Information Processing Letters 116 (2016) 241–244

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Computing runs on a general alphabet

Dmitry Kosolobov

Ural Federal University, Ekaterinburg, Russia

ARTICLE INFO

Article history:
Received 6 July 2015
Received in revised form 24 November 2015
Accepted 26 November 2015
Available online 30 November 2015
Communicated by Ł. Kowalik

Keywords:
Runs
General alphabet
Algorithms
Repetitions

ABSTRACT

We describe a RAM algorithm computing all runs (maximal repetitions) of a given string of length n over a general ordered alphabet in $O(n\log^{\frac{2}{3}} n)$ time and linear space. Our algorithm outperforms all known solutions working in $\Theta(n\log\sigma)$ time provided $\sigma = n^{\Omega(1)}$, where σ is the alphabet size. We conjecture that there exists a linear time RAM algorithm finding all runs.

1. Introduction

Repetitions in strings are fundamental objects in both stringology and combinatorics on words. In some sense the notion of *run*, introduced by Main [13], allows to grasp the whole repetitive structure of a given string in a relatively simple form. Informally, a run of a string is a maximal periodic substring that is at least as long as twice its minimal period (the precise definition follows). In [9] Kolpakov and Kucherov showed that any string of length n contains $O(n)$ runs and proposed an algorithm computing all runs in linear time on an integer alphabet $\{0, 1, \dots, n^{O(1)}\}$ and $O(n\log\sigma)$ time on a general ordered alphabet, where σ is the number of distinct letters in the input string. Recently, Bannai et al. described another interesting algorithm computing all runs in $O(n\log\sigma)$ time [1]. Modifying the approach of [1], we prove the following theorem.

Theorem. *For a general ordered alphabet, there is an algorithm that computes all runs in a string of length n in* $O(n\log^{\frac{2}{3}} n)$ *time and linear space.*

This is in contrast to the result of Main and Lorentz [14] who proved that any algorithm deciding whether a string over a general *unordered* alphabet has at least one run requires $\Omega(n\log n)$ comparisons in the worst case.

Our algorithm outperforms all known solutions when the number of distinct letters in the input string is sufficiently large (e.g., $\sigma = n^{\Omega(1)}$). It should be noted that the algorithm of Kolpakov and Kucherov can hardly be improved in a similar way since it strongly relies on a structure (namely, the Lempel–Ziv decomposition) that cannot be computed in $o(n\log\sigma)$ time on a general ordered alphabet (see [11]).

Based on some theoretical observations of [11], we conjecture that one can further improve our result.

Conjecture. *For a general ordered alphabet, there is a linear time algorithm computing all runs.*

2. Preliminaries

A *string of length n* over an alphabet Σ is a map $\{1, 2, \dots, n\} \mapsto \Sigma$, where n is referred to as the length of w, denoted by $|w|$. We write $w[i]$ for the ith letter of w and $w[i..j]$ for $w[i]w[i+1]\dots w[j]$. A string u is a *substring* (or a *factor*) of w if $u = w[i..j]$ for some i and j. The

E-mail address: dkosolobov@mail.ru.

http://dx.doi.org/10.1016/j.ipl.2015.11.016

pair (i, j) is not necessarily unique; we say that i specifies an *occurrence* of u in w. A string can have many occurrences in another string. A substring $w[1..j]$ (respectively, $w[i..n]$) is a *prefix* (respectively, *suffix*) of w. An integer p is a *period* of w if $0 < p \le |w|$ and $w[i] = w[i+p]$ for all $i = 1, \dots, |w|-p$; p is the *minimal period* of w if p is the minimal positive integer that is a period of w. For integers i and j, the set $\{k \in \mathbb{Z}: i \le k \le j\}$ (possibly empty) is denoted by $[i..j]$. Denote $[i..j) = [i..j-1]$ and $(i..j] = [i+1..j]$.

A *run* of a string w is a substring $w[i..j]$ whose period is at most half of the length of $w[i..j]$ and such that both substrings $w[i-1..j]$ and $w[i..j+1]$, if defined, have strictly greater minimal periods than $w[i..j]$.

We say that an alphabet is *general* and *ordered* if it is totally ordered and the only allowed operation is comparing two letters. Hereafter, w denotes the input string of length n over a general ordered alphabet.

In the *longest common extension (LCE)* problem one has to preprocess w for queries $LCE(i, j)$ returning for given positions i and j of w the length of the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. It is well known that one can perform the *LCE* queries in constant time after preprocessing w in $O(n \log \sigma)$ time, where σ is the number of distinct letters in w (e.g., see [7]). It turns out that the time consumed by the *LCE* queries is dominating in the algorithm of [1]; namely, one can prove the following lemma.

Lemma 1. *(See [1, Alg. 1 and Sect. 4.2].) Suppose we can answer in an online fashion any sequence of $O(n)$ LCE queries on w in $O(f(n))$ time for some function $f(n)$; then we can find all runs of w in $O(n + f(n))$ time.*

In what follows we describe an algorithm that computes $O(n)$ *LCE* queries in $O(n \log^{\frac{2}{3}} n)$ time and thus prove theorem using Lemma 1. The key notion in our construction is a *difference cover*. Let $k \in \mathbb{N}$. A set $D \subset [0..k)$ is called a difference cover of $[0..k)$ if for any $x \in [0..k)$, there exist $y, z \in D$ such that $y - z \equiv x \pmod{k}$. Clearly $|D| \ge \sqrt{k}$. Conversely, for any $k \in \mathbb{N}$, there is a difference cover of $[0..k)$ with $O(\sqrt{k})$ elements: for example, the difference cover $[0..\lfloor\sqrt{k}\rfloor] \cup \{2\lfloor\sqrt{k}\rfloor, 3\lfloor\sqrt{k}\rfloor, \dots\}$, which is depicted in Fig. 1. For further discussions and estimations of minimal difference covers, see [4,15,16].

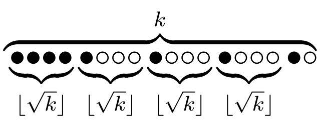

Fig. 1. Simple difference cover of $[0..k)$ with $k = 18$.

Example. The set $D = \{1, 2, 4\}$ is a difference cover of $[0..5)$.

x	0	1	2	3	4
y, z	1, 1	2, 1	1, 4	4, 1	1, 2

Our algorithm utilizes the following interesting property of difference covers.

Lemma 2. *(See [3].) Let D be a difference cover of $[0..k)$. For any integers i, j, there exists $d \in [0..k)$ such that $(i + d) \bmod k \in D$ and $(j + d) \bmod k \in D$.*

3. Longest common extensions

At the beginning, our algorithm fixes an integer τ (the precise value of τ is given below). Let D be a difference cover of $[0..\tau^2)$ of size $O(\tau)$. Denote $M = \{i \in [1..n]: (i \bmod \tau^2) \in D\}$. Obviously, we have $|M| = O(\frac{n}{\tau})$. Our algorithm builds in $O(\frac{n}{\tau}(\tau^2 + \log n)) = O(\frac{n}{\tau}\log n + n\tau)$ time a data structure that can calculate $LCE(x, y)$ in constant time for any $x, y \in M$. To compute $LCE(x, y)$ for arbitrary $x, y \in [1..n]$, we simply compare $w[x..n]$ and $w[y..n]$ from left to right until we reach positions $x + d$ and $y + d$ such that $x + d \in M$ and $y + d \in M$, and then we obtain $LCE(x, y) = d + LCE(x + d, y + d)$ in constant time. By Lemma 2, we have $d < \tau^2$ and therefore, the value $LCE(x, y)$ can be computed in $O(\tau^2)$ time. Thus, our algorithm can execute any sequence of $O(n)$ *LCE* queries in $O(\frac{n}{\tau}\log n + n\tau^2)$ time. Putting $\tau = \lceil \log^{\frac{1}{3}} n \rceil$, we obtain $O(\frac{n}{\tau}\log n + n\tau^2) = O(n \log^{\frac{2}{3}} n)$. Now it suffices to describe the data structure answering the *LCE* queries on the positions from M.

Let $i_1, i_2, \dots, i_m$ be the sequence of all positions from M in the increasing lexicographical order of the corresponding suffixes $w[i_1..n], w[i_2..n], \dots, w[i_m..n]$. Our algorithm builds a *longest common prefix array* $\mathsf{lcp}[1..m-1]$ such that $\mathsf{lcp}[j] = LCE(i_j, i_{j+1})$ for $j \in [1..m)$ and a *sparse suffix array* $\mathsf{sa}[1..n]$ such that $i_{\mathsf{sa}[x]} = x$ for $x \in M$ and $\mathsf{sa}[x] = 0$ for $x \notin M$. Obviously $LCE(i_j, i_k) = \min\{\mathsf{lcp}[j], \mathsf{lcp}[j+1], \dots, \mathsf{lcp}[k-1]\}$ for $j < k$. Based on this observation, we equip the lcp array with the *range minimum query (RMQ)* structure [5] that allows to compute $\min\{\mathsf{lcp}[j], \mathsf{lcp}[j+1], \dots, \mathsf{lcp}[k-1]\}$ for any $j < k$ in $O(1)$ time. Now, to answer $LCE(x, y)$ for $x, y \in M$, we first obtain $j = \mathsf{sa}[x]$ and $k = \mathsf{sa}[y]$ and then answer $LCE(i_j, i_k)$ using the RMQ structure on the lcp array. Since the RMQ structure can be built in $O(n)$ time [5], it remains to describe how to construct lcp and sa.

In general our construction is similar to that of [10]. We use the fact that the set M has "period" τ^2, i.e., for any $x \in M$, we have $x + \tau^2 \in M$ provided $x + \tau^2 \le n$. For simplicity, assume that $w[n]$ is a special letter that is smaller than any other letter in w. Our algorithm iteratively inserts the suffixes $\{w[x..n]: x \in M\}$ in the arrays lcp and sa from right to left. Suppose, for some $k \in M$, we have already inserted in lcp and sa the suffixes $w[x..n]$ for all $x \in M \cap (k..n]$. More precisely, denote by $i'_1, i'_2, \dots, i'_{m'}$ the sequence of all positions $M \cap (k..n]$ in the increasing lexicographical order of the corresponding suffixes $w[i'_1..n], w[i'_2..n], \dots, w[i'_{m'}..n]$; we suppose that $\mathsf{lcp}[j] = LCE(i'_j, i'_{j+1})$ for $j \in [1..m')$, $i'_{\mathsf{sa}[x]} = x$ for $x \in M \cap (k..n]$, and $\mathsf{sa}[x] = 0$ for $x \notin M \cap (k..n]$. We are to insert the suffix $w[k..n]$ in lcp and sa. In order to perform the insertions efficiently, during the construction, the arrays lcp and sa