Theoretical Computer Science 410 (2009) 3942–3948

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A fast algorithm for finding the positions of all squares in a run-length encoded string☆

J.J. Liu [a], G.S. Huang [b], Y.L. Wang [c,*]

[a] Department of Information Management, Shih Hsin University, Taipei, Taiwan

[b] Department of Computer Science and Information Engineering, National Chi Nan University, Nantou, Taiwan

[c] Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 24 October 2007
Received in revised form 13 February 2009
Accepted 28 May 2009
Communicated by A. Apostolico

Keywords:
Squares
Divide-and-conquer
Repetition
Run-length encoding

ABSTRACT

Squares are strings of the form ww where w is any nonempty string. Main and Lorentz proposed an $O(n \log n)$-time algorithm for finding the positions of all squares in a string of length n. Based on their result, we show how to find the positions of all squares in a run-length encoded string in time $O(N \log N)$ where N is the number of runs in this string, provided that we do not explicitly compute at all "trivial squares" occurring within runs. The algorithm is optimal and its time complexity is independent of the length of the original uncompressed string.

1. Introduction

A recent trend in stringology deals with string manipulations on compressed texts directly [17]. It is evident that data compression can reduce space for storing data significantly in a computer. The ordinary approach inflates the compressed data before using it, but this causes additional overheads on efficiency. However, data compression should not always be harmful to efficiency. In many applications, strings can be manipulated directly in the compressed form. Since compressed text usually uses less storage space, an algorithm for it is more efficient even if its time complexity takes the same order of magnitude when compared with an algorithm for the original uncompressed text. As a result, efficiency for both time and space is improved. In this paper, we consider a recognition problem on run-length encoded strings. We show how to locate the positions of all squares in a run-length encoded string in time $O(N \log N)$ where N is the number of runs in the string, provided that we do not explicitly compute at all "trivial squares" occurring within runs.

A *square* is an immediately repeated nonempty string, such as *aa*, *abab*, or *xyzxyz*. The interest in squares within strings has attracted the attention of researchers in diverse fields for a long time. In 1906, Thue discussed the construction of square-free infinite sequences [21]. In recent years, identifying occurrences of squares plays an important role in formal language theory, data compression, and computational molecular biology [1,8]. For a string of length n, a straightforward implementation would take $O(n^3)$-time to find all squares within this string by inspecting all possible substrings. Adapting KMP's failure function [10], the time complexity can be reduced to $O(n^2)$ (for more details, see [5]).

☆ This work was supported in part by the National Science Council of the Republic of China under contracts NSC 96-2221-E-260-018 and NSC 95-2221-E-260-016-MY2.

* Corresponding address: Department of Information Management, National Taiwan University of Science and Technology, No. 43, Sec. 4, Keelung Road, Taipei, Taiwan, ROC. Tel.: +886 49 2910960.

E-mail addresses: yuelwang@ncnu.edu.tw, ylwang@cs.ntust.edu.tw (Y.L. Wang).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.05.032

Table 1
Squares occurring within a run.

String	a^k for $k \geq 2$
Squares	a^ℓ for $2 \leq \ell \leq k$ and even, i.e., $([\ell, k], \ell)$
Example	String a^6
Squares	Five a^2s, three a^4s, and one a^6
$(\mathcal{I}, \ell)$	([2, 6], 2), ([4, 6], 4), ([6, 6], 6)

We remark that the total number of occurrences of squares in a^n is $\Theta(n^2)$, which indicates that an $O(n^2)$-time algorithm is optimal if one needs to enumerate one occurrence at a time. However, this observation does not block developing other faster algorithms for this problem. Main and Lorentz first observed that many squares in a string are related [13,14]. These related squares occur consecutively and all are with an equal length. Note that a square in a string can be uniquely specified by its length and ending position. And thus, they can also be parameterized as a whole unit by listing the interval $\mathcal{I}$ of their ending positions, together with the common length ℓ of every square, i.e., $(\mathcal{I}, \ell)$. See Table 1 for an example. Note that, in Table 1, ([2, 6], 2) means that there are five squares of length two and their ending positions are 2, 3, . . ., and 6, respectively. Furthermore, two consecutive squares of the same length are not necessary to have the same pattern. For example, in string *babaaabaaa*, ([9, 10], 8) contains squares *abaaabaa* and *baaabaaa*. As a consequence, it is possible to report a family of squares in constant time. They then extended their previous algorithm for searching for one square [13] to an algorithm for searching for all squares [14] in time $O(n \log n)$.

Different notions for representing squares have been developed. A square is called *primitive* if and only if it cannot be expressed as w^k for some string w and $k > 2$. Searching for all *primitive squares* can be done in $O(n \log n)$-time [2,3,20], which achieves the optimality in Fibonacci words [3]. A substring is called a *maximal repetition* if and only if its period is increased while extending to either direction. Recall that *the period* of a string $w = a_1 a_2 \ldots a_n$ is the smallest integer p such that $a_i = a_{i+p}$ for all i, provided $1 \leq i, i + p \leq n$. Maximal repetitions in a string can be identified in linear time under different settings: 'distinct' in [19], 'left-most' in [16], and 'all' in [11]. We remark that the result of [11] also implies an $O(n + S)$-time algorithm to report all squares in a string, where S is the output size.

A seemingly much simpler problem is to determine if a string is square-free, which can be done in linear time [4,5,15]. A surprising fact is that a string of length n can have at most $O(n)$ distinct squares and all of them can also be located in $O(n)$-time [9]. Online detection of squares is also discussed in [12], with time complexity $O(h \log^2 h)$ where h is the length of the longest prefix that is square-free. Delacourt et al. [6] and Garcia et al. [7] gave algorithms for detecting repetitions on multi-computers.

Run-length encoding uses a simple idea to compress strings. It divides a string into *runs*, each run consists of identical letters, and then represents the string by consecutive pairs of the representative letter and the length of the corresponding run. For example, the run-length encoded string of *bdcccaaaaaa* is $b^1d^1c^3a^6$. For more details, please see [18].

In this paper, inspired by [14], we show how to find the positions of all squares that use at least two distinct symbols in a run-length encoded string in time $O(N \log N)$ where N is the number of runs in this string. Like Main and Lorentz's result in [14], squares compressed in run-length can also be grouped so that each group can be uniquely specified by two parameters $(\mathcal{I}, \ell)$, i.e., an interval $\mathcal{I}$ of ending positions of squares together with the common length ℓ of every square. This allows our algorithm to 'report all squares' in sub-quadratic time. As for trivial squares occurring within runs, our algorithm simply reports their existence by mathematical expressions. Those squares can be trivially extracted by a supplementary computation in time $O(n)$. The time complexity of our result is optimal under the character-comparison model, and it is independent of the length of the original uncompressed string. In Section 2, we briefly introduce Main and Lorentz's idea in [14] on how to locate the positions of all squares in a string by applying the divide-and-conquer technique. It takes $O(n \log n)$-time where n is the length of the string. Then, we modify their algorithm and propose an $O(N \log N)$-time algorithm for run-length encoded strings in Section 3.

2. Main and Lorentz's idea

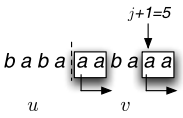
Main and Lorentz in [14] applied divide-and-conquer to locate the positions of all squares in a string. Suppose that a string x is divided into two nonempty strings u and v. If ww is a square of x, then clearly either ww occurs in u or in v, or as the concatenation of a suffix of u and a prefix of v. Therefore, if we can handle the last case well, squares in u and v can be found recursively by applying divide-and-conquer. More precisely, let

$$\texttt{square}(uv) = \texttt{square}(u) \cup \texttt{square}(v) \cup \texttt{cross}(u, v) \quad \text{for } u, v \text{ nonempty;} \tag{1}$$

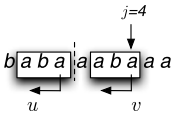
$$\texttt{square}(y) = \emptyset \quad \text{if } |y| \leq 1. \tag{2}$$

Here $\texttt{square}(x)$ is the locations of all squares occurring in x and $\texttt{cross}(u, v)$ records squares such that each one starts in u and ends in v. Clearly, we have to adjust the locations of squares in $\texttt{square}(u)$, $\texttt{square}(v)$, and $\texttt{cross}(u, v)$ when combining their results to obtain $\texttt{square}(uv)$. In order to evaluate $\texttt{cross}(u, v)$ efficiently, we need the following arrays:

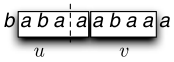
- $LP(v, j)$: the length of the longest substring of v that starts at position j (for $2 \leq j \leq |v|$) and matches a *prefix* of v;
- $LS(u, v, j)$: the length of the longest substring of v that ends at position j (for $1 \leq j \leq |v|$) and matches a *suffix* of u.



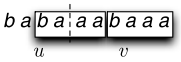
(a) $LP(v, 5) = 2$.



(b) $LS(u, v, 4) = 3$.



(c) A square with length 8 (boxed) ending at position 5 of v.



(d) A square with length 8 (boxed) ending at position 6 of v.

Fig. 1. An illustration for identifying squares in uv in Example 1 where $u = baba$ and $v = aabaaa$ are separated by a vertical dashed line.

Both arrays LP and LS can be computed in time $O(|uv|)$ by using the algorithms in [14,16], which are similar to the preprocessing procedure for the KMP algorithm described in [10]. The following lemma gives the algorithmic condition for detecting squares in $\texttt{cross}(u, v)$ whose center is at v by using arrays LP and LS.

Lemma 1 (*Repetition Lemma, [14]*). *Let j and k be integers with $1 \le j \le |v|$ and $j \le k < 2j$. There is a square of length $2j$ in uv ending at position k of v if and only if $2j - LS(u, v, j) \le k \le j + LP(v, j + 1)$.*

The meaning of Lemma 1 can be understood by observing the following fact. The condition $LS(u, v, j) + LP(v, j + 1) \ge j$ for some position j in v implies that a prefix of length $LP(v, j+1)$ in v connects to a suffix of length $LS(u, v, j)$ ending at position j. Furthermore, there are

$$\min\{LP(v, j + 1), j - 1\} + LS(u, v, j) - j + 1 \tag{3}$$

squares ending at consecutive positions of v with length $2j$. Recall that those squares can be reported by showing the interval of their ending positions and their common length, namely

$$([|u| + 2j - LS(u, v, j), |u| + j + \min\{LP(v, j + 1), j - 1\}], 2j)\,. \tag{4}$$

Symmetrically, reversing string uv, squares centered at u can also be computed in $O(|uv|)$-time. Thus, according to the above discussion, $\texttt{cross}(u, v)$ can be implemented in $O(|uv|)$-time.

Example 1. Consider $u = baba$ and $v = \underset{123456}{aabaaa}$. Let $j = 4$. Then, $LP(v, 5) = 2$ (in Fig. 1(a)) and $LS(u, v, 4) = 3$ (in Fig. 1(b)). Two squares ($LS(u, v, 4) + LP(v, 5) - j + 1 = 2$) of uv with length 8 ending at positions 5 and 6 (i.e. $2 \times 4 - 3 = 5 \le k \le 6 = 4 + 2$) in v are identified (see Fig. 1(c) and (d), respectively). This family of squares can be specified by ([5, 6], 8). Clearly, it will be adjusted to ([9, 10], 8), i.e., by adding $|u|$ to the two endpoints of interval [5, 6], in order to obtain their corresponding positions in uv. This result can also be obtained from Eq. (4) by the above setting.

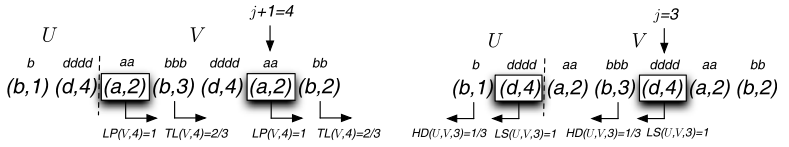
As for a string x of length n, the standard divide-and-conquer technique can be applied to identify all occurrences of squares by breaking x into u and v with nearly equal length, as shown in Eqs. (1) and (2). Hence, the time complexity of $\texttt{square}(x)$ can be obtained by solving $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n)$. Thus, $\texttt{square}(x)$ takes $O(n \log n)$-time to locate the positions of all squares in x.

3. Coping with run-length encoding

Recall that in run-length encoding, a string x is represented by $a_1^{r_1} a_2^{r_2} \dots a_N^{r_N}$ where a_i and a_{i+1} are different symbols, for $1 \le i < N$. When implementations are taken into consideration, a new symbol $\sigma(a^r) = (a, r) \in \Sigma \times \mathbb{N}$ replaces each run a^r. Thus, string x of N runs is represented by $\sigma(x) = (a_1, r_1)(a_2, r_2) \cdots (a_N, r_N)$. Conversely, given $X = \sigma(x)$, define $\sigma^-(X) = x$. Our algorithm computes the positions of all squares of x from $\sigma(x)$ and its time complexity is independent of the length of each run. For $X = \sigma(x)$, we denote (a_i, r_i) by $X[i]$, i.e., the ith run, and call a_i the *base character* of $X[i]$ and r_i the *exponent* of $X[i]$.

Next, we establish a correspondence of indices between x and $\sigma(x)$. Define $\rho : \Sigma^* \times \mathbb{N} \to \mathbb{Q}$, for indices of x to indices of $\sigma(x)$, as follows. Let $R_k = \sum_{i=1}^{k} r_i$. If i indicates the sth character of run t in x, that is $i = s + R_{t-1}$, then $\rho(x, i) = t - 1 + \frac{s}{r_t}$; otherwise, leave $\rho(x, i)$ undefined when $i < 0$ or $i > |x|$. We can also define the inverse of ρ, denoted by ρ^-, from indices of $X \in (\Sigma \times \mathbb{N})^*$ to indices of $x = \sigma^-(X)$, as $\rho^-(X, i) = i'$ if and only if there exists i' such that $\rho(x, i') = i$, and otherwise leave $\rho^-(X, i)$ undefined.

Main and Lorentz's algorithm cannot be applied directly to find all squares inside $\sigma(x)$ because some squares are missing by using their algorithm. For example, let $x = a^2b^3$. Then, $\sigma(x) = (a, 2)(b, 3)$ and no square can be found by using Main and Lorentz's algorithm (since $(a, 2)$ and $(b, 3)$ are different symbols). However, it is easy to see that a^2 and b^2 are squares in x. The situation may become more complicated when strings are more complex. For example, let $\sigma(x) = (a, 2)(b, 3)(a, 1)(b, 5)$,



(a) $LP(V,4) = 1$ and $TL(V,4) = 2/3$. (b) $LS(U,V,3) = 1$ and $HD(U,V,3) = 1/3$.

Fig. 2. An illustration of identifying squares in UV for $j = 3$ in Example 2 where $U = (b,1)(d,4)$ and $V = (a,2)(b,3)(d,4)(a,2)(b,2)$ are separated by a vertical dashed line.

then square $a^1b^3a^1b^3$ cannot be found. In the following paragraphs, we will show how to modify Main and Lorentz's algorithm so that the positions of all squares can still be found.

Observe that a square of x either occurs entirely within a run or overlaps with at least two consecutive runs (in the latter case in fact it has to span at least four runs). All squares in the former case can be identified in $O(N)$-time as follows. There are $O(k^2)$ squares in a^k and their positions can be expressed as $([2,k],2),([4,k],4),\ldots,([2\times\lfloor k/2\rfloor,k],2\times\lfloor k/2\rfloor)$. Actually, there is a more compact way to represent these positions, namely $([2t,k],2t)|_{t=[1,\lfloor k/2\rfloor]}$, in which we use another interval to describe the range of t, namely $t=[1,\lfloor k/2\rfloor]$. Thus, these $O(k^2)$ positions can be implicitly expressed by a mathematical expression in $O(1)$ time. As for the latter case when the divide-and-conquer technique is applied on X, we only need to handle the following situation.

D1: $X = UV$ and neither U nor V is empty.
D2: We want to identify each square that overlaps both $u = \sigma^-(U)$ and $v = \sigma^-(V)$. Let ww be such a square. Then, ww can be further divided into three cases:
Case A: centered at the junction of u and v;
Case B: centered at v;
Case C: centered at u.

Let $U = (u_1,\mu_1)(u_2,\mu_2)\cdots(u_P,\mu_P)$ and $V = (v_1,\nu_1)(v_2,\nu_2)\cdots(v_Q,\nu_Q)$. Since divide-and-conquer is always applied on the boundary of runs, Condition D1 implies $P, Q > 0$ and u_P is different from v_1.

It is clear that $LP(V,i)$ and $LS(U,V,j)$ for $2 \le i \le Q$ and $1 \le j \le Q$ can be evaluated in time $O(P+Q)$ by Main and Lorentz's algorithm as described in Section 2. However, $LP(V,i)$ might not be equal to $\rho(v, LP(v,i'))$ where $i' = \rho^-(V, i-1)+1$. For example, let $v = a^2b^2a^2b^3$ and $V = (a,2)(b,2)(a,2)(b,3)$. The fifth character in v is in the third run of V, i.e., $i' = 5$ and $i = 3$. Then, $LP(v,i') = LP(v,5) = 4$, i.e., the length of a^2b^2. However, $LP(V,i) = LP(V,3) = 1$ which reveals only the prefix a^2 of the actually matched a^2b^2 in $LP(v,5)$ and truncates the *tail* b^2.

The insufficiency for LP and LS to determine squares in UV can be supplemented by two additional tables TL and HD which are defined as follows. Let $k_1 = LP(V,i)$. If $k_1 \le Q-i$, then by the definition of LP, we have $V[1+k_1] \ne V[i+k_1]$. Let $X[\alpha_i] = (v_{\alpha_i}, \nu_{\alpha_i}) = V[1+k_1]$ and $X[\beta_i] = (v_{\beta_i}, \nu_{\beta_i}) = V[i+k_1]$. When v_{α_i} is the same as v_{β_i}, let $TL(V,i)$ be $\min\{\nu_{\alpha_i}, \nu_{\beta_i}\}/\nu_{\alpha_i}$. In other cases, let $TL(V,i) = 0$. Then, it is not difficult to see that

$$LP(V,i) + TL(V,i) = \rho(v, LP(v,i')) \quad \text{where } i' = \rho^-(V,i-1)+1. \tag{5}$$

Similarly, define $HD(U,V,j)$ to compensate the truncated head of $LS(u,v,j')$ where $j' = \rho^-(V,j)$ as follows. Let $k_2 = LS(U,V,j)$. If $k_2 < \min\{P,j\}$, then by the definition of LS, we have $U[P-k_2] \ne V[j-k_2]$. Let $X[\gamma_j] = (u_{\gamma_j}, \mu_{\gamma_j}) = U[P-k_2]$ and $X[\delta_j] = (v_{\delta_j}, \nu_{\delta_j}) = V[j-k_2]$. When u_{γ_j} is the same as v_{δ_j}, let $HD(U,V,j)$ be $\min\{\mu_{\gamma_j}, \nu_{\delta_j}\}/\nu_{\delta_j}$ and otherwise $HD(U,V,j) = 0$. Then, it is not difficult to see that

$$j - LS(U,V,j) - HD(U,V,j) = \rho(v, j' - LS(u,v,j')) \quad \text{where } j' = \rho^-(V,j). \tag{6}$$

Example 2. Consider $U = (b,1)(d,4)$ and $V = (a,2)(b,3)(d,4)(a,2)(b,2)$. Let $j = 3$ and $i = j+1 = 4$. Then, $k_1 = LP(V,4) = 1, X[\alpha_4] = (v_{\alpha_4}, \nu_{\alpha_4}) = V[1+k_1] = V[2]$, and $X[\beta_4] = (v_{\beta_4}, \nu_{\beta_4}) = V[i+k_1] = V[5]$. Thus, $TL(V,4) = \min\{\nu_{\alpha_4}, \nu_{\beta_4}\}/\nu_{\alpha_4} = \min\{\nu_2, \nu_5\}/\nu_2 = \min\{3,2\}/3 = 2/3$ (see Fig. 2(a)). We can also find that $k_2 = LS(U,V,3) = 1$, $X[\gamma_3] = (u_{\gamma_3}, \mu_{\gamma_3}) = U[P-k_2] = U[2-1] = U[1]$, and $X[\delta_3] = (v_{\delta_3}, \nu_{\delta_3}) = V[j-k_2] = V[3-1] = V[2]$. Hence, $HD(U,V,3) = \min\{\mu_{\gamma_3}, \nu_{\delta_3}\}/\nu_{\delta_3} = \min\{\mu_1, \nu_2\}/\nu_2 = \min\{1,3\}/3 = 1/3$ (see Fig. 2(b)).

We have the analog of Lemma 1 for run-length encoded strings as follows.

Lemma 2. *Let u and v be two nonempty strings over Σ such that the last character of u is different from the first character of v. Let $U = \sigma(u)$, $V = \sigma(v)$, $p = |u|$, and $q = |v|$. Then, there exists a position j' on v with $1 \le j' \le q-1$ such that*

$$LS(u,v,j') + LP(v,j'+1) \ge j', \tag{7}$$

$$LS(u,v,j') > 0 \quad \textit{and} \quad LP(v,j'+1) > 0 \tag{8}$$

if and only if there exists $j = \rho(v,j')$ such that

$$LS(U,V,j) + HD(U,V,j) + LP(V,j+1) + TL(V,j+1) \ge j, \tag{9}$$

$$LS(U,V,j) + HD(U,V,j) > 0 \quad \textit{and} \quad LP(V,j+1) + TL(V,j+1) > 0. \tag{10}$$

Moreover, $V[1\ldots j] = \sigma(v[1\ldots j'])$ and $V[j+1\ldots Q] = \sigma(v[j'+1\ldots q])$.

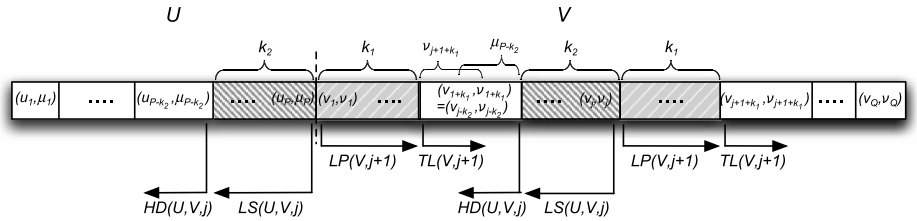


Fig. 3. An illustration for the case where $\min\{\mu_{P-k_2}, \nu_{j-k_2}\} = \mu_{P-k_2}$ and $\min\{\nu_{1+k_1}, \nu_{j+1+k_1}\} = \nu_{j+1+k_1}$.

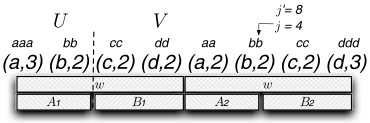


Fig. 4. A square ww of UV centered at V.

Proof. First, observe that Eq. (8) implies $v[j'] = u[p]$ and $v[j'+1] = v[1]$. By the assumption that $u[p] \neq v[1]$, we have $v[j'] \neq v[j'+1]$. That is, $v[j']$ and $v[j'+1]$ are on different runs. Since $j = \rho(v, j')$, we conclude that $V[1 \ldots j]$ and $V[j+1 \ldots Q]$ are the run-length encoded strings of $v[1 \ldots j']$ and $v[j'+1 \ldots q]$, respectively.

We then show that Eqs. (8) and (10) are equivalent. From the identity $LP(V, j+1) + TL(V, j+1) = \rho(v, LP(v, j'+1))$, we establish that $LP(v, j'+1) > 0$ if and only if $LP(V, j+1) + TL(V, j+1) > 0$. From the identity $j - LS(U, V, j) - HD(U, V, j) = \rho(v, j' - LS(u, v, j'))$, we get that $j' - LS(u, v, j') < j'$ if and only if $j - LS(U, V, j) - HD(U, V, j) < j$, which is equivalent to $LS(u, v, j') > 0$ if and only if $LS(U, V, j) + HD(U, V, j) > 0$.

Next, we prove the equivalence of Eqs. (7) and (9). Now assume Eq. (7) holds and $LS(U, V, j) + LP(V, j+1) < j$ (otherwise Eq. (9) holds trivially). In this case, let $k_1 = LP(V, j+1)$ and $k_2 = LS(U, V, j)$. Hence, $X[\alpha_{j+1}] = V[1 + k_1]$, $X[\beta_{j+1}] = V[j+1+k_1]$, $X[\gamma_j] = U[P-k_2]$, and $X[\delta_j] = V[j-k_2]$. Furthermore, $X[\alpha_{j+1}] \neq X[\beta_{j+1}]$ and $X[\gamma_j] \neq X[\delta_j]$. Observe that $X[\alpha_{j+1}]$ and $X[\delta_j]$ must be the same run, otherwise $LS(u, v, j') + LP(v, j'+1) < j'$. Therefore, $LS(U, V, j) + LP(V, j+1) = j - 1$. The part which yet remains to be proved is $HD(U, V, j) + TL(V, j+1) \geq 1$. Note that $X[\gamma_j] = (u_{P-k_2}, \mu_{P-k_2})$, $X[\delta_j] = (v_{j-k_2}, \nu_{j-k_2})$, $X[\alpha_{j+1}] = (v_{1+k_1}, \nu_{1+k_1})$, and $X[\beta_{j+1}] = (v_{j+1+k_1}, \nu_{j+1+k_1})$. Since (v_{1+k_1}, ν_{1+k_1}) and (v_{j-k_2}, ν_{j-k_2}) are the same run, u_{P-k_2}, v_{1+k_1}, and v_{j+1+k_1} are the same character. Summing HD and TL together yields

$$\begin{aligned} HD(U, V, j) + TL(V, j+1) &= \min\{\mu_{\gamma_j}, \nu_{\delta_j}\}/\nu_{\delta_j} + \min\{\nu_{\alpha_{j+1}}, \nu_{\beta_{j+1}}\}/\nu_{\alpha_{j+1}} \\ &= \min\{\mu_{P-k_2}, \nu_{j-k_2}\}/\nu_{j-k_2} + \min\{\nu_{1+k_1}, \nu_{j+1+k_1}\}/\nu_{1+k_1} \\ &= (\min\{\mu_{P-k_2}, \nu_{j-k_2}\} + \min\{\nu_{1+k_1}, \nu_{j+1+k_1}\})/\nu_{1+k_1} \\ &= (\min\{\mu_{P-k_2}, \nu_{1+k_1}\} + \min\{\nu_{1+k_1}, \nu_{j+1+k_1}\})/\nu_{1+k_1} \ . \end{aligned} \tag{11}$$

Now we prove that $\min\{\mu_{P-k_2}, \nu_{1+k_1}\} + \min\{\nu_{1+k_1}, \nu_{j+1+k_1}\} \geq \nu_{1+k_1}$. If $\nu_{1+k_1} \leq \mu_{P-k_2}$ or $\nu_{1+k_1} \leq \nu_{j+1+k_1}$, then the above inequality holds directly. Thus, we only need to consider the case where $\nu_{1+k_1} > \mu_{P-k_2}$ and $\nu_{1+k_1} > \nu_{j+1+k_1}$. Since Eq. (7) holds, $\mu_{P-k_2} \geq \nu_{1+k_1} - \nu_{j+1+k_1}$. Thus,

$$\begin{aligned} \min\{\mu_{P-k_2}, \nu_{1+k_1}\} + \min\{\nu_{1+k_1}, \nu_{j+1+k_1}\} &= \mu_{P-k_2} + \nu_{j+1+k_1} \\ &\geq \nu_{1+k_1} - \nu_{j+1+k_1} + \nu_{j+1+k_1} \\ &= \nu_{1+k_1}. \end{aligned} \tag{12}$$

By combining Eqs. (11) and (12), therefore, $HD(U, V, j) + TL(V, j+1) \geq 1$ (see Fig. 3 for an illustration). The only-if part is proved. Conversely, it follows similarly that Eq. (9) implies Eq. (7) and we omit the proof. Q.E.D.

Let us return to the discussion of Condition D2. We consider Case B. In Fig. 4, a square ww is divided into $A_1B_1A_2B_2$ where A_1 is a suffix of u, B_1 is a prefix of v, $A_1 = A_2$, and $B_1 = B_2$. Main and Lorentz's result (cf. Lemma 1) identifies the end point of A_2 (call it j') such that Eqs. (7) and (8) in Lemma 2 hold. However, we have shown that Eqs. (7) and (8) are equivalent to Eqs. (9) and (10). Therefore, we can replace the testing of Eqs. (7) and (8) by Eqs. (9) and (10). Hence, all squares in Case B can also be identified. By reversing U and V, Case C can be handled similarly. That is, we can apply Lemma 2 to locate the positions of all squares centered at U^R and across V^R and U^R.

For example, let $u = a^3b^2$ and $v = c^2d^2a^2b^2c^2d^3$. We consider $j' = 8$. Then, $LS(u, v, 8) + LP(v, 9) = 4 + 4 \geq 8$ so that Eqs. (7) and (8) hold for $j' = 8$. As for Eqs. (9) and (10), let $j = 4$, and then

$$LS(U, V, 4) + HD(U, V, 4) + LP(V, 5) + TL(V, 5) = 1 + \frac{2}{2} + 1 + \frac{2}{2} \geq 4.$$

The following example shows how fractional values can occur in computation. Let $u = a^2b^2$ and $v = a^4b^2a^3$. We consider $j' = 6$. From Eq. (7), we have $LS(u, v, 6) + LP(v, 7) = 4 + 3 = 7 \geq 6$. And from Eq. (9),

$$LS(U, V, 2) + HD(U, V, 2) + LP(V, 3) + TL(V, 3) = 1 + \frac{2}{4} + 0 + \frac{3}{4} \geq 2$$

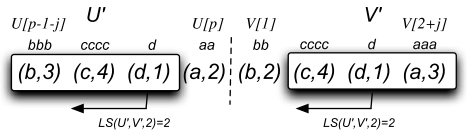


Fig. 5. Testing a square centered at the boundary of U and V.

also holds.

Case A requires to find a suffix of u which is also a prefix of v. This is equivalent to testing whether $LS(u, v, j')$ is equal to j' for some j' with $1 \le j' \le q$. Again, testing whether $LS(U, V, j)$ is equal to j for some $1 \le j \le Q$ is not enough. For example, let $U = (b, 3)(c, 5)(a, 2)$ and $V = (b, 2)(c, 5)(a, 3)$. Then, $LS(U, V, j) \ne j$ for $j = 1, 2, 3$. Lemma 3 gives the correct test. Note that, in Lemma 3, we use $U - U[i]$ to denote the run-length encoded string by removing $U[i]$, i.e., the ith run, from U.

Lemma 3. *Let $U' = U - U[P]$ and $V' = V - V[1]$. There is a square centered at the boundary of $u = \sigma^-(U)$ and $v = \sigma^-(V)$ if and only if there exists j with $0 \le j \le \min\{P - 2, Q - 2\}$ such that*

1. *$LS(U', V', j)$ equals to j;*
2. *$u_{P-1-j} = v_1$, $u_P = v_{2+j}$, $v_1 \le \mu_{P-1-j}$, and $\mu_2 \le v_{2+j}$.*

Proof. (If-part) Let j satisfy $LS(U', V', j) = j$, $u_{P-1-j} = v_1$, $u_P = v_{2+j}$, $v_1 \le \mu_{P-1-j}$, and $\mu_2 \le v_{2+j}$. See Fig. 5 for an illustration. Then, $\sigma^-\left((u_{P-1-j}, v_1)U[P-j]\dots U[P]\right)$ is a suffix of u and $\sigma^-\left(V[1]\dots V[1+j](v_{2+j}, \mu_P)\right)$ is a prefix of v, and these two substrings are equal. Thus, this is a square centered at the boundary of u and v.

(Only-if-part) Suppose there is a square ww centered at the boundary of u and v. Since w is a suffix of u, we can always express w as $\sigma^-\left((u_{P-1-j}, \ell)U[P-j]\dots U[P]\right)$ for some j with $0 \le j \le \min\{P-2, Q-2\}$ and some ℓ with $1 \le \ell \le \mu_{P-1-j}$. Similarly, w is a prefix of v, and thus, it can be expressed as $\sigma^-\left(V[1]\dots V[1+j](v_{2+j}, \ell)\right)$ for the same j and ℓ. Therefore, (u_{P-1-j}, ℓ) must equal to $V[1]$, and thus, $v_1 = \ell \le \mu_{P-1-j}$ and $u_{P-1-j} = v_1$. Similarly, $u_P = v_{2+j}$ and $\mu_P \le v_{2+j}$. Q.E.D.

We summarize the above discussion by Algorithm SQUARES. The initial call is triggered by applying SQUARES(X, 0, N). Since Steps 5 and 6 are based on Lemmas 3 and 2, respectively, we also used the terms defined in these two lemmas. Furthermore, for clarity, the positions of all squares are still expressed as their original positions in an uncompressed string.

Algorithm SQUARES(X', f, K)

Input: A run-length encoded substring $X' = (a'_1, r'_1)(a'_2, r'_2)\cdots(a'_K, r'_K)$ of X. Note that $\sigma^-(X')$ occurs at the position $f + 1$ of $\sigma^-(X)$.
Output: The positions of all squares of $\sigma^-(X')$ with respect to $\sigma^-(X)$.
begin

Step 1. If there are less than four runs in X',
then for each run $X'[i] = (a'_i, r'_i)$ in X' with $r'_i > 1$,
output $([f + 2k, f + r'_i], 2k)|_{k=[1,\lfloor r'_i/2 \rfloor]}$ and return.

Step 2. Break X' evenly into U and V such that $X' = UV$.
Let $R_0 = 0$. For $i = 1$ to K do $R_i = R_{i-1} + r'_i$.

Step 3. SQUARES($U, f, |U|$).

Step 4. SQUARES($V, f + R_{|U|}, |V|$).

Step 5. `// Find the positions of all squares centered at the junction of U and V.`
For $j = 0, 1, \dots, \min\{P-2, Q-2\}$,
if $LS(U', V', j) = j$, $u_{P-1-j} = v_1$, $u_P = v_{2+j}$, $v_1 \le \mu_{P-1-j}$, and $\mu_2 \le v_{2+j}$,
then output $([f + k, f + k], 2(k - R_{|U|}))$ where $k = R_{|U|+j+1} + \mu_P$.

Step 6. `// Find the positions of all squares centered at V and overlapping U.`
For $j = 1, 2, \dots, Q$,
Substep 6.1. Compute $LS(U, V, j)$ for $2 \le j \le Q - 1$ and $LP(V, i)$ for $3 \le i \le Q$.
Substep 6.2. Based on the above two arrays, compute $HD(U, V, j)$ and $TL(V, i)$.
Substep 6.3. If $LS(U, V, j) + HD(U, V, j) > 0$, $LP(V, j + 1) + TL(V, j + 1) > 0$,
and $LS(U, V, j) + HD(U, V, j) + LP(V, j + 1) + TL(V, j + 1) \ge j$, then
output $\left(\left[f + R_{|U|} + j' + \rho^-(V, j - t_2), f + R_{|U|} + j' + \min\{\rho^-(V, t_1), j' - 1\}\right], 2j'\right)$
where $j' = \rho^-(V, j)$, $t_1 = LP(V, j + 1) + TL(V, j + 1)$,
and $t_2 = LS(U, V, j) + HD(U, V, j)$.

Step 7. Reverse U and V. Then, by using a similar process as Step 6, we can find the positions of all squares centered at U and overlapping V.

end

Theorem 1. *The positions of all squares occurring in a run-length encoded string with N runs can be reported in $O(N \log N)$-time by using Algorithm* SQUARES *if one does not explicitly compute all trivial squares occurring within runs.*

Proof. The correctness of this algorithm is described as below. A square occurring in $X' = UV$ can only fall into one of the cases: (1) within a run; (2) entirely in U or V; (3) overlapping with U and V. Squares in the first case are handled by Step 1 of this algorithm. Squares in the second case are recursively reported in Steps 3 and 4. Squares in the third case are

considered in Steps 5, 6, and 7. By Lemmas 2 and 3, the positions of all squares in the third case are also reported. This proves the correctness of Algorithm SQUARES. We remark that the formula for the output of Substep 6.3 can be derived from Eqs. (4)–(6).

The time complexity is analyzed. Step 1 of Algorithm SQUARES takes $O(1)$-time. Step 2 can be done in $O(N)$-time. Let $T(N)$ be the time-complexity for solving SQUARES(X). Therefore, each of Steps 3 and 4 needs $T(N/2)$ time. By using the algorithms in [14,16], both arrays LP and LS can be computed in time $O(|UV|)$. Thus, arrays HD and TL can also be computed in time $O(N)$ after LP and LS are obtained. Thus, Steps 5, 6, and 7 can be done in time $O(N)$. Therefore, the time complexity of Algorithm SQUARES can be obtained by solving $T(N) = T(\lfloor \frac{N}{2} \rfloor) + T(\lceil \frac{N}{2} \rceil) + O(N)$, which is $O(N \log N)$. The optimality also comes from Main and Lorentz's result: assume that all characters in the alphabet can only be distinguished by doing character comparisons. This completes the proof. Q.E.D.

4. Conclusion

Main and Lorentz [14] developed an $O(n \log n)$-time algorithm to locate all occurrences of squares in a string over an unbounded alphabet. In this paper, we successfully applied their idea to run-length encoded strings, and obtained an optimal algorithm in time $O(N \log N)$ where N is the number of runs in a string, provided that we do not explicitly compute at all trivial squares occurring within runs. The time complexity is independent of the lengths of runs, and thus it is beneficial especially when many characters in a string are duplicate in a consecutive way. It is known that all *distinct squares* can be identified in $O(n)$-time [9]. However, we do not know whether all distinct squares in a run-length encoded string can still be reported in $O(N)$-time.

Acknowledgements

We thank the anonymous reviewers for their valuable comments which helped to improve the quality of this paper.

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in: J. Van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990.
[2] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, Theoretical Computer Science 22 (3) (1983) 297–315.
[3] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Information Processing Letters 12 (5) (1981) 244–250.
[4] M. Crochemore, Transducers and repetitions, Theoretical Computer Science 45 (1) (1986) 63–86.
[5] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, Singapore, 2002.
[6] E. Delacourt, J.F. Myoupo, D. Smem, A constant time parallel detection of repetitions, Parallel Processing Letters 9 (1) (1999) 81–92.
[7] T. Garcia, David Semé, A coarse-grained multicomputer algorithm for the detection of repetitions, Information Processing Letters 93 (6) (2005) 307–313.
[8] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[9] D. Gusfield, J. Stoye, Linear time algorithm for finding and representing all the tandem repeats in a string, Journal of Computer and System Sciences 69 (4) (2004) 525–546.
[10] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern-matching in string, SIAM Journal on Computing 6 (2) (1977) 323–350.
[11] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 596–604.
[12] H.F. Leung, Z.S. Peng, H.F. Ting, An efficient algorithm for online square detection, Theoretical Computer Science 363 (1) (2006) 69–75.
[13] M.G. Main, R.J. Lorentz, An $O(n \log n)$ algorithm for recognizing repetitions, Technical Report CS-79-056, Washington State University, 1979.
[14] M.G. Main, R.J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, Journal of Algorithms 5 (3) (1984) 422–432.
[15] M.G. Main, R.J. Lorentz, Linear time recognition of square-free strings, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series, vol. F12, Springer, Berlin, 1985, pp. 271–278.
[16] M.G. Main, Detecting leftmost maximal periodicities, Discrete Applied Mathematics 25 (1–2) (1989) 145–153.
[17] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (1) (2007) 1–61.
[18] K. Sayoood, E. Fow (Eds.), Introduction to Data Compression, second edition, Morgan Kaufmann, 2000.
[19] A.O. Slisenko, Detection of periodicities and string-matching in real time, Journal of Mathematical Sciences 22 (3) (1983) 1316–1387.
[20] J. Stoye, D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, Theoretical Computer Science 270 (1–2) (2002) 843–856.
[21] A. Thue, Uber unendlich Zeichenreihen, Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7 (1906) 1–22. Reprinted in: T. Nagell, A. Selberg, S. Selberg, K. Thalberg (Eds.), Selected Mathematical Papers of Axel Thue. Oslo, Norway, Universitetsforlaget, 1977, pp. 139–158.