Σ, then for any u, v in Σ^*,

$$(au, v)\backslash L(r) = (u, v)\backslash L(r').$$

(2) UNPACKING TAPE 2

If s is in K_2 and s—$(e, a)\rightarrow s'$ for a in Σ, then for any u, v in Σ^*,

$$(u, av)\backslash L(s) = (u, v)\backslash L(s').$$

(3) REPLACEMENT

If r is in K_2, s, t are in K_1, u, v are in Σ^*, then

$$L(r) = (e, u)\backslash L(s) \text{ and } L(r) = (e, v)\backslash L(t) \text{ if and only if}$$

$$L(r) = (e, u)\backslash L(s) \text{ and } (e, u)\backslash L(s) = (e, v)\backslash L(t).$$

(4) STRAIGHT SIMULATION

(A) TAPE 1

If r and s are in K_1, then for all u, v in Σ^*, $(e, u)\backslash L(r) = (e, v)\backslash L(s)$ if and only if

(a) *(e, e) does not distinguish $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$*

and

(b) *for all a in Σ, $(e, u)\backslash L(\delta(r, a)) = (e, v)\backslash L(\delta(s, a))$,*

(B) TAPE 2

If r and s are in K_2, then $L(r) = L(s)$ if and only if

(a) *(e, e) does not distinguish $L(r)$ and $L(s)$*

and

(b) *for all a in Σ, $L(\delta(r, a)) = L(\ (s, a))$.*

(5) READ-AHEAD SIMULATION

If r is in K_2, s is in K_1, and v is in Σ^*, then $L(r) = (e, v)\backslash L(s)$ if and only if

(A) *(e, e) does not distinguish $L(r)$ and $(e, v)\backslash L(s)$,*

(B) *for all u in Σ^+, (u, v) is not in $L(s)$*

and

(C) *for all a in Σ, $L(\delta(r, a)) = (e, va)\backslash L(s)$.*

We claim that if p and q are equivalent, then $S(M, p, q)$ never enters state ACCEPT for any input pair. Moreover, if p is not equivalent to q, then there is some input pair which takes $S(M, p, q)$ to state ACCEPT. This is captured in Theorem 3.3 below. Again, because $S(M, p, q)$ will not be used in our final algorithm, we omit the proof.

THEOREM 3.3. *Let $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ be a 2-dfsa, let p, q be any two states in $K_1 \cup K_2$, and let $S(M, p, q)$ be constructed as described earlier. Then*

$$L(p) = L(q)$$

if and only if

$$L(S(M, p, q)) = \varnothing.$$

4. The equivalence algorithm. The SIMULATING machine $S(M, p, q)$ described in the last section provides an exponential time algorithm for testing equivalence of states in a 2-dfsa. In this section we improve this bound by the following argument.

Notice first that if S and S' are two accessible SIMULATING states, with respective CURRENT segments $[(r, e), (s, v)]$ and $[(r, e), (s', v')]$, then it would be valid to use (s, v) for a REPLACEMENT for S' into a state with CURRENT segment $[(s, v), (s', v')]$; S' need not be accessible from S. Second, a careful study of $S(M, p, q)$ would show that if $L(p) \neq L(q)$, then there is a path from S_0 to ACCEPT during which all states have distinct CURRENT segments. This suggests building the accessible submachine of $S(M, p, q)$ and simultaneously testing for emptiness, i.e., whether or not ACCEPT is accessible from S_0. Only the CURRENT segments of the SIMULATING states are stored, while a uniform HISTORY (the *same* for all SIMULATING states) is constructed with space for exactly $|K_2|$ entries. (In the notation of Valiant [9] and Beeri [2], the uniform HISTORY is a REPLACEMENT function constructed along with $S(M, p, q)$.) When a new accessible state S with CURRENT segment $[(r, e), (s, v)]$ is found, the r component of HISTORY is set to (s, v) if previously NIL; if the r component of HISTORY is $(t, u) \neq (s, v)$, then a SIMULATING state with CURRENT segment $[(t, u), (s, v)]$ is constructed. An overall list of CURRENT segments of accessible states is kept along with a pushdown store of CURRENT segments of accessible but unexamined states. Essentially the algorithm performs a depth-first search of accessible states looking for ACCEPT. If ACCEPT appears on the overall list, $L(p) \neq L(q)$; if the pushdown store is emptied before ACCEPT appears, then $L(p) = L(q)$. The details of the algorithm are presented below.

To determine whether two states in a 2-dfsa are equivalent, we apply procedure EQUIVALENCE to 2-dfsa $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ and states p, q in $K_1 \cup K_2$, which are inputs to the program. Eventually EQUIVALENCE, or one of the procedures that it invokes, will HALT and print either "$L(p) = L(q)$" or "$L(p) \neq L(q)$", as appropriate. EQUIVALENCE and all other procedures in this section are written in a dialect of Pidgin ALGOL; the reader is referred to [1] for more details.

Three global data structures are central to the operation of the algorithm—LIST, STACK, and HISTORY. LIST is a list of CURRENT segments of states in machine

$S(M, p, q)$ that the algorithm has found to be accessible. Stack (i.e., pushdown store) structure STACK holds those elements of LIST whose immediate successors in $S(M, p, q)$ have not yet been examined. HISTORY is a $|K_2|$ element vector holding the relevant HISTORY components. To begin, both STACK and LIST contain the single element $[(p, e), (q, e)]$; each component of HISTORY is NIL.

EQUIVALENCE searches through the elements in STACK looking for ACCEPT. If STACK empties without finding ACCEPT to be accessible, then EQUIVALENCE halts and prints "$L(p) = L(q)$". Otherwise, EQUIVALENCE removes the elements from STACK, one by one, from the top. Suppose the topmost element is $S = [(r, u), (s, v)]$. Procedure ACCEPT(S) is invoked to determine if $S(M, p, q)$ would enter state ACCEPT (PREPARATION rule of type (4). If ACCEPT(S) returns value "YES", then EQUIVALENCE halts and prints "$L(p) \neq L(q)$". If the value returned is "NO", then the algorithm continues by placing those CURRENT segments in $S(M, p, q)$ that are immediate successors of S onto the STACK for later examination by EQUIVALENCE. This is accomplished by procedure EXAMINE(S) and the subroutines that it invokes.

First, procedure EXAMINE performs a HISTORY-RECORDING action (PREPARATION rule of type 5) when necessary. Second, if $S(M, p, q)$ could perform a READ type action when having CURRENT segment S, procedure EQUIVALENCE invokes subroutine NEXT. NEXT finds all possible CURRENT segments resulting from a single READ move (for each symbol in Σ), and stores them in both LIST and STACK if they are not already in LIST, that is they have not previously been found accessible. Otherwise, when S indicates that $S(M, p, q)$ could not perform a READ action, EXAMINE calls subroutine PREPARATION to see if either an UNPACK, SWITCH or REPLACEMENT type action would be indicated by the SIMULATING machine. The resulting CURRENT segment from such a move is placed on both STACK and LIST, if not already in LIST. The detailed definitions of procedures EQUIVALENCE, ACCEPT, NEXT, EXAMINE, and PREPARATION follow.

```
procedure EQUIVALENCE:
begin
    STACK ← [(p, e), (q, e)];
    LIST ← [(p, e), (q, e)];
    HISTORY ← NIL;
    while STACK not empty do
        begin
            S ← top element of STACK;
            pop STACK;
            if ACCEPT(S) = "YES" then
                begin
                    print "L(p) ≠ L(q)";
                    halt
                end
            else
                EXAMINE(S)
        end
    print "L(p) = L(q)";
    halt
end
```

```
procedure ACCEPT(S):
being
    comment Suppose that S has form [(r, u), (s, v)];
    if (e, e) distinguishes (e, u)\L(r) and (e, v)\L(s) then
        return ("YES")
    else
        if u = e, r is in K2, s is in K1,
            and there exists some w ≠ e such that
            (w, v) is in L(s) then
            return ("YES")
        else
            return ("NO")
end

procedure EXAMINE(S):
begin
    comment Suppose that S has form [(r, u), (s, v)];
    if r is in K2, s is in K1, u = e
        and HISTORY (r) = NIL, then
            HISTORY (r) ← (s, v);
    if r and s are in K1, or
        r and s are in K2, u = v = e, or
        r is in K2, s is in K1, u = e, HISTORY (r) = (s, v) then
            NEXT(S)
    else
            PREPARATION(S)
end

procedure NEXT(S):
begin
    comment Suppose that S has form [(r, u), (s, v)];
    for each a in Σ do
        begin
            comment BOTH READ TAPE 1;
            if r, s are in K1 then
                Sa ← [(δ(r, a), u), (δ(s, a), v)];
            comment BOTH READ TAPE 2;
            if r, s are in K2, u = v = e then
                Sa ← [(δ(r, a), e), (δ(s, a), e)];
            comment READ-AHEAD;
            if r is in K2, s is in K1, u = e, then
                Sa ← [(δ(r, a), e), (s, va)];
            comment ADD NEW ELEMENT TO STACK AND LIST;
            if Sa is not in LIST then
                begin
                    add Sa to LIST;
                    push Sa onto STACK
                end
        end
end
```

procedure PREPARATION(S):
begin
comment Suppose that S has form $[(r, u), (s, v)]$;
comment UNPACK LEFT or RIGHT;
if r is in K_2, $u = au'$, a is in Σ **then**
$S_P \leftarrow [(\delta(r, a), u'), (s, v)]$
else
if s is in K_2, $v = av'$, a is in Σ **then**
$S_P \leftarrow [(r, u), (\delta(s, a), v')]$;
comment SWITCH;
if r is in K_1, $v = e$, and either $u \neq e$ or s is in K_2 **then**
$S_P \leftarrow [(s, e), (r, u)]$;
comment REPLACEMENT;
if r is in K_2, s is in K_1, $u = e$,
and HISTORY $(r) = (s', v') \neq (s, v)$ **then**
$S_P \leftarrow [(s', v'), (s, v)]$;
if S_P is not on LIST **then**
begin
add S_P to LIST;
push S_P onto STACK;
end
end

This concludes the definition of our algorithm for determining equivalence of states in a 2-dfsa. The next several lemmas establish that our procedure is well-defined, in particular, that the lengths of the strings stored in elements of LIST, STACK, and HISTORY are bounded. We shall then show that the algorithm always halts, and gives the correct answer as to whether or not $L(p) = L(q)$ in time $O(n^{12})$, where n is the size of machine M. First we need some notation.

Let I denote the number of input symbols in M, so $I = |\Sigma|$. At time t during the execution of the algorithm we have the following values defined.

$$\text{HISTORYSIZE} = |\{r \mid \text{HISTORY}(r) \neq \text{NIL}\}|,$$
$$\text{HISTORYWORD} = \{v \mid \text{HISTORY}(r) = (s, v)\},$$
$$\text{HISTORYWORDSIZE} = \begin{cases} \text{MAX}\{|v| \mid v \text{ is in HISTORYWORD}\}, & \text{if HISTORYWORD} \neq \varnothing \\ -1, & \text{if HISTORYWORD} = \varnothing, \end{cases}$$
$$\text{SUFFIX} = \{x \mid x \text{ is a suffix of some word in HISTORYWORD}\},$$
$$\text{SUFFIXPLUS} = \{xa \mid x \text{ is in SUFFIX}, a \text{ is in } \Sigma\}.$$

HISTORYSIZE is the number of the components of the HISTORY vector that are not NIL at time t; HISTORYWORD contains all nonNIL strings stored in HISTORY and HISTORYWORDSIZE gives the maximum length of any such string. SUFFIX contains all suffixes of words stored in HISTORY, whereas words in SUFFIXPLUS have an additional input symbol concatenated on the right. We make the following observations:

$$|\text{SUFFIX}| \leqq (\text{HISTORYSIZE})(\text{HISTORYWORDSIZE} + 1),$$
$$|\text{SUFFIXPLUS}| \leqq (I)(|\text{SUFFIX}|)$$
$$\leqq (I)(\text{HISTORYSIZE})(\text{HISTORYWORDSIZE} + 1),$$
$$\text{HISTORYSIZE} \leqq |K_2|.$$

The following lemma is immediate from the comments above.

LEMMA 4.1. *Upon any call of* EXAMINE(S) *from procedure* EQUIVALENCE, *the following conditions must hold.*

(1) $S_1 \neq S$ *is in* LIST *but not in* STACK *if and only if* EXAMINE(S_1) *has been called previously.*

(2) *If* HISTORY $(r) = (s, v)$, *then* $[(r, e), (s, v)]$ *is in* LIST *but not in* STACK; *so* $[(r, e), (s, v)]$ *was previously on the top of* STACK.

(3) S *has never been on top of* STACK *before.*

We can now state the Bounding Lemma needed to establish bounds on the size of the words stored in the data structures.

LEMMA 4.2 (Bounding lemma). *Upon any call of* EXAMINE(S) *from procedure* EQUIVALENCE, *the following two conditions hold.*

(1) HISTORYWORDSIZE < HISTORYSIZE.

(2) *If* $[(r, u), (s, v)]$ *appears in* LIST, *then either* $u = v = e$ *or both* u *is in* SUFFIX *and* v *is in* SUFFIX $\cup$ SUFFIXPLUS.

Proof. We proceed by induction on the number of times that EXAMINE has been called. Both (1) and (2) hold trivially initially.

Suppose that (1) and (2) hold upon the current call of EXAMINE (S), and $S = [(r, u), (s, v)]$. It suffices to show that (1) and (2) hold after the execution of EXAMINE (S), since when the ACCEPT subroutine returns value "YES" the algorithm terminates.

Either NEXT (S) or PREPARATION (S) is executed. Suppose that NEXT (S) is executed. Then for each a in Σ, a value of S_a is determined. There are two cases. If both r and s are in K_1 or both r and s are in K_2, then S_a is of the form $[(r', u), (s', v)]$, so (2) holds whether or not S_a is added to LIST. Also, HISTORY is unaffected so (1) still holds. If r is in K_2, s is in K_1, $u = e$ and at the time NEXT (S) is executed HISTORY $(r) = (s, v)$, then $S_a = [(\delta(r, s), e), (s, va)]$. Now if HISTORY (r) had been (s, v) at the start of EXAMINE (S), then Lemma 4.1(2) insures that $[(r, e), (s, v)] = S$ would have been on the top of STACK previously, a contradiction to Lemma 4.1(3). So at the start of EXAMINE (S), HISTORY (r) was NIL and it was reset to (s, v). Thus HISTORYSIZE is increased by 1. At the start of EXAMINE (S), v is in SUFFIX $\cup$ SUFFIXPLUS $\cup \{e\}$, so $|v| \leq$ HISTORYWORDSIZE $+ 1$. Hence HISTORYWORDSIZE is either unaffected or increased by 1. So after EXAMINE (S) is executed, (1) holds and since v is now in HISTORY, va is in SUFFIXPLUS, and (2) holds also.

On the other hand, suppose PREPARATION (S) is executed. If upon call of EXAMINE (S), r is in K_2, s is in K_1, $u = e$, and HISTORY (r) = NIL, then NEXT (S) would have been executed. Hence HISTORY is unaffected by PREPARATION (S), so (1) holds afterwards. There are four cases. If an UNPACK is applied, S' is of the form $[(r', u'), (s', v')]$, where u' is a suffix of u and v' is a suffix of v, so (2) holds for S'. If SWITCH is applied, $v = e$ and $S' = [(s, e), (r, u)]$, u in SUFFIX, so again (2) holds for S'. In the remaining case (REPLACEMENT), r is in K_2, s is in K_1, $u = e$, HISTORY $(r) = (s', v') \neq (s, v)$, and $S' = [(s', v'), (s, v)]$. We still have v in SUFFIX $\cup$ SUFFIXPLUS. By definition, v' is in HISTORYWORD $\subseteq$ SUFFIX. Hence (2) holds after PREPARATION (S) in this case too. □

Now we can establish upper bounds on the lengths of the words stored in entries of LIST, STACK and HISTORY.

THEOREM 4.3. *The following conditions must hold throughout the execution of the algorithm.*

(1) *If* HISTORY $(r) = (s, w)$, *then* $|w| < |K_2|$.

(2) *If* $[(r', u), (s', v)]$ *appears in either* LIST *or* STACK, *then* $|u| < |K_2|$ and $|v| \leqq |K_2|$.

Proof. Statement (1) follows from the bounding lemma and the fact that HISTORYSIZE $\leqq |K_2|$. Then (2) follows from (1) since either $u = v = e$ or $|u| \leqq$ HISTORYWORDSIZE and $|v| \leqq$ HISTORYWORDSIZE $+ 1$. □

We shall now establish an upper bound on the time needed to execute the algorithm. Observe that before subroutine NEXT or PREPARATION can add an entry to LIST, a search must first be made through LIST to determine whether this entry was previously added. The length of LIST clearly affects the timing of the algorithm, and the next theorem provides a bound on this length.

THEOREM 4.4. *Let* $I = |\Sigma|$, $k = |K_2|$, $N = |K_1| + |K_2|$. *Then the number of entries in* LIST *is bounded by*

$$(I+1)N^2k^2(k+1)^2 \leqq (I+1)N^6.$$

Proof. By the bounding lemma (1), throughout execution of the algorithm

$$\text{HISTORYWORDSIZE} < k$$

and

$$\text{HISTORYSIZE} \leqq k,$$

so

$$|\text{SUFFIX}| \leqq k^2$$

and

$$|\text{SUFFIXPLUS}| \leqq Ik^2.$$

Any entry in LIST is of the form $[(r, u), (s, v)]$, with r, s in $K_1 \cup K_2$. By the bounding lemma, either $u = v = e$, or u is in SUFFIX and v is in SUFFIX $\cup$ SUFFIXPLUS. There are N possibilities each for r and s, k^2 possibilities for u, and $(I+1)k^2$ possibilities for v, yielding a bound of

$$N^2(I+1)k^4$$

which is bounded by

$$(I+1)N^6.$$

COROLLARY 4.5. *The time complexity of the algorithm is*

$$O(I^3N^{12})$$

or

$$O(n^{12})$$

taking machine size as $n = IN$ (= *size of the transition table*).

Proof. First consider the time needed to execute procedure ACCEPT (S), where $S = [(r, u), (s, v)]$. The test for whether pair (e, e) distinguishes $(e, u) \backslash L(r)$ and $(e, v) \backslash L(s)$ can clearly be done in time $O(|u| + |v|)$. On the other hand, the **else** clause is applicable if and only if $S = [(r, e), (s, v)]$ for some r in K_2, s in K_1, $0 \leqq |v| \leqq |K_2|$ and $(\Sigma^+ \times \{v\}) \cap L(s) = \varnothing$. This test can be completed within time $O(IN(|v|+1))$. Therefore, ACCEPT (S) has time complexity $O(1) + O(|u| + |v|) + O(IN(|v|+1))$, and since the bounding lemma shows that $|u| + |v| \leqq 2N - 1$, the time complexity is $O(IN^2)$.

Next we consider procedures PREPARATION (S) and NEXT (S). The time to execute PREPARATION quickly becomes dominated by the time required to search through LIST. Since the size of LIST is bounded by $(I+1)N^6$, the time needed to complete NEXT is $O(1)+O((I+1)N^6) \leqq O((I+1)N^6)$. Similar arguments show that each iteration of the **for** loop in NEXT requires this same amount of time, and since the **for** loop is executed I times, the time needed to complete NEXT is $O(I(I+1)N^6) \leqq O(I^2N^6)$.

Procedure EXAMINE uses some fixed amount of time to set HISTORY, when necessary, and then the time to execute either NEXT or PREPARATION. So EXAMINE has time bound $O(I^2N^6)$.

Finally we come to procedure EQUIVALENCE. Every execution of the **while** loop removes an entry from STACK, and no two executions can have the same topmost entry. Since each entry in STACK must also be in LIST, and there are at most $(I+1)N^6$ elements in LIST, the **while** loop can be executed at most $(I+1)N^6$ times. Each such execution may call both ACCEPT and EXAMINE. So the time to execute EQUIVALENCE is bounded by $O(1)+(I+1)(O(IN^2)+O(I^2N^6)) \leqq O(I^3N^{12})$. □

We wish to show that if $L(p)=L(q)$, then the algorithm never halts and prints "$L(p) \neq L(q)$". Lemma 4.6 below establishes this claim.

Let $S_0=[(p, e), (q, e)]$. Define

$$\text{EQUIV} = \{[(r, u), (s, v)] \mid (e, u)\backslash L(r) = (e, v)\backslash L(s)\}.$$

Observe that S_0 is in EQUIV if and only if $L(p)=L(q)$.

LEMMA 4.6 (Preserve equivalence). *If S_0 is in* EQUIV, *then every S which appears on* LIST *is in* EQUIV.

Proof. We proceed by induction on the numer of times the **while** loop in procedure EQUIVALENCE is executed. The lemma holds trivially initially.

Suppose that the lemma holds at the start of the current execution of the loop. It suffices to show that if S is on top of STACK, then ACCEPT (S) returns "NO" and any entry added to STACK (and hence LIST) by subroutine NEXT or PREPARATION is in EQUIV.

Let $S=[(r, u), (s, v)]$. ACCEPT (S) returns "YES" if either (e, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$ or $u=e$, r is in K_2, s is in K_1, and for some $w \neq e$, (w, v) is in $L(s)$. In the first case S is obviously not in EQUIV. In the second case, (w, e) is in $(e, v)\backslash L(s)$ but not in $(u, e)\backslash L(r)=L(r)$ (since r is in K_2 and $w \neq e$), so S is not in EQUIV. Hence if S is in EQUIV, ACCEPT (S) returns value "NO".

Case analysis shows easily that if S is in EQUIV and NEXT (S) adds S_a to STACK, then S_a is in EQUIV. Suppose PREPARATION (S) applies, and adds S' to STACK. IF an UNPACK or SWITCH applies, S' codes the same sets as S and so S' is in EQUIV.

Consider when a REPLACEMENT applies. In this case $S'=[(r, e), (s, v)]$, and HISTORY $(r)=(s', v')$. Since S is in EQUIV, $L(r)=(e, v)\backslash L(s)$. At a previous time, $S''=[(r, e), (s', v')]$ was on top of STACK. So if all members of LIST are in EQUIV, $L(r)=(e, v')\backslash L(s')$, so $(e, v)\backslash L(s)=(e, v')\backslash L(s')$. Hence S' is in EQUIV. This covers all cases. □

From Lemma 4.6 we can conclude that if p and q are equivalent, then the procedure EQUIVALENCE never halts and prints "$L(p) \neq L(q)$". But what if p and q are not equivalent? Lemma 4.7 below establishes the desired result: EQUIVALENCE eventually prints "$L(p) \neq L(q)$".

Given any element $S = [(r, u), (s, v)]$ in LIST or STACK, we say that pair (x, y) *distinguishes* S if (x, y) distinguishes sets $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$. If (x, y) distinguishes S, then S is $(|(x, y)|)$-*distinguishable*. We say that (x, y) distinguishes LIST if it distinguishes some element in LIST; in such a case, LIST is $(|(x, y)|)$-distinguishable.

Consider the **while** loop located in procedure EQUIVALENCE. We write LIST_t to denote the value of LIST just prior to the tth iteration through the loop; similarly, STACK_t denotes the value of STACK and TOP_t the value of the topmost element of STACK_t at this time. When the tth execution of the loop finds value "YES" returned from procedure ACCEPT, then we say that $\text{LIST}_{t+1} = \text{ACCEPT}$.

LEMMA 4.7 (Find accept). *If* LIST_t *is* n*-distinguishable, then the following two statements hold.*

(1) *There exists some* m *for which either*

(A) $\text{LIST}_m = \text{ACCEPT}$

or

(B) LIST_m *is* $(n-1)$*-distinguishable.*

(2) *There exists some* p *with*

$$\text{LIST}_p = \text{ACCEPT}.$$

Proof. We first show (1) by examination of procedures NEXT and PREPARATION, and then obtain (2) from (1) by induction on n.

First consider (1). We can assume that TOP_t is n-distinguishable for the following reasons. Suppose TOP_t is not n-distinguishable, and S is some n-distinguishable entry in LIST_t. If S is in LIST_t but not STACK_t, then at the start of some previous execution of the **while** loop S was on top of STACK. Otherwise, if S is in STACK_t, then during future executions of the loop entries above S in STACK are popped off unless ACCEPT returns "YES" before this can happen.

Suppose $S = \text{TOP}_t = [(r, u), (s, v)]$ and (x, y) distinguishes S, $|x| + |y| = n$. If $n = 0$, then ACCEPT (S) returns "YES", and if ACCEPT (S) returns "YES", then (A) holds. So assume that $n \geqq 1$ and that ACCEPT (S) returns "NO". First suppose that NEXT (S) is executed. There are three cases.

(i) r, s are both in K_1. Then $x = ax'$ for some a in Σ (or else (x, y) would be in neither $(e, u)\backslash L(r)$ nor $(e, v)\backslash L(s)$). Hence NEXT (S) finds value $S_a = [(\delta(r, a), u), (\delta(s, a), v)]$, and clearly (x', y) distinguishes S_a, with $|x'| + |y| = n - 1$. Either S_a is already on LIST_t or is added to it by NEXT. So S_a is on LIST_{t+1}, and (B) holds.

(ii) r, s are both in K_2, and $u = v = e$. Then $y = ay'$ for some a in Σ. Hence NEXT (S) finds $S_a = [(\delta(r, a), e), (\delta(s, a), e)]$, which is distinguished by (x', y) and so is $(n-1)$-distinguishable. Thus S_a is on LIST_{t+1} and (B) holds.

(iii) r is in K_2, s is in K_1, $u = e$, and HISTORY $(r) = (s, v)$. If we had $y = e$, $x \neq e$, then we would have $(x, e) = (x, y)$ in $(e, v)\backslash L(s)$ (since (x, e) cannot be in $L(r)$ for r in K_2) and hence ACCEPT (S) would return "YES". So $y = ay'$, for some a in Σ. Now NEXT (S) finds $S_a = [(\delta(r, a), e), (s, va)]$ which is distinguished by (x, y') and so is $(n-1)$-distinguishable. Since S_a is on LIST_{t+1}, (B) holds.

Now we consider the cases in which PREPARATIONS (S) is executed.

(iv) First suppose that a REPLACEMENT rule applies. This means that r is in K_2, s is in K_1, $u = e \neq v$, HISTORY $(r) = (s', v') \neq (s, v)$, for s' in K_1. We have entry $S_1 = [(r, e), (s', v')]$ also in LIST_t. So PREPARATION (S) finds value $S_P = [(s', v'), (s, v)]$, which it ensures is on LIST_{t+1}. There are two cases. Either (x, y) distinguishes $(e, v')\backslash L(s')$ and $(e, v)\backslash L(s)$ or (x, y) distinguishes $(e, v')\backslash L(s')$ and $L(r)$.

In the first case, (x, y) distinguishes S_P. Since S_P is in LIST_{t+1}, similar arguments to those seen earlier show that either S_P was at the top of STACK before some previous execution of the loop, or subsequent executions of the loop will make S_P the top element (unless ACCEPT returns "YES" during some execution, at which point (A) holds). So assume that $S_p = \text{TOP}_m$ for some m. If ACCEPT (S_P) returns "YES", (A) holds; otherwise NEXT (S_P) is executed and the arguments of Case (i) apply to S_P $(s, s'$ in $K_1)$.

Now suppose on the contrary that (x, y) distinguishes $(e, v')\backslash L(s')$ and $L(r)$. Since HISTORY $(r) = (s', v')$, at some previous time $S_1 = [(r, e), (s', v')]$ was on top of STACK. So $S_1 = \text{TOP}_m$ for some $m < t$. The arguments of Case (iii) apply to S_1 (r in K_2 and s' in K_1).

(v) Finally we consider the other cases in which PREPARATION (S) is executed. These cases (UNPACK or SWITCH) find a value for S_P encoding the same sets $\{(e, u)\backslash L(r), (e, v)\backslash L(s)\}$; thus (x, y) distinguishes S_P and ACCEPT (S_P) returns "NO". Arguments similar to those seen earlier show that unless ACCEPT returns "YES", there is some execution of the loop that has S_P on top of STACK. In the worst case, $|u|+|v|$ UNPACK rules, and possibly one SWITCH rule can apply. So PREPARATION rules other than REPLACEMENTS can apply at most $|u|+|v|+1$ times until one of Cases (i)–(iv) must occur and the appropriate arguments apply.

This establishes (1). Let us consider (2)—i.e., Case (A) of (1) always holds eventually. We proceed by induction on n.

If $n = 0$, then S is 0-distinguishable. Hence ACCEPT (S) returns "YES" and so (2) holds. Suppose (2) holds whenever $n' < n$, for $n \geqq 1$.

By (1), either (a) holds and thus (2), or else STACK_m is $(n-1)$-distinguishable for some m. By the induction hypothesis, (2) holds. □

Corollary 4.5 and Lemmas 4.6 and 4.7 yield the desired result.

THEOREM 4.8. *The equivalence problem for 2-dfsa's is decidable in time* $O(n^{12})$, *where n is the size of the machines involved.*

5. Remarks. The UNION-FIND algorithm has been used to improve the upper bound for determining equivalence of deterministic one tape finite state acceptors from $O(n^2)$ to $O(nG(n))$, where $G(n)$ grows very slowly [1]. We can use it in a similar fashion to improve the time complexity of our algorithm from $O(n^{12})$ to $O(n^4)$.

In a typical application of UNION-FIND, one starts with a collection of pairwise disjoint sets. An execution of UNION(A, A', B) joints the sets named A and A' and names the new set B. An execution of FIND(x) locates the name of the set to which element x currently belongs. For any constant c, another constant c' can be found (depending only on c) such that a series of up to cn UNION-FIND operations can be performed on n elements in time $c'nG(n)$. The reader is referred to [1] for details. We sketch briefly below the ideas behind the use of the UNION-FIND algorithm to improve our algorithm.

We wish to compute equivalence among m sets of the form $(e, v)\backslash L(r)$—henceforth denoted (r, v)—for v in SUFFIX $\cup$ SUFFIXPLUS, with $m \leqq (I+1)N^3$. Initially, we hypothesize that "$L(p) = L(q)$". If an entry $[(r, u), (s, v)]$ appears in our algorithm, that means that if $L(p) = L(q)$, then we must also have $(e, u)\backslash L(r) = (e, v)\backslash L(s)$. Equivalence is transitive so if we now hypothesize that "$(r, u) = (s, v)$", all the languages to which (r, u) or (s, v) is equivalent must be pairwise equivalent. The ACCEPT subroutine described above gives the conditions under which a proposed equivalence is contradictory and hence $L(p) \neq L(q)$. If all equivalences are established without finding a contradiction (ACCEPT does not return "YES"), then $L(p) = L(q)$.

The Improved Algorithm starts with each (r, v) in a set by itself. A STACK gives a list of proposed equivalences to test; initially it contains only $[(p, e), (q, e)]$ for the hypothesis "$L(p) = L(q)$". Suppose $S = [(r, u), (s, v)]$ is on the top of the STACK. It is removed and the following tasks are performed. If (e, e) distinguishes (r, u) and (s, v), then the algorithm HALTs with the answer "$L(p) \neq L(q)$". If r is in K_2, s is in K_1, $u = e$ and HISTORY (r) = NIL, then HISTORY (r) is changed to (s, v), and the algorithm determines whether $(e, v)\backslash L(s) \cap [\Sigma^+ \times \{e\}] = \varnothing$. If the answer is no, it HALTs with the answer "$L(p) \neq L(q)$". The next step in the algorithm has FIND$((r, u))$ and FIND$((s, v))$ locate the sets A and A' to which (r, u) and (s, v) belong. If $A = A'$, no further action is taken and the next element on the STACK is examined. If $A \neq A'$, UNION (A, A', A) joints A and A'. Then the appropriate NEXT or PREPARATION subroutine is executed as in the original algorithm. Either a sequence of I (for NEXT) entries or one entry (for PREPARATION) is made on the STACK. Again, when the STACK empties the algorithm HALTs with the answer "$L(p) = L(q)$".

Now the UNION operation can be performed at most $m - 1$ times, since the algorithm starts with m unit sets. Each execution of the UNION algorithm adds at most I entries to the STACK. Hence there are at most $I(m-1)$ entries on the STACK altogether and so at most $I(m-1)$ executions of the FIND operation. Hence this part of the algorithm has cost at most $O(ImG(m))$ or $O(I^2N^3G(IN))$ [1]. Each execution of UNION or FIND is associated with other subroutines. The NEXT or PREPARATION subroutines following a UNION can be assigned cost no more than $O(IN)$, for a total of $O(mIN)$ or $O(I^2N^4)$. Before each FIND operation, there is a part of the ACCEPT subroutine (does (e, e) distinguish (r, u) and (s, v)?) which can be assigned cost $O(N)$, for a total of $O(ImN)$ or $O(I^2N^4)$. Finally, in the up to $|K_2|$ steps which change HISTORY, the expensive part of the ACCEPT subroutine (does $(e, v)\backslash L(s)$ contain a word (w, e), $w \neq e$?) is performed. As we saw, this can be done in time $O(IN^2)$, for a total cost of $O(IN^3)$. The dominant term in this analysis in $O(I^2N^4)$, so altogether we get $O(n^4)$ for $n = IN$.

There is one difficulty with the algorithm outlined above. Initially we do not know which strings will be in HISTORY and thus in T = SUFFIX $\cup$ SUFFIXPLUS. Hence, we cannot predict which m sets will be merged eventually, and so we cannot foresee which m sets need initialization. However, we do know that strings in T will be bounded in length by k for $k = |K_2|$, and that for each i, $0 \leqq i \leqq k$, there will be at most MIN $(I^i, (I+1)k)$ distinct strings of size i in T. So initially we set up an array WORD (i, j) for $0 \leqq i \leqq k$, $0 \leqq j \leqq$ Min $(I^i, (I+1)k)$, with entries NIL (except for WORD $(0, 1) = e$). When a new entry in HISTORY appears, we update the appropriate entries WORD (i, j). This setup and updating can be considered to have a one-time cost of $O(I^2N^3)$ (there are up to N updates, each involves taking at most $(I+1)N$ possible members of T and then determining whether a candidate v already fills one of the up to $(I+1)N$ slots for WORD $(|v|, j)$). A language $(e, v)\backslash L(r)$ is denoted (r, i, j) where v = WORD (i, j). The updating precedes each READ-AHEAD step, so when $[(r, u), (s, v)]$ is to be added to STACK, u and v are already in the WORD array. Their proper identifiers (i.e., $i = |u|$, $i' = |v|$, j, j' with WORD $(i, j) = u$ and WORD $(i', j') = v$) can be found in time $O(IN)$. This cost can be assigned to a preceding UNION operation, and we have already assigned a cost of $O(IN)$ to the subroutines following each UNION, and so these considerations do not affect the order of the algorithm.

Thus, the time of the algorithm can be improved to $O(n^4)$ on a RAM under the uniform cost criterion.

We believe that the results of this paper strongly support the conjecture that the equivalence problem is decidable for general multitape deterministic finite state acceptors, and, perhaps, polynomially decidable.

REFERENCES

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

[2] C. BEERI, *An improvement on Valiant's decision procedure for equivalence of deterministic finite turn pushdown machines*, Theoret. Comput. Sci., 3 (1976), pp. 305–320.

[3] M. BIRD, *The equivalence problem for deterministic two-tape automata*, J. Comput. Syst. Sci., 7 (1973), pp. 218–236.

[4] P. C. FISCHER AND A. L. ROSENBERG, *Multitape one-way nonwriting automata*, J. Comput. Syst. Sci., 2 (1968), pp. 88–101.

[5] J. E. HOPCROFT AND R. M. KARP, *An algorithm for testing the equivalence of finite automata*, TR-71-114, Department of Computer Science, Cornell University, Ithaca, NY, 1971.

[6] M. O. RABIN AND D. SCOTT, *Finite automata and their decision problems*, IBM J. Res. Develop., 3 (1959), pp. 114–125.

[7] A. L. ROSENBERG, *Nonwriting extensions of finite automata*, Ph.D. thesis, Harvard University, Cambridge, MA, August, 1965.

[8] L. G. VALIANT, *The equivalence problem for deterministic finite-turn pushdown automata*, Inf. Control, 25 (1974), pp. 123–133.

[9] ———, *Decision procedures for families of deterministic pushdown automata*, Ph.D. thesis, Theory of computation – Report No. 1, Dept. of Computer Science, University of Warwick, Coventry, England, August, 1973.