A FAST EQUIVALENCE-CHECKING ALGORITHM FOR CIRCULAR LISTS

Yossi SHILOACH *
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Received 15 December 1978; revised version received 10 March 1979

Circular lists, comparisons, equivalence-checking algorithm

1. Introduction

We are given two circular lists, $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$. Being circular lists of the same size, A and B are *equivalent* ($A \equiv B$) iff there exists l_0 such that

$$a_l = b_{(l+l_0)(\text{mod } n)} \quad \text{for all } 1 \leqslant l \leqslant n. \tag{1}$$

Since we don't want to carry the 'mod n' throughout the paper, we define:

$$a_l = a_{l(\text{mod } n)}, \quad b_l = b_{l(\text{mod } n)} \quad \text{for all } l > n.$$

We assume that the elements of A and B are well-ordered so that any comparison between two elements a and b yields one of the three results $a < b$, $a > b$, or $a = b$. Furthermore, we will assume (without loss of generality) that these elements are real numbers and the order is the natural one.

In this paper we present an algorithm that uses at most $3n - 3$ comparisons in order to check equivalence. Our algorithm not only deals with a very fundamental data-processing problem but also has immediate applications to other problems. It can be used for example to improve the time bounds of the problem of checking similarity of two polygons. The present most efficient algorithms that solve this problem [1,4], employ the fast pattern-matching algorithm of Knuth, Morris, and Pratt [3] to check equivalence of two circular lists. The KMP algorithm, when applied to this problem, uses about $5n$ comparisons.

Another application of our algorithm (which, in fact, motivated this paper) deals with a fast generation of certain chemical structures without repetitions [2].

2. The algorithm

Given a circular list $A = (a_1, ..., a_n)$, we can turn it into a linear list by fixing a starting point. If, for example, we pick a_i as a starting point, A will turn into the linear list $[a_i, ..., a_n, a_1, ..., a_{i-1}]$. Henceforth we'll use brackets to denote linear lists and parentheses to denote circular lists. There are n linear lists $A_1, ..., A_n$ associated with a single circular list A. A_i, for $1 \leqslant i \leqslant n$, is obtained from A by choosing a_i as a starting point. In the same way, the linear lists $B_1, ..., B_n$ can be derived from $B = (b_1, ..., b_n)$.

The lists $A_1, ..., A_n, B_1, ..., B_n$ are well-ordered with respect to the lexicographic order, and we use the notation $A_i \preccurlyeq B_j$ to denote that A_i precedes or equals B_j in this order. $A_i \prec B_j$ if $A_i \preccurlyeq B_j$ and $A_i \neq B_j$.

Let $A_{i_0} = \min_{1 \leqslant i \leqslant n} A_i$ (i.e., $A_{i_0} \preccurlyeq A_i$, $1 \leqslant i \leqslant n$) and let $B_{j_0} = \min_{1 \leqslant j \leqslant n} B_j$. The following lemma is self-evident.

Lemma 1. *$A \equiv B$ iff $A_{i_0} = B_{j_0}$.*

The following lemma is also trivial but should be pointed out.

Lemma 2. *If $a_{i+l} = b_{j+l}$ for all $0 \leqslant l < k$ and $a_{i+k} < b_{j+k}$, then $A_{i+l} \prec B_{j+l}$ for all $0 \leqslant l \leqslant k$.*

* This research was supported by a Chaim Weizmann Postdoctoral Fellowship and by National Science Foundation Grant MCS-75-22870.

During the algorithm we store and update a (growing) set D of starting points, which is defined as follows:

$D = \{a_i : \exists j$ such that $B_j \prec A_i\} \cup \{b_k : \exists l$ such that $A_l \prec B_k\}$.

Lemma 3. *If* $D \supseteq \{a_1, ..., a_n\}$ *or* $D \supseteq \{b_1, ..., b_n\}$, *then* $A \not\equiv B$.

Proof. If $A \equiv B$, then a_{i_0} and b_{j_0} would never belong to D.

The main point of the algorithm is to accumulate the starting points that should be in D fast (one comparison per starting point in D), until one of the following occurs:

(1) $D \supseteq \{a_1, ..., a_n\}$,
(2) $D \supseteq \{b_1, ..., b_n\}$,
(3) $D = \{a_1, ..., a_{n-1}, b_1, ..., b_{n-1}\}$.

In cases (1)–(3) the algorithm returns $A \not\equiv B$.

(4) The algorithm discovers that $A \equiv B$ by n successive comparisons of a_{i+l} with b_{j+l}, $0 \leqslant l \leqslant n-1$, for some i and j.

COMPARE(A_i, B_j)

This routine is the core of the algorithm. It compares a_i with b_j, a_{i+1} with b_{j+1} and so on, until the first inequality occurs or n comparisons have been made without encountering any inequality. In the last case it returns $A \equiv B$ and stops. In the first case, if $a_{i+k} \prec b_{j+k}$ is the first inequality encountered, $(k \geqslant 0)$, then it adds $b_j, ..., b_{j+k}$ to D. If $b_{j+k} \prec a_{i+k}$ is the first inequality encountered, it adds $a_i, ..., a_{i+k}$ to D and returns (see Lemma 2).

Let $i_D = \max\{i : a_i \in D\}$ and $j_D = \max\{j : b_j \in D\}$.

COMP(A, B)
Initialize: $D = \emptyset$, $i_D = j_D = 0$
while $D \not\supseteq \{a_1, ..., a_n\}$ and $D \not\supseteq \{b_1, ..., b_n\}$
and ($i_D < n - 1$ or $j_D < n - 1$)
do COMPARE(A_{i_D+1}, B_{j_D+1})
If it returns $A \equiv B$, return $A \equiv B$ and stop
od
Return $A \not\equiv B$ and stop.

Example 1.

$A = (1, 1, 1, 1, 1, 2, 0, 1) = (a_1, ..., a_8)$,

$B = (1, 1, 1, 1, 1, 1, 2, 0) = (b_1, ..., b_8)$,

$A_1 = [1, 1, 1, 1, 1, 2, 0, 1]$,

$B_1 = [1, 1, 1, 1, 1, 1, 2, 0]$,

$D = \emptyset$, $\quad i_D = j_D = 0$.

We first call to COMPARE(A_1, B_1). It compares a_i with b_i for $i = 1, ..., 6$ and then discovers that $b_6 \prec a_6$. Thus $D \leftarrow D \cup \{a_1, ..., a_6\}$, $i_D = 6, j_D = 0$. $A_{i_D+1} = A_7 = [0, 1, 1, 1, 1, 1, 1, 2]$. We now call COMPARE($A_7, B_1$). It compares a_7 and b_1 and since $a_7 \prec b_1$ we have: $D \leftarrow D \cup \{b_1\}$, $j_D = 1$ and we call COMPARE(A_7, B_2). Now $a_7 \prec b_2$ and in fact $a_7 \prec b_j$ for all $j < 8$ and therefore we'll successively call COMPARE(A_7, B_j) for all $j < 8$ and update D to be $\{a_1, ..., a_6, b_1, ..., b_7\}$. Now $i_D = 6$ and we call COMPARE(A_7, B_8) which halts after 8 comparisons returning $A \equiv B$.

If we change the example a little bit by setting $b_7 = 3$, we'll still have the same sequence of calls to COMPARE, but COMPARE(A_7, B_8) will discover inequality in the last comparison and b_8 will be added to D. Then $D \supseteq \{b_1, ..., b_8\}$ and therefore COMP(A, B) will terminate, returning $A \not\equiv B$.

In both cases we have used 21 comparisons. It is easy to see that if $A_1 = [1, 1, ..., 1, 2, 0, 1]$ and $B_1 = [1, 1, ..., 1, 2, 0]$ and both have length n, then COMP(A, B) will use $3n - 3$ comparisons. Later on we'll show that this is an upper bound on its performance.

3. Correctness

Lemma 4. *During the execution of* COMP(A, B), D *always has the form* $\{a_1, ..., a_{i_D}, b_1, ..., b_{j_D}\}$, *i.e., no element between* a_1 *and* a_{i_D} *or between* b_1 *and* b_{j_D} *is missing.*

The proof follows from the fact that D is updated only in one of the two ways $D \leftarrow D \cup \{a_{i_D+1}, ..., a_l\}$ or $D \leftarrow D \cup \{b_{j_D+1}, ..., b_l\}$.

Theorem 1. COMP(A, B) *always terminates.*

Proof. If for some i and j, COMPARE(A_i, B_j) returns $A \equiv B$, then COMP(A, B) also returns the same and stops. If COMPARE(A_i, B_j) does not return $A \equiv B$ for any i and j, then $i_D + j_D$ is incremented by at

least one, with each iteration of COMPARE. Thus, eventually either $i_D = n$ or $j_D = n$ and by Lemma 4, either $D \supseteq \{a_1, ..., a_n\}$ or $D \supseteq \{b_1, ..., b_n\}$ and COMP(A, B) terminates.

Theorem 2. COMP(A, B) *returns* $A \equiv B$ *iff* $A \equiv B$.

Proof. Obviously, if COMP(A, B) returns $A \equiv B$, then $A \equiv B$. If COMP(A, B) does not return $A \equiv B$, then it returns $A \not\equiv B$, since by Theorem 1, it always terminates. If either $D \supseteq \{a_1, ..., a_n\}$ or $D \supseteq \{b_1, ..., b_n\}$, then by Lemma 3, $A \not\equiv B$. If $i_D = j_D = n - 1$, then $A \equiv B$ only if $A_{i_0} = A_n = B_n = B_{j_0}$. However, in this case $A_1 = B_1$ and that should have been detected by COMPARE(A_1, B_1) and since it didn't, $A \not\equiv B$.

4. Complexity

Definitions. (1) We say that COMPARE(A_i, B_j) is *successful* if it returns $A \equiv B$. Otherwise it is *unsuccessful.*

(2) Let D^* denote the set D, just before the last call to COMPARE.

Lemma 6. *The total number of comparisons that are made by all the* COMPAREs, *except the last one, is* $|D^*|$.

Proof. Every COMPARE that is not the last one is unsuccessful and increments $|D|$ exactly by the number of comparisons that it makes. The only (unsuccessful) COMPARE which might increment $|D|$ by less than the number of comparisons that it makes, is the last one (which has been excluded). These facts can be easily verified from the definition of COMPARE.

Lemma 7. (a) $|D^*| \leqslant 2n - 3$,

(b) *the algorithm uses at most* $3n - 3$ *comparisons.*

Proof. (a) If $|D^*| \geqslant 2n - 2$, then either $\{a_1, ..., a_n\} \subseteq D^*$ or $\{b_1, ..., b_n\} \subseteq D^*$ or $i_D = j_D = n - 1$. All 3 cases are impossible according to the definition of D^*,

(b) follows immediately from (a) and Lemma 6 since the last call to COMPARE can use at most n comparisons.

The following two examples show that COMP does not perform much better, even when restricted to binary lists. In the first example $A \equiv B$ and COMP(A, B) makes $3n - 3$ comparisons, and in the second, $A \not\equiv B$ and COMP(A, B) makes $3n - 7$ comparisons.

Example 2. $A_1 = [1, 1, ..., 1, 0]$; $B_1 = [1, ..., 0, 1]$.

Example 3. $A_1 = [1, ..., 1, 0, 1, 1, 0, 0]$;
$B_1 = [1, ..., 1, 0, 1, 0, 0, 1]$.

References

[1] A.G. Akl and G.T. Toussaint, An improved algorithm to check for polygon similarity, Information Processing Lett. 7 (3) (1978) 127–128.

[2] R. Carhart, Stanford Computer Science Department, private communication.

[3] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.

[4] G. Manacher, An application of pattern matching to a problem in geometrical complexity, Information Processing Lett. 5 (1) (1976) 6–7.