Theoretical Computer Science 521 (2014) 29–41

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Extracting powers and periods in a word from its runs structure

M. Crochemore [a,b], C.S. Iliopoulos [a,c], M. Kubica [d], J. Radoszewski [d,*], W. Rytter [d,e], T. Waleń [f,d]

[a] King's College London, London WC2R 2LS, UK
[b] Université Paris-Est, France
[c] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth, WA 6845, Australia
[d] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
[e] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, ul. Chopina 12/18, 87-100 Toruń, Poland
[f] Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, ul. Ks. Trojdena 4, 02-109 Warsaw, Poland

ARTICLE INFO

Article history:
Received 13 June 2013
Received in revised form 31 October 2013
Accepted 15 November 2013
Communicated by D. Perrin

Keywords:
Run in a word
Square
Local period
Primitive word

ABSTRACT

A breakthrough in the field of text algorithms was the discovery of the fact that the maximal number of runs in a word of length n is $O(n)$ and that they can all be computed in $O(n)$ time. We study some applications of this result. New simpler $O(n)$ time algorithms are presented for classical textual problems: computing all distinct k-th word powers for a given k, in particular squares for $k = 2$, and finding all local periods in a given word of length n. Additionally, we present an efficient algorithm for testing primitivity of factors of a word and computing their primitive roots. Applications of runs, despite their importance, are underrepresented in existing literature (approximately one page in the paper of Kolpakov and Kucherov, 1999 [25,26]). In this paper we attempt to fill in this gap. We use Lyndon words and introduce the Lyndon structure of runs as a useful tool when computing powers. In problems related to periods we use some versions of the Manhattan skyline problem.

1. Introduction

Repetitions and periodicities in words are two of the fundamental topics in combinatorics on words [1,10,28]. These notions are widely used in many fields, such as combinatorics on words, pattern matching, automata theory, formal language theory, data compression, molecular biology, etc. The most commonly studied types of repetitions are powers (i.e. squares, cubes) and runs (also called maximal repetitions). A power of a word is simply composed of a number of juxtaposed occurrences of the word. A run is an inclusion-maximal periodic factor of a word in which the shortest period repeats at least twice. A survey by Crochemore et al. [5] further explains the motivation and describes known combinatorial and algorithmic properties of powers and runs.

The notion of a run was introduced in [25,26,29]. The crucial property of runs is that their maximal number in a word of length n is $O(n)$, see Kolpakov and Kucherov [25,26]. Due to the work of several authors more accurate bounds have been

* Corresponding author. Tel.: +48 22 55 44 484; fax: +48 22 55 44 400.
E-mail addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), c.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), kubica@mimuw.edu.pl (M. Kubica), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), walen@mimuw.edu.pl (T. Waleń).

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.11.018

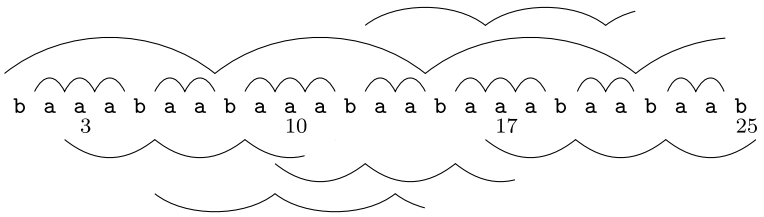

Fig. 1. The structure of runs in the word baaabaabaaabaabaaabaabaab. This word contains: 7 runs with period 1, 3 runs with period 3, 2 runs with period 4 and one run with period 7. The runs of period 3 are $(3, 10, 3)$, $(10, 17, 3)$ and $(17, 25, 3)$.

obtained, eventually it has been shown [6,31] that this number is between $0.944n$ and $1.029n$. The structure of all runs in a word provides succinct and very useful information about periodic properties of the word.

Several basic applications of runs were given in [25,26]. These include extracting all branching squares and counting primitively-rooted powers of a given exponent starting at each position of the word. We present a number of other algorithmic applications of runs. Our methods are considerably simpler than previously known linear time algorithms and show new structural relations between the notions of periodicity. A preliminary version of this paper appeared at SPIRE 2010 [7].

First we consider the problem of computing all *distinct* k-th powers in a word of length n, for a given k. It is known that the number of distinct squares ($k = 2$) does not exceed $2n$ [13,17,18] and for cubes ($k = 3$) there is a tighter $0.8n$ bound [27], which implies same bound for any value $k \geqslant 4$. Gusfield and Stoye [15] gave an $O(n)$ time algorithm for computing all the distinct squares. Unfortunately, this algorithm is complicated and uses suffix trees which are a rather heavyweight data structure and add a logarithmic factor depending on the size of alphabet in most implementations. We present a much simpler $O(n)$ time algorithm which computes all distinct k-th powers in a word of length n using suffix arrays instead of suffix trees. The most important combinatorial tool in our algorithm are Lyndon words, see [28].

Another application of the runs structure is the computation of local periods which are related to the critical factorizations of a word [10]. The known $O(n)$ time algorithm by Duval et al. [11] employs several different techniques modified in a non-trivial way. We present an equally efficient but simpler algorithm using the solution of the Manhattan Skyline Problem.

Finally, we consider factor-primitivity queries, which consist in checking, for any factor of a given word, whether it is primitive and what is its primitive root. This problem has potential applications in data compression, in particular, in run-length encoding and its derivatives. It is a special case of the factor period queries problem (finding the shortest period of specified factors) that was already considered, see [24]. We provide a solution to factor-primitivity problem with $O(n)$ space and $O(\log n)$ query time, $O(n \log^{\epsilon} n)$ space and $O(\log\log n)$ query time, or $O(n^{1+\epsilon})$ space and $O(1)$ query time, for any $\epsilon > 0$.

2. Preliminaries

Let u be a word of length n over a bounded alphabet Σ: $u = u[1]u[2]\ldots u[n]$. By $u[i\,..\,j]$ we denote a factor of u equal to $u[i]\ldots u[j]$. If $i = 1$ then it is called a prefix of u, and if $j = n$ then it is called a suffix of u. We say that an integer p is the (shortest) *period* of $u[1\,..\,n]$ (notation: $p = \mathsf{per}(u)$) if p is the smallest positive integer such that $u[i] = u[i + p]$ holds for all $1 \leqslant i \leqslant n - p$.

A *run* v (a maximal repetition) in the word u is an interval $[i, j]$ such that the shortest period $p = \mathsf{per}(v)$ of the associated factor $u[i\,..\,j]$ satisfies $2p \leqslant j - i + 1$, and the interval cannot be extended to the left nor to the right without violating the above property, that is, $u[i-1] \neq u[i+p-1]$ and $u[j-p+1] \neq u[j+1]$, provided that the respective letters exist. Denote by $\mathcal{R}(u)$ the set of all runs in u, each represented as a triple (i, j, p), see also Fig. 1. It is known that $|\mathcal{R}(u)| = O(n)$ [5] and all elements of $\mathcal{R}(u)$ can be computed in $O(n)$ time [26] (a more practical algorithm for computing all runs is given in [3]).

If $w^k = u$ (k is a positive integer) then we say that u is the k-th power of the word w. A *square* (*cube*) is the 2nd (3rd) power of a nonempty word. The *primitive root* of a word u, denoted $\mathsf{root}(u)$, is the shortest word w such that $w^k = u$ for some positive integer k. We call a word u *primitive* if $\mathsf{root}(u) = u$, otherwise it is called *non-primitive*.

Let us recall two useful data structures in word processing.

Suffix arrays. The suffix array of a word u consists in three tables: SUF, LCP and RANK, see Table 1. The SUF array stores the list of positions in u sorted according to the increasing lexicographic order of suffixes starting at these positions, i.e.:

$$u\big[\mathsf{SUF}[1]\,..\,n\big] < u\big[\mathsf{SUF}[2]\,..\,n\big] < \cdots < u\big[\mathsf{SUF}[n]\,..\,n\big].$$

Thus, indices of SUF are ranks of the respective suffixes in the increasing lexicographic order. The LCP array is also indexed by the ranks of the suffixes, and stores the lengths of the longest common prefixes of consecutive suffixes in SUF. Denote by $lcp(i, j)$ the length of the longest common prefix between $u[i\,..\,n]$ and $u[j\,..\,n]$ (for $1 \leqslant i, j \leqslant n$). Then, we set $\mathsf{LCP}[1] = -1$ and, for $1 < i \leqslant n$, we have:

$$\mathsf{LCP}[i] = lcp\big(\mathsf{SUF}[i-1], \mathsf{SUF}[i]\big).$$