AN OPTIMAL ALGORITHM FOR COMPUTING THE REPETITIONS IN A WORD

Max CROCHEMORE

Laboratoire d'Informatique, Université de Haute – Normandie, BP 67 76130, Mont-Saint-Aignan, France

Received 27 January 1981; revised version received 14 May 1981

Analysis of algorithm, optimality, partitioning, repetitions in words

A word has a repetition when it has at least two consecutive equal factors. For instance, abab is a repetition (a square) in aababba.

Recently, it has been proved that the set of words containing a square is not context-free [3,7].

This paper presents an algorithm to compute all the repetitions of primitive factors in a word x in time $O(|x| \log_2 |x|)$. A straightforward adaption of the Knuth, Morris and Pratt's string-matching algorithm [5] also allows to solve the problem, but in time $O(|x|^2)$.

Main and Lorentz have given an $O(|x| \log_2 |x|)$ algorithm to find one square in a word x. Their method cannot be directly extended to solve the present problem since they eliminate many repetitions when they are guaranteed to find another one later in the search.

Our algorithm uses an improved version of the well-known partitioning technique [1] for refinements of equivalence relations. This version has already been fruitful in a problem concerning partitions on graphs [2].

The optimality of the algorithm is proved by showing that there exist words which have indeed $O(|x| \log_2 |x|)$ repetitions. These particular words are Fibonacci words.

With a slight modification, the algorithm gives the maximal repetitions of a word. This algorithm is also optimal since it computes all the $O(|x| \log_2 |x|)$ maximal repetitions of a Fibonacci word x in time $O(|x| \log_2 |x|)$.

1. Repetitions in words

Let A be a finite alphabet and A^* be the free monoid generated by A.

The length of a word x in A^* is denoted by $|x|$. Let $x = x_1 x_2 \cdots x_n$ be a word ($x_i \in A$). A *position* of x is, as in [4], any integer in $\{1, 2, ..., n\}$.

A word u of length p is said to *occur* at position i in x if

$$i + p - 1 \leqslant n \quad \text{and} \quad u = x_i x_{i+1} \cdots x_{i+p-1}.$$

As usual, a word is said to be *primitive* if it cannot be written v^e with $v \in A^*$ and $e \geqslant 2$.

Then, a repetition in x is a non trivial power of a primitive word which occurs in x.

More accurately, a *repetition* in x is defined to be a triple (i, p, e) so that, setting $u = x_i \cdots x_{i+p-1}$, one has:
u^e occurs at position i in x.
u^{e+1} does not occur at position i in x.
u is primitive.

The integers p and e are called respectively the *period* and the *exponent* of the repetition (i, p, e).

For instance, (1, 3, 2), (3, 1, 2), (4, 2, 2) and (5, 2, 2) are repetitions in abaababa.

Maximal repetitions are also considered and defined by: a repetition (i, p, e) in x is *maximal* if $i - p \leqslant 0$ or $x_i x_{i+1} \cdots x_{i+p-1}$ does not occur at position $i - p$.

The algorithm given in this paper computes, for a word x in A^*, its set of repetitions or only its set of maximal repetitions.

2. Equivalences on positions

A sequence $(E_p)_{p \geqslant 1}$ of equivalences on the positions in a word is defined as follows:

Let $x \in A^*$, $n = |x|$ and $p \geqslant 1$; then $(i, j) \in E_p$ iff $i + p - 1 \leqslant n$, $j + p - 1 \leqslant n$ and $x_i \cdots x_{i+p-1} = x_j \cdots x_{j+p-1}$.

So, two positions in x are equivalent according to E_p when the factors of x of length p and starting at i and j are equal.

For each $p \geqslant 1$, is also defined a function on the positions of x, which gives, for a position i, its difference to the least position in the same equivalence class:

$$D_p(i) = \begin{cases} \text{the least integer } k > 0 \\ \quad \text{s.t. } (i, i+k) \in E_p\,, \\ \infty \quad \text{if there is no such } k. \end{cases}$$

Then, repetitions in x are characterized in term of differences D_p:

Lemma 1. (i, p, e) is a repetition iff

$$D_p(i) = D_p(i+p) = \cdots = D_p(i + (e-2)p) = p$$

and

$$D_p(i + (e-1)p) \neq p.$$

Maximal repetitions are characterized in the same way:

Lemma 2. (i, p, e) is a maximal repetition iff (i, p, e) is a repetition and $i - p \leqslant 0$ or $D_p(i-p) \neq p$.

Proof of Lemma 1. Of course, the conditions are sufficient. Now, if (i, p, e) is a repetition we have:

$$\forall j \in \{i, i+p, \ldots, i+(e-2)p\} \quad D_p(j) \leqslant p,$$

and

$$D_p(i + (e-1)p) \neq p.$$

Suppose that $D_p(j) = p' < p$ for one j. The word $u = x_j \cdots x_{j+p-1}$ occurs also at positions $j + p'$ and $j + p$. In such a situation, denoting by v the word $x_j \cdots x_{j+p'-1}$ and by w the word $x_{j+p'} \cdots x_{j+p-1}$ it can easily be seen that u = vw = wv. In this case u is a power of a word in A^* [6] that contradicts the fact that u is primitive.

3. The basic lemma for computing the equivalences

One can easily check that any equivalence E_{p+1} is a refinement of $E_p(E_p \geqslant E_{p+1})$; furthermore, there clearly exists a smallest integer N, $1 \leqslant N \leqslant n$, so that

$$E_1 > E_2 > \cdots > E_N,$$

and $E_N = E_{N+1} = \cdots$ is the equality relation on $\{1, \ldots, n\}$.

The computation of the equivalences E_p may be done by the classical Moore's algorithm which computes successively $E_1, E_2, \ldots, E_N$. It is based on the relation:

$$(i, j) \in E_p \quad \text{iff } (i, j) \in E_{p-1},$$

and

$$(i+1, j+1) \in E_{p-1}.$$

Exploiting this relation directly leads to an $O(n^2)$ algorithm to compute the equivalences.

The other classical partitioning algorithm, Hopcroft's one [1], does not work for this problem since it computes E_N via other equivalences than the E_p' s.

The method retained here was used in [2] to partition graphs. It leads to an $O(n \log_2 n)$ algorithm.

Let us consider two consecutive values of the equivalences, E_{p-1} and E_p. Let $\{C_1, \ldots, C_q\}$ be the equivalence classes according to E_p (E_p-classes) and $\{C'_1, \ldots, C'_{q'}\}$ the E_{p-1}-classes. E_p being a refinement of E_{p-1}, each E_{p-1}-class is a union of E_p-classes.

A *choice function* is a function

$$f\colon \{C'_1, \ldots, C'_{q'}\} \to \{C_1, \ldots, C_q\},$$

with the properties: for any C' in $\{C'_1, \ldots, C'_{q'}\}$ $[f(C') \subset C'$ and for any C in $\{C_1, \ldots, C_q\}$ $C \subset C' \Rightarrow |C| \leqslant |f(C')|]$.

So, f associates to each E_{p-1}-class one of its E_p-subclasses of maximal size.

Given a choice function f, each E_p-class $f(C')$ is called a *big class*; the others are called *small classes.* Of course, there are as many big E_p-classes than E_{p-1}-classes. In particular, $E_p = E_{p-1}$ iff there is no small E_p-class. By definition, all the E_1-classes are small.

Now, a new sequence $(S_p)_{p \geqslant 1}$ of equivalences on the positions of x are defined:

$$(i, j) \in S_p \quad \text{iff} \begin{cases} (i, j) \in E_p \text{ or,} \\ \text{both i and j are in big } E_p\text{-classes.} \end{cases}$$

Equivalently we have:

$(i, j) \in S_p$ iff for any small E_p-class C,

$i \in C$ iff $j \in C$.

Lemma 3. For any $p \geq 1$, $(i, j) \in E_{p+1}$ iff $(i, j) \in E_p$ and $(i + 1, j + 1) \in S_p$.

Denoting by $\tilde{S}_p$ the equivalence:

$(i, j) \in \tilde{S}_p$ iff $(i + 1, j + 1) \in S_p$,

Lemma 3 asserts that $E_{p+1} = E_p \cap \tilde{S}_p$.

Proof. E_p being a refinement of S_p we have $E_{p+1} \subset E_p \cap S_p$. Let i and j be two positions such that

$(i, j) \in E_p$ and $(i + 1, j + 1) \in S_p$.

If i + 1 is in a small E_p-class then j + 1 is in the same E_p-class; so, $(i, j) \in E_{p+1}$. If i + 1 is in a big E_p-class, so it is for j + 1. From $(i, j) \in E_p$ we deduce $(i + 1, j + 1) \in E_{p-1}$, which proves that i + 1 and j + 1 must be in the same big E_p-class. Thus, we have again $(i, j) \in E_{p+1}$.

4. Outline of the algorithm

A schema of the algorithm is drawn in Fig. 1. From the word x, E_1 and D_1 are computed and their values put in E and D. The indices of the E_1-classes (which are all small) are put in SMALL. Then, in the "while" loop, the successive values of E are computed using Lemma 3. The difference function D is updated at the same time, and the new small E-classes are determined and memorized in SMALL. At the beginning of each execution of the loop, the new repetitions are calculated as stated in Lemma 1.

It is shown in the next section how to implement steps 5 and 6 efficiently, with a time complexity

$$O\left(\sum_{s \in \text{SMALL}} |\text{E-class } s|\right),$$

that is, with a complexity proportional to the union of small classes. Thus, the cost of all executions of steps 5 and 6 is

$$C = \sum_{p=1}^{N} \left(\sum_{s \text{ index of a small } E_p\text{-class}} |E_p\text{-class } s| \right),$$

where N is the first integer such that $E_N = E_{N+1}$ or equivalently such that E_{N+1} has no small equivalence class.

Lemma 4. $C \leq n \log_2(n - m + 1)$ where m is the number of distinct letters in x.

Proof. Consider a position i in a small E_p-class C, and let C′ be its E_{p-1}-class. By definition of the small classes (and choice functions) $|C| \leq |C'|/2$. Thus, a position i cannot belong to a small class more than $\log_2(n - m + 1)$, since the E_1-class of i has a cardinality less than n − m + 1. As there are n positions, $C \leq n \log_2(n - m + 1)$.

5. The algorithm

The algorithm that gives in R the repetitions in a word x is given in Fig. 2 as a procedure named REP. It parallels the schema in Fig. 1.

The data structures used to implement the algorithm are now described.

The equivalence E is represented twice: an array E gives for each position the index of its E-class; a double-linked list ECLASS gives for each equivalence class index the positions in the equivalence class. Doing so, transferring a position from an E-class to another is realized in constant time. To each E-class is associated its number of elements.

A stack NEWINDEX contains the available indices of E-classes. This stack may be seen as a 'garbage collection'. An index k is available when Eclass(k) is empty.

The difference function D is realized by an array; simultaneously a double-linked list DCLASS is maintained and gives for each period p the set of positions i satisfying D(i) = p. The function D together with the list DCLASS permit a search of the repetitions of period p linear in their number.

Steps 5.1 and 5.2 realize step 5 in Fig. 1. First, in step 5.1, the small E-classes are copied in a queue

```
       procedure REP(x)
(1)       define E to be E_1 on the word x; define D to be D_1;
          p ← 1; make R empty; SMALL ← {indices of E-classes};
(2)       while SMALL ≠ ∅ do
(3)          begin add to R the repetitions of period p (Lemma 1);
(4)             p ← p + 1; if p > |x|/2 then return R;
(5)             E ← E ∩ S̃ (Lemma 3); update D from the value of E,
(6)             SMALL ← {indices of small E-classes};
             end;
       return R.
```

Fig. 1. Schema of the repetition-searching algorithm.

QUEUE in order to preserve the increasing order on the positions in each small class. At the same time the set, SPLIT, of E-classes submitted to the 'splitting instruction' 5.2 is created. For each E-class k in SPLIT, a set SUBCLASS(k) is initialized to contain its subclass indices, together with a variable LAST-SMALL(k). This indicator gives in step 5.2 the last small class s that has been used to split the E-class k.

During step 5.2 the equivalence classes are split. One position at a time is transferred to a new class $\overline{k}$, from the E-class k. Let us assume that i′ is the last position in ECLASS(k) that has been transferred to a class k′, using a small class s′; in this case LAST-SMALL(k) = s′; if s′ is used again to transfer i into ECLASS($\overline{k}$) then i and i′ are equivalent according to the value of E being computed and $\overline{k}$ is defined to be k′. If not, a new index is extracted from NEWINDEX to define $\overline{k}$, and LASTSMALL(k) is set to be s.

While a position is transferred, D and DCLASS are updated. The computation of D use heavily the fact that positions in equivalence classes are in increasing order.

At step 6, a new value of SMALL is calculated. The array that gives the number of elements in each E-class allows to find the small classes efficiently.

Theorem 5. The procedure REP in Fig. 2 computes all the repetitions in a word x.

Proof. It is easy to see that the algorithm stops. The computation of a new value of the equivalence E is done in steps 5.1 and 5.2 exactly as stated in Lemma 3. If we assume that D is correctly calculated, then from Lemma 1 it can be shown that all the repetitions of period p are added to R at step 3.

It remains to prove that D is well updated. At each execution of the while loop 5.2 exactly one position i is transferred from its equivalence class k to another $\overline{k}$. If i′ is the position that preceeds i in ECLASS(k) then the value of $D_p(i')$ after i has been extracted from ECLASS(k) is $D_{p-1}(i') + D_{p-1}(i)$ since positions in ECLASS(k) are in increasing order. When i is added to ECLASS($\overline{k}$) its predecessor i″ in ECLASS($\overline{k}$) must satisfy

$$D_p(i'') = i - i'',$$

since the positions in the small classes (copied in QUEUE) are in increasing order. Furthermore, i being the greatest position in ECLASS(k), we have $D_p(i) = \infty$. These three points correspond to what is done during step 5.2.

The procedure REP may be immediately modified to calculate maximal repetitions in the word x. Regarding Lemma 2 we have only to move the instruction 3.1 after the step 3.2. Let this new procedure be called REPMAX.

Theorem 6. The procedure REPMAX computes all the maximal repetitions of a word x.

Theorem 7. The time complexity of procedure REP (or REPMAX) is $O(|x| \log_2 |x| + |A| |x|)$.

Proof. Step 1 in Fig. 2 contributes to $O(m|x|)$ in the total complexity, where m is the number of distinct letters in the word x. This is bounded by $O(|A| |x|)$.

Next, we discuss the complexity of the "while" loop 2. All the executions of step 3 take a time proportional to the number of repetitions in the word x. This number is bounded by $|x| \log_2 |x|$ [6].

The cost of the executions of steps 5.1, 5.2 and 6

```
        procedure REP(x)
        for k ← 2n step 1 until 1 do begin push k onto NEWINDEX; make ECLASS(k) empty;
                                     end;
(1)     for i ← 1 until n do begin if (x_i already occurs at j) then k ← E(j)
                                   else pop k from NEWINDEX; E(i) ← k; add i at the end of ECLASS(k);
                             end;
        define D; put in same DCLASS the positions that have same values of D; p ← 1; make R, QUEUE, SPLIT empty;
         SMALL ← {indices of the E-classes};
(2)     while SMALL ≠ ∅ do
        begin comment computation of the repetitions of period p;
(3)        while DCLASS(p) ≠ ∅ do
           begin i ← a position in DCLASS(p);
              repeat i ← i + p until D(i) ≠ p; e ← 1;
              repeat begin i ← i − p; e ← e + 1;
(3.1)                    add (i, p, e) to R; erase i from DCLASS(p);
                     end;
              until (i − p ⩽ 0 or D(i − p) ≠ p);
(3.2)         comment see computation of maximal repetitions;
           end;
(4)        p ← p + 1; if p > n/2 then return R;
           comment copy of small classes in QUEUE;
(5.1)      while SMALL ≠ ∅ do
           begin extract s from SMALL;
              for j from the first to the last element of ECLASS(s) do
              begin if j ≠ 1 then
                 begin add (j, s) at the end of QUEUE; k ← E(j − 1);
                    if k ∉ SPLIT then
                    begin add k to SPLIT; set SUBCLASS(k) = {k}; LASTSMALL(k) ⇐ 0;
                    end;
                 end;
              end
           comment computation of the new values of E and D;
(5.2)      while QUEUE ≠ ∅ do
           begin (j, s) ⇐ the first pair in QUEUE; i ⇐ j − 1; k ← E(i);
              if LASTSMALL(k) ≠ s then
              begin LASTSMALL(k) ← s; pop NI from NEWINDEX; add NI to SUBCLASS(k);
              end;
              k̄ ← the last index put in SUBCLASS(k);
              if (i has a predecessor i' in ECLASS(k)) then
              begin D(i') ← D(i') + D(i); transfer i' to DCLASS(D(i'));
              end;
              transfer i at the end of ECLASS(k̄); E(i) ← k̄; D(i) ← ∞; transfer i to DCLASS(∞);
              if (i has a predecessor i' in ECLASS(k̄)) then
              begin D(i') ← i − i'; transfer i' to DCLASS(D(i'));
              end;
           end;
           comment determination of the small classes;
           while SPLIT ≠ ∅ do
           begin extract k from SPLIT;
              if |ECLASS(k)| = 0 then
              begin push k onto NEW INDEX; erase k from SUBCLASS(k);
              end;
              add to SMALL all the indices in SUBCLASS(k) but one, corresponding to a greatest E-class;
           end;
        end;
        return R.
```

Fig. 2. Searching repetitions in a word x.

is proportional to the length of QUEUE which is

$$\sum_{\substack{s \text{ index of} \\ \text{a small E-class}}} |\text{ECLASS}(s)|.$$

Thus, applying Lemma 4, the aggregate cost of all the executions of steps 5.1, 5.2 and 6 is $O(|x| \log_2 |x|)$.

6. Optimality

Theorem 8. The procedure REP is optimal in the class of algorithms computing all the repetitions of a word.

The proof is a direct consequence of Lemma 10 on the number of squares in Fibonacci words. Observing that Fibonacci words do not contain repetition of exponent 4, together with Lemma 10, we obtain also the optimality of the procedure REPMAX:

Theorem 9. The procedure REPMAX is optimal in the class of algorithms computing all the maximal repetitions of a word.

Lemma 10. Let us define the sequence of Fibonacci words by: $f_0 = b$, $f_1 = a$ and $f_{q+1} = f_q f_{q-1} q$ integer $\geqslant 1$. Then, the number R_q of squares (repetition of exponent 2) in f_q satisfy, for any $q \geqslant 5$:

$$R_q \geqslant \tfrac{1}{6}|f_q| \log_2 |f_q|.$$

Proof. The property can be checked for $q = 5$ and 6. We proceed by induction on q. Suppose $q \geqslant 6$ and consider word f_{q+1} which is $f_q f_{q-1}$. Then

$$R_{q+1} = R_q + R_{q-1} + r_{q+1},$$

where r_{q+1} is the number of squares in $f_{q+1} = f_q f_{q-1}$ that are neither squares in f_q nor in f_{q-1}, i.e. squares that overlap over the border line between f_q and f_{q-1}. By induction hypothesis, we have:

$$R_{q+1} \geqslant \tfrac{1}{6} |f_q| \log_2 |f_q| + \tfrac{1}{6} |f_{q-1}| \log_2 |f_{q-1}| + r_{q+1}.$$

To get the results, it suffices to prove:

$$\tfrac{1}{6} |f_q| \log_2 |f_q| + \tfrac{1}{6} |f_{q-1}| \log_2 |f_{q-1}| + r_{q+1} \geqslant$$

$$\geqslant \tfrac{1}{6} |f_{q+1}| \log_2 |f_{q+1}|,$$

or

$$r_{q+1} \geqslant \tfrac{1}{6} \geqslant |f_q| \log_2 \frac{|f_{q+1}|}{|f_q|} + \tfrac{1}{6} |f_{q-1}| \log_2 \frac{|f_{q+1}|}{|f_{q-1}|},$$

using the relation $|f_{q+1}| = |f_q| + |f_{q-1}|$.

It is well known that Fibonacci words satisfy, for $q \geqslant 4$

$$|f_q| / |f_{q-1}| \leqslant |f_4| / |f_3| = \tfrac{5}{3}.$$

So it remains to prove that

$$r_{q+1} \geqslant \tfrac{1}{6}(|f_q| + |f_{q-1}|) \log_2 \tfrac{8}{3},$$

or (1)

$$r_{q+1} \geqslant \tfrac{1}{4} |f_{q+1}|.$$

First, we prove that f_{q+1} contains $|f_{q-3}| + 1$ squares of period $|f_{q-1}|$; so, these squares contribute to r_{q+1}. We have successively:

$$\begin{aligned} f_{q+1} &= f_q f_{q-1} = f_{q-1} f_{q-2} f_{q-2} f_{q-3} \\ &= f_{q-1} f_{q-2} f_{q-3} f_{q-4} f_{q-3} \\ &= f_{q-1} f_{q-1} f_{q-4} f_{q-3}. \end{aligned}$$

The square $f_{q-1} f_{q-1}$ is then a prefix of f_{q+1}.

For $q \geqslant 6$, f_{q-3} is a prefix of $f_{q-4} f_{q-3}$ since:

$$\begin{aligned} f_{q-4} f_{q-3} &= f_{q-4} f_{q-4} f_{q-5} \\ &= f_{q-4} f_{q-5} f_{q-6} f_{q-5} \\ &= f_{q-3} f_{q-6} f_{q-5}. \end{aligned}$$

The word f_{q-3} being also a prefix of f_{q-1} we get $|f_{q-3}|$ other squares of period $|f_{q-1}|$.

Secondly, f_{q+1} may be written $f_{q-1} f_{q-2} f_{q-2} f_{q-3}$. Analogously, we get $|f_{q-3}| + 1$ squares which contributes to r_{q+1}, since f_{q-3} is a prefix of f_{q-2}.

So, for $q \geqslant 6$, we have $r_{q+1} > 2|f_{q-3}|$. The result (1) follows from the inequality:

$$|f_{q-3}| \geqslant \tfrac{1}{8} |f_{q+1}|.$$

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, MA, 1974) 157–162.

[2] A. Cardon and M. Crochemore, Partitioning a graph in $O(|A| \log_2 |V|)$, Theoret. Comput. Sci., to appear.

[3] A. Ehrenfeucht and G. Rosenberg, On the separating

power of EOL systems, RAIRO, to appear.
[4] M.A. Harrison, Introduction to Formal Language Theory (Addison-Wesley, MA, 1978).
[5] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern-matching in strings, SIAM J. Comput. 6 (1977) 323–350.
[6] A. Lentin and M.P. Schutzenberger, A combinatorial problem in the theory of free monoids, in: Combinatorial Mathematics and Its Applications (University of North Carolina Press, NC, 1969) 128–144.
[7] R. Ross and R. Winklmann, Repetitive strings are not context-free, CS-81-070 (Washington State University, Pullman, WA, 1981).