Theoretical Computer Science 407 (2008) 554–563

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Efficient on-line repetition detection

Jin-Ju Hong, Gen-Huey Chen *

Department of Computer Science and Information Engineering, National Taiwan University, Taiwan

ARTICLE INFO

Article history:
Received 7 July 2006
Received in revised form 23 December 2007
Accepted 25 August 2008
Communicated by M. Crochemore

Keywords:
Repetition
Square
On-line detection

ABSTRACT

A *repetition* is a nonempty string of the form X^q, where $q \geq 2$. Given a string S character by character and the value of q, the on-line repetition detection problem is to detect and report the first repetition in S, if it exists, in an on-line manner. Leung, Peng and Ting first studied the problem for $q = 2$ and gave an $O(m \log^2 m)$ time algorithm (refer to [H.-F. Leung, Z. Peng, H.-F. Ting, An efficient algorithm for online square detection, Theoretical Computer Science 363 (1) (2006) 69–75]), where m is the ending position of the first repetition in S. In this paper, we improve the above cited work by reducing the time complexity to $O(m \log \beta)$, where β is the number of distinct characters in the first m characters of S. Moreover, we also solve the problem for $q \geq 3$ with the same time complexity.

1. Introduction

A *repetition* is the concatenation of q copies of a nonempty string X, denoted by X^q, where $q \geq 2$. A repetition of the form X^2 is called a *square*. Detecting repetitions in a string is a fundamental problem in many areas such as combinatorics [20,22], automata and formal language theory [13,24], data compression [9], bioinformatics [11,16,23], *etc*. Several algorithms [1–5, 12,17,18,21,25] were proposed to detect repetitions in a string with $O(n \log n)$ time over a general alphabet or with $O(n)$ time over an integer alphabet (e.g., [4,5,12,17,25]), where n is the length of the string. These algorithms are all executed in an off-line manner.

Given a string S character by character and an integer $q \geq 2$, the *on-line repetition detection problem* is to detect the first repetition of the form X^q in S, and report it, if it exists, as soon as the repetition is completely read. The execution halts either when the repetition is reported or when the entire S is read. Leung, Peng and Ting [19] first presented an $O(m \log^2 m)$ time algorithm to solve the problem for $q = 2$, where m is the number of characters of S read (m is the length of S if no square exists in S). Throughout this paper, we use $|S|$ to denote the length of S, $S[i]$ to denote the ith character of S read, $S[i..j]$ $(i \leq j)$ to denote the substring of S from $S[i]$ to $S[j]$, and XY to denote the concatenation of strings X and Y. A string is *square-free* if it does not contain any square.

In [5], Crochemore gave an algorithm that could solve the off-line square detection problem in $O(n \log \alpha)$ time, where n is the length of the string S and α is the number of distinct characters in S. In this paper, we adapt Crochemore's algorithm to an on-line manner. The resulting algorithm runs in $O(m \log \beta)$ time, where m is the number of characters of S read and β is the number of distinct characters in $S[1..m]$. Besides, we also solve the on-line repetition detection problem for $q \geq 3$ in $O(m \log \beta)$ time.

In the next section, we first review Crochemore's algorithm. The on-line repetition detection problems for $q = 2$ and $q \geq 3$ are solved in Sections 3 and 4, respectively.

* Corresponding address: Department of Computer Science and Information Engineering, National Taiwan University, No. 1, Section 4, Roosevelt Road, Taipei 106, Taiwan. Fax: +886 2 23628167.
E-mail addresses: d89010@csie.ntu.edu.tw (J.-J. Hong), ghchen@csie.ntu.edu.tw (G.-H. Chen).

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2008.08.038

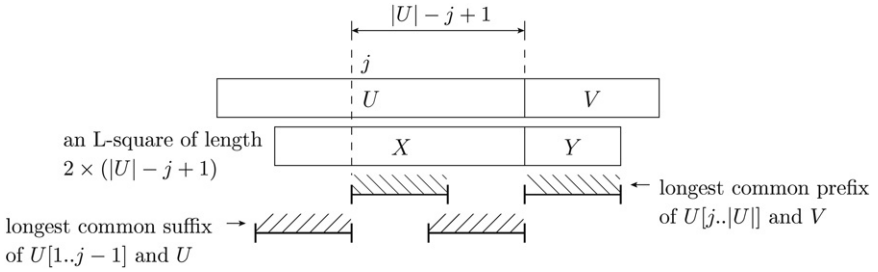

Fig. 1. Computation of left(U, V).

2. Preliminaries

Crochemore's algorithm [5] for the off-line square detection problem first partitions the input string S into nonempty substrings $F_1, F_2, \ldots, F_t$ (i.e., $S = F_1F_2 \cdots F_t$), and then sequentially determines whether or not a square ends in F_k for $k = 1, 2, \ldots, t$. The partition, which is referred to as the *f-factorization* (or *s*-factorization in [4]) of S, is performed as follows. Each F_k is defined as either a single character not in $F_1F_2 \cdots F_{k-1}$ or the longest prefix of $S[b_k..|S|]$ that is also a prefix of $S[j..|S|]$ for some $1 \le j < b_k$, where b_k is the starting position of F_k (i.e., $b_k = |F_1F_2 \cdots F_{k-1}| + 1$). For example, if $S = \mathtt{abaabbba}$, then $(F_1, F_2, F_3, F_4, F_5, F_6) = (\mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{ab}, \mathtt{bb}, \mathtt{a})$. By the aid of Ukkonen's on-line suffix tree construction algorithm [26], the f-factorization of S can be obtained on-line in $O(|S| \log \alpha)$ time, where α is the number of distinct characters in S.

Two functions, named *left* and *right*, are used in Crochemore's algorithm to search for a square. Given two square-free strings U and V, left(U, V) (right(U, V)) searches for a square, named an *L-square* (an *R-square*), which can be written as the concatenation of two nonempty strings X and Y such that X is a suffix of U, Y is a prefix of V, and $|X| \ge |Y|$ ($|X| < |Y|$). The process for determining whether or not a square ends in F_k is performed only when $F_1F_2 \cdots F_{k-1}$ is square-free. It is not difficult to see that there is a square ending in F_k if and only if a square can be found by left(F_{k-1}, F_k) or right(F_{k-1}, F_k) or right$(F_1F_2 \cdots F_{k-2}, F_{k-1}F_k)$. Notice that no square can be found by left$(F_1F_2 \cdots F_{k-2}, F_{k-1}F_k)$, because $S[b_{k-1}..b_k]$ is not contained in $F_1F_2 \cdots F_{k-1}$. Also, if there is a square completely contained in F_k, then there is an occurrence of it starting before b_k.

Given two strings X and Y, let $\ell_p(X, Y)$ and $\ell_s(X, Y)$ be the lengths of their longest common prefix and longest common suffix, respectively. It was shown in [21] that left(U, V) and right(U, V) can be obtained by means of computing $\ell_p(X, Y)$ and $\ell_s(X, Y)$. More concretely, computing left(U, V) is equivalent to searching for a position j in U such that $\ell_p(U[j..|U|], V) \ge 1$ and $\ell_s(U[1..j-1], U) + \ell_p(U[j..|U|], V) \ge |U| - j + 1$, and computing right$(U, V)$ is equivalent to searching for a position j in V such that $\ell_s(U, V[1..j]) \ge 1$, $\ell_p(V[j+1..|V|], V) \ge 1$, and $\ell_s(U, V[1..j]) + \ell_p(V[j+1..|V|], V) \ge j$. The computation of left(U, V) is further illustrated with Fig. 1. There is an L-square of length $2 \times (|U| - j + 1)$ if and only if a position j in U, as described above, can be found.

It should be noted that given a string W and $1 \le i_1 \le i_2 \le |W|$, $\ell_p(W[i_1..i_2], W)$ can be computed in constant time, if an $O(|W|)$ time preprocessing using the Knuth–Morris–Pratt algorithm [15] or the Z-algorithm [11] is made. The preprocessing can be achieved on-line in linear time (see [14]), as explained below.

A string W has a *period* p if $W[i_1] = W[i_2]$ for every pair of i_1 and i_2 satisfying $i_1 \equiv i_2 \pmod p$, where $1 \le p \le |W|$. Let $\pi(W)$ denote the smallest period of W. Characters of W are read in the sequence of $W[1], W[2], \ldots, W[|W|]$. When $W[i]$ is read, it takes $O\big(\pi(W[1..i]) - \pi(W[1..i-1])\big)$ time to compute $\pi(W[1..i])$ and $\ell_p(W[j..i], W[1..i])$ for all $\pi(W[1..i-1]) < j \le \pi(W[1..i])$ and store the computed values in $M_\pi[i]$ and $M_\ell[j]$, where $1 \le i \le |W|$ and M_π, M_ℓ are two arrays. Therefore, the on-line processing for $W[1], W[2], \ldots, W[i]$ can be completed in $O(i)$ time.

After $W[i]$ is processed, each $\pi(W[1..r])$ can be found in $M_\pi[r]$, where $1 \le r \le i \le |W|$. Moreover, each $\ell_p(W[r..i], W[1..i])$ can be determined in constant time as follows:

$$\ell_p(W[r..i], W[1..i]) = \begin{cases} |W[r..i]|, & \text{if } M_\ell[r'] \ge |W[r..i]|; \\ M_\ell[r'], & \text{else,} \end{cases}$$

where $r' \equiv r \pmod{M_\pi[i]}$ and $1 \le r' \le M_\pi[i]$.

The on-line processing above for W is referred to as an *on-line LCP* (*longest common prefix*) *preprocessing* for W. Similarly, there is a linear-time *on-line LCS* (*longest common suffix*) *preprocessing* for W in which characters of W are read in the sequence of $W[|W|], W[|W|-1], \ldots, W[1]$. With this preprocessing, each $\pi(W[r..|W|])$ and each $\ell_s(W[i..r], W[i..|W|])$ can be determined in constant time after $W[i]$ is processed, where $1 \le i \le r \le |W|$.

3. On-line square detection

In this section, we solve the on-line square detection problem in $O(m \log \beta)$ time, where m is the number of characters of S read and β is the number of distinct characters in $S[1..m]$. Characters of S are read in the sequence of $S[1], S[2], \ldots, S[|S|]$. When $S[i]$ is read, we first compute the f-factorization of $S[1..i]$ on-line, where $2 \le i \le |S|$. In the following, assuming

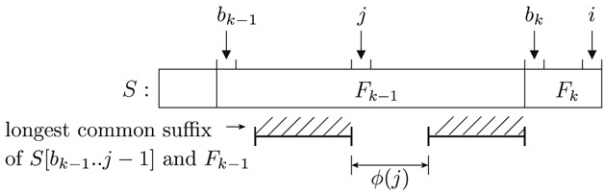

Fig. 2. Computation of $\phi(j)$.

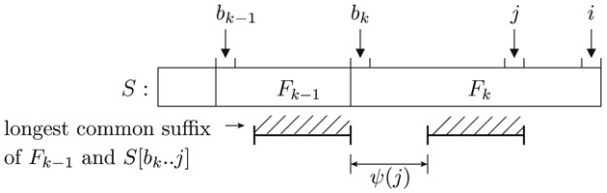

Fig. 3. Computation of $\psi(j)$.

that $S[1..i-1]$ is square-free and $S[i]$ belongs to F_k, we show the on-line processing of left(F_{k-1}, F_k), right(F_{k-1}, F_k) and right$(F_1F_2\cdots F_{k-2}, F_{k-1}F_k)$, in order to determine whether a square ends at i in S or not, where $k \geq 2$.

3.1. On-line processing of left(F_{k-1}, F_k)

Define $\phi(j) = b_k - j - \ell_s(S[b_{k-1}..j-1], F_{k-1})$, where $b_{k-1} < j < b_k$ (refer to Fig. 2). Also, set $\phi(b_{k-1}) = |F_{k-1}|$. It is not difficult to see that there is an L-square ending at i if and only if there exists $b_{k-1} \leq j < b_k$ such that F_k $(= S[b_k..i])$ is a prefix of $S[j..b_k-1]$ and $|F_k| \geq \phi(j)$.

To facilitate the computation of left(F_{k-1}, F_k), the following preprocessing is performed when $i = b_k$. First, $\phi(j)$'s (and $\ell_s(S[b_{k-1}..j-1], F_{k-1})$'s) for all $b_{k-1} < j < b_k$ are computed, which takes total $O(|F_{k-1}|)$ time. Second, a suffix tree, denoted by T_1, of $F_{k-1}\$$ is built with $O(|F_{k-1}|\log\alpha)$ time, where \$ is a character not in S and α is the number of distinct characters in F_{k-1}. Each leaf of T_1 corresponds to a suffix of $F_{k-1}\$$. For each internal node w of T_1, let $\widetilde{\phi}_w = \min\{\phi(j) : b_{k-1} \leq j < b_k$ and $S[j..b_k-1]\$$ corresponds to a leaf in the subtree rooted at $w\}$, and let j_w be one of those j's satisfying $\phi(j) = \widetilde{\phi}_w$. In this paper, unless mentioned particularly, whenever a node of a suffix tree is referred to, it means an explicit node. Third, $\widetilde{\phi}_w$ and j_w for all internal nodes w of T_1 are computed (and stored at w) with $O(|F_{k-1}|)$ time in a bottom-up traversal of T_1. We have the following lemma.

Lemma 3.1. *The preprocessing for computing* left(F_{k-1}, F_k) *takes* $O(|F_{k-1}|\log\alpha)$ *time, where* α *is the number of distinct characters in* F_{k-1}.

The *path label* of a (explicit or implicit) node v in T_1 is the concatenation of the edge labels of the path from the root to v. For a substring X of $F_{k-1}\$$, let $v(X)$ denote the node of T_1 whose path label has a prefix X and is shortest. The processing of left(F_{k-1}, F_k) first determines whether or not F_k is a substring of F_{k-1} by traversing T_1 to find the (explicit or implicit) node whose path label is F_k. If the traversal fails, then F_k is not a substring of F_{k-1} and the processing stops. Otherwise, there is a square of length $2 \times (|F_{k-1}| - j_u + 1)$ ending at i provided $|F_k| \geq \widetilde{\phi}_u$, where $u = v(F_k)$. The correctness is explained below.

Clearly, when the traversal succeeds, F_k is a substring of F_{k-1}. Moreover, F_k is a prefix of $S[j..b_k-1]$ if and only if a leaf of T_u corresponds to $S[j..b_k-1]\$$, where $b_{k-1} \leq j < b_k$ and T_u is the subtree of T_1 rooted at u. Hence, there is an L-square ending at i if and only if there exists $b_{k-1} \leq j < b_k$ such that a leaf of T_u corresponds to $S[j..b_k-1]\$$ and $|F_k| \geq \phi(j)$. Further, according to the definition of $\widetilde{\phi}_u$, there is an L-square ending at i if and only if $|F_k| \geq \widetilde{\phi}_u$.

The time complexity is analyzed as follows. The traversal for processing $S[b_k], S[b_k+1], \ldots, S[i]$ takes total $O(|F_k|\log\alpha)$ time, where α is the number of distinct characters in F_{k-1}. Both determining $v(F_k)$ and determining whether $|F_k| \geq \widetilde{\phi}_u$ or not take constant time. Therefore, we have the following lemma.

Lemma 3.2. *With the preprocessing, computing* left(F_{k-1}, F_k)*, which involves processing* $S[b_k], S[b_k+1], \ldots, S[i]$*, takes total* $O(|F_k|\log\alpha)$ *time, where* α *is the number of distinct characters in* F_{k-1}.

3.2. On-line processing of right(F_{k-1}, F_k)

Define $\psi(j) = j - b_k + 1 - \ell_s(F_{k-1}, S[b_k..j])$, where $b_k < j < i$ (refer to Fig. 3). Since $S[1..i-1]$ is square-free, it is not difficult to see that there is an R-square ending at i if and only if there exists $b_k < j < i$ such that $S[j+1..i]$ is a prefix of $S[b_k..j-1]$ and $i - j = \psi(j)$.

Define $K_r = \{j : b_k < j < r$ and $r - j = \psi(j)\}$, where $b_k < r < i + |F_k|$. Initially, each K_r is set to empty. Then, when $S[j]$ is read, $K_{j+\psi(j)}$ is updated with $K_{j+\psi(j)} \cup \{j\}$, where $b_k < j \leq i$. Since K_i is available as $S[i]$ is read, there is an R-square ending at i if and only if there exists $j \in K_i$ such that $S[j+1..i]$ is a prefix of $S[b_k..j-1]$ (i.e., $\ell_p(S[j+1..i], S[b_k..j-1]) = i - j$).

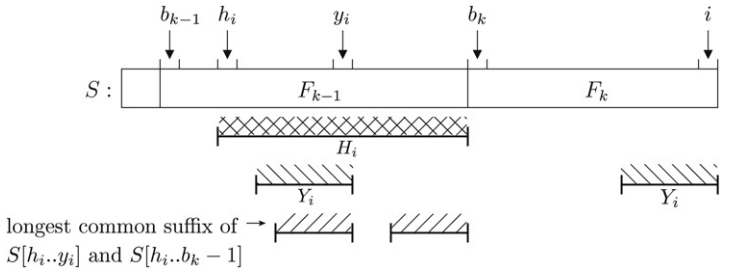

Fig. 4. Computation of $\ell_s(F_{k-1}, S[b_k..i])$.

In addition, $\ell_s(F_{k-1}, S[b_k..i])$ (and $\psi(i)$) can be computed as follows. Let H_i be the suffix of F_{k-1} whose length is $\min\{|F_{k-1}|, |F_k|\}$, i.e., $H_i = S[h_i..b_k - 1]$, where h_i $(= b_k - \min\{|F_{k-1}|, |F_k|\})$ is the starting position of H_i in S. Also, let T_2 be the suffix tree of $H_i\$$, which is built when $S[b_k]$ is read and updated when $S[b_k + 1], S[b_k + 2], \ldots, S[i]$ are read. We use Y_i to denote the longest suffix of $S[b_k..i]$ that is also a substring of H_i. The string Y_i can be determined by traversing T_2 to find the (explicit or implicit) node, denoted by c_i, whose path label is Y_i. The traversing starts from the root when $i = b_k$, and starts from c_{i-1} when $i > b_k$. If Y_i is empty, then $\ell_s(F_{k-1}, S[b_k..i]) = 0$. Otherwise, by the aid of c_i, we can find a position y_i in F_{k-1} such that Y_i is a suffix of $S[h_i..y_i]$. Observe Fig. 4, and it is not difficult to see that $\ell_s(F_{k-1}, S[b_k..i]) = \min\{|Y_i|, \ell_s(S[h_i..y_i], S[h_i..b_k - 1])\}$.

The time complexity is analyzed as follows. Building T_2 requires $O(|F_k| \log \alpha)$ time [27], where α is the number of distinct characters in F_{k-1}. It takes total $O(|F_k| \log \alpha)$ time to find nodes $c_{b_k}, c_{b_k+1}, \ldots, c_i$ when reading characters of F_k. By the aid of the linear-time on-line LCS preprocessing for $S[h_i..b_k - 1]$, $\ell_s(S[h_i..y_i], S[h_i..b_k - 1])$ can be computed in constant time. The set K_i is implemented with a linked list, and so updating $K_{i+\psi(i)}$ takes constant time and reporting all elements of K_i takes $O(|K_i|)$ time. By the aid of the linear-time on-line LCP preprocessing for $S[b_k..i]$, determining whether $\ell_p(S[j + 1..i], S[b_k..j]) = i - j$ or not for each $j \in K_i$ can be achieved in constant time. Notice that $|K_{b_k}| + |K_{b_k+1}| + \cdots + |K_i| < |F_k|$. We have the following lemma.

Lemma 3.3. *Computing* right(F_{k-1}, F_k) *takes total* $O(|F_k| \log \alpha)$ *time, where* α *is the number of distinct characters in* F_{k-1}.

3.3. On-line processing of right$(F_1F_2 \cdots F_{k-2}, F_{k-1}F_k)$

Computing right$(F_1F_2 \cdots F_{k-2}, F_{k-1}F_k)$ is similar to computing right(F_{k-1}, F_k). We need to additionally process characters in F_{k-1} when $S[b_k]$ is read. Hence, we have the following lemma.

Lemma 3.4. *Computing* right$(F_1F_2 \cdots F_{k-2}, F_{k-1}F_k)$ *takes total* $O(|F_{k-1}F_k| \log \alpha)$ *time, where* α *is the number of distinct characters in* $F_1F_2 \cdots F_{k-2}$.

3.4. Time complexity

Suppose that the execution halts after reading $S[m]$, i.e., there is a square ending at m or $m = |S|$, and $S[m]$ belongs to F_s, where $s \geq 2$. As a consequence of Lemmas 3.1–3.4, the total execution time is bounded above by $\sum_{2 \leq k \leq s} O(|F_{k-1}F_k| \log \beta) = O(m \log \beta)$, where β is the number of distinct characters in $S[1..m]$. The following theorem summarizes the result of this section.

Theorem 3.5. *The on-line square detection problem can be solved in* $O(m \log \beta)$ *time, where* m *is the number of characters read and* β *is the number of distinct characters among them.*

4. On-line repetition detection

In this section, we further solve the on-line repetition detection problem for $q \geq 3$. Characters of S are read in the sequence of $S[1], S[2], \ldots, S[|S|]$. When $S[i]$ is read, we first compute the f-factorization of $S[1..i]$, which is all the same as the situation of $q = 2$, and then determine whether or not a repetition X^q ends at i in S. A string is *primitive* if it is not a repetition. A repetition X^q is called a *q-repetition* if X is primitive. For our purpose, we only need to determine whether or not a q-repetition ends at i in S.

Throughout this section, we assume that $S[1..i - 1]$ contains no q-repetition and $S[i]$ belongs to F_k, where $i \geq b_k \geq 2$. According to the definition of the f-factorization, if a q-repetition is completely contained in F_k, then there is an occurrence of it ending before i in S. Consequently, we only need to determine whether or not there is a q-repetition that starts before b_k and ends at i in S.

Suppose that $S[r..i]$ is a q-repetition, where $1 \leq r < b_k$. It falls into one of the following four types:

- Type-A, if $b_{k-1} \leq r < b_k$ and $|F_k| \leq \pi(S[r..i])$;

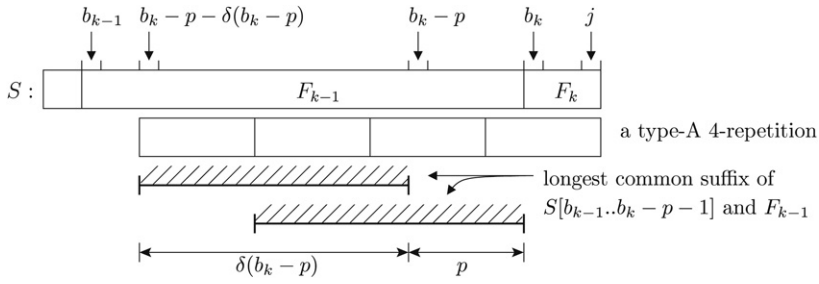

Fig. 5. A type-A 4-repetition.

- Type-B, if $b_{k-1} \le r < b_k$ and $\pi(S[r..i]) < |F_k| < 2 \times \pi(S[r..i])$;
- Type-C, if $b_{k-1} \le r < b_k$ and $|F_k| \ge 2 \times \pi(S[r..i])$;
- Type-D, if $r < b_{k-1}$.

Detecting a q-repetition can be performed according to the four types. Moreover, when $S[b_k]$ is read, the following preprocessing is performed, in order to facilitate the detection of a type-A or type-B q-repetition.

4.1. Preprocessing

Define $\delta(x) = \ell_s(S[b_{k-1}..x-1], F_{k-1})$, where $x > b_{k-1}$. If there is a type-A q-repetition of length $p \times q$ ending at j, then the following two conditions hold:

(C1) $j - b_k + 1 + \delta(b_k - p) + p = p \times q$;
(C2) $j - b_k + 1 \le p$.

Refer to Fig. 5, where a type-A 4-repetition ending at j in S is illustrated. For $b_k \le j < b_k + |F_{k-1}|$, define

$$P_j^A = \{p : p \text{ satisfies (C1) and (C2)}\}.$$

Similarly, if there is a type-B q-repetition of length $p \times q$ ending at j, then (C1) and the following condition hold:

(C2′) $p < j - b_k + 1 < 2p$.

For $b_k \le j < b_k + 2 \times |F_{k-1}|$, define

$$P_j^B = \{p : p \text{ satisfies (C1) and (C2′)}\}.$$

Clearly, we have $p < |F_{k-1}|$ for each $p \in P_j^A$ or $p \in P_j^B$. The objective of the preprocessing is to compute P_j^A and P_j^B as follows.

Step P1: Compute $\delta(x)$ for all $b_{k-1} < x \le b_k$.
Step P2: Set P_j^A empty for all $b_k \le j < b_k + |F_{k-1}|$, and set P_j^B empty for all $b_k \le j < b_k + 2 \times |F_{k-1}|$.
Step P3: For each integer $1 \le p < |F_{k-1}|$, let $j = b_k - 1 - \delta(b_k - p) + p \times (q-1)$ and insert p into P_j^A $\left(P_j^B\right)$ if $j - b_k + 1 \le p$ $\left(p < j - b_k + 1 < 2p\right)$.

Therefore, we have the following lemma.

Lemma 4.1. *The preprocessing, which constructs P_j^A for all $b_k \le j < b_k + |F_{k-1}|$ and P_j^B for all $b_k \le j < b_k + 2 \times |F_{k-1}|$, takes $O(|F_{k-1}|)$ time.*

4.2. Detection of a type-A q-repetition

We first show the following lemma.

Lemma 4.2. *$S[i - p \times q + 1..i]$ is a type-A q-repetition if and only if $p \in P_i^A$ and $F_k = S[b_k - p..i - p]$.*

Proof. ($\Rightarrow$) It holds as an immediate consequence of Section 4.1.

($\Leftarrow$) As a consequence of $p \in P_i^A$, we have

$$i - b_k + 1 \le p;$$
$$i = b_k - 1 - \delta(b_k - p) + p \times (q-1),$$

from which $\delta(b_k - p) \ge p \times (q-2) > 0$ is derived. It follows that $S[r..b_k - p - 1] = S[r + p..b_k - 1]$, where $r = b_k - p - \delta(b_k - p)$ (also refer to Fig. 5). Since $F_k = S[b_k - p..i - p]$, we have $S[r..i - p] = S[r + p..i]$, which means that p is a period of $S[r..i]$. Further, since $i = r - 1 + p \times q$ (i.e., $|S[r..i]| = p \times q$), $S[1..i-1]$ contains no q-repetition, and $|F_k| \le p$, $S[r..i]$ is a type-A q-repetition. □

According to Lemma 4.2, detecting whether or not there is a type-A q-repetition ending at i in S can be accomplished as follows.

Step A1: Decide whether or not $F_k = S[b_k - p..i - p]$ for some $p \in P_i^A$.

If $F_k = S[b_k - p..i - p]$ for some $p \in P_i^A$, then $S[i - p \times q + 1..i]$ is a type-A q-repetition. The following six lemmas are crucial to the analysis of the time requirement.

Lemma 4.3 (*[10]*). *Suppose that p_1 and p_2 are two periods of a string W. If $|W| \geq p_1 + p_2 - \gcd(p_1, p_2)$, then $\gcd(p_1, p_2)$ is also a period of W.*

Lemma 4.4. *Suppose that W' is a substring of W. If $|W'| \geq 2 \times \pi(W)$, then $\pi(W') = \pi(W)$.*

Proof. Let $p = \pi(W)$ and $p' = \pi(W')$. Suppose that $|W'| \geq 2p$. Then, p is also a period of W'. So, $p' \leq p$. If $p' < p$, then $|W'| \geq 2p > p + p' - \gcd(p, p')$. By Lemma 4.3, $\gcd(p, p')$ is also a period of W', which implies that $p' = \gcd(p, p')$ and p' divides p. Without loss of generality, assume $W' = W[x..y]$. For any $1 \leq i_1 \leq i_2 \leq |W|$ with $i_1 \equiv i_2 \pmod{p}$, there are $x \leq i'_1 \leq i'_2 \leq y$ such that $i'_1 \equiv i_1 \pmod{p}$ and $i'_2 \equiv i_2 \pmod{p}$. Now that p' divides p, we have $i'_1 \equiv i'_2 \pmod{p'}$, which implies $W[i_1] = W[i'_1] = W[i'_2] = W[i_2]$. Hence, p' is also a period of W, a contradiction. Therefore, $p' = p$. □

Lemma 4.5 (*[7]*). *Suppose that W is a string and $1 \leq p_1 < p_2 < p_3 \leq \frac{1}{2} \times |W|$. If $\pi(W[1..2p_1]) = p_1$, $\pi(W[1..2p_2]) = p_2$ and $\pi(W[1..2p_3]) = p_3$, then $p_1 + p_2 \leq p_3$.*

Lemma 4.6. *Suppose that $\pi(W[1..2p_j]) = p_j$ for all $1 \leq j \leq t$, where $1 \leq p_1 < p_2 < \cdots < p_t \leq \frac{1}{2} \times |W|$. Then, $p_1 + p_2 + \cdots + p_t < \frac{3}{2} \times |W|$.*

Proof. Clearly, this lemma holds for $t \leq 3$. When $t \geq 4$,

$$\begin{aligned} 2 \times \sum_{1 \leq j \leq t} p_j &= p_1 + \left(\sum_{1 \leq j \leq t-2} (p_j + p_{j+1}) \right) + p_{t-1} + 2p_t \\ &\leq p_1 + \left(\sum_{1 \leq j \leq t-2} p_{j+2} \right) + p_{t-1} + 2p_t \quad \text{(by Lemma 4.5)} \\ &= \left(\sum_{1 \leq j \leq t} p_j \right) - p_2 + p_{t-1} + 2p_t. \end{aligned}$$

Therefore, $\sum_{1 \leq j \leq t} p_j \leq 2p_t + p_{t-1} - p_2 < 3p_t \leq \frac{3}{2} \times |W|$. □

The following lemma can be proved similarly.

Lemma 4.7. *Suppose that $\pi(W[|W| - 2p_j + 1..|W|]) = p_j$ for all $1 \leq j \leq t$, where $1 \leq p_1 < p_2 < \cdots < p_t \leq \frac{1}{2} \times |W|$. Then, $p_1 + p_2 + \cdots + p_t < \frac{3}{2} \times |W|$.*

Lemma 4.8. $\sum_{b_k \leq j \leq i} \sum_{p \in P_j^A} p < \frac{3}{2} \times |F_{k-1}|$.

Proof. For each $p \in P_j^A$, we have $\delta(b_k - p) \geq p \times (q - 2)$ (refer to the proof of Lemma 4.2). Besides, p is a period of $S[r..b_k - 1]$, where $r = b_k - p - \delta(b_k - p)$. Now that $S[b_k - 2p..b_k - 1]$ is a substring of $S[r..b_k - 1]$ and $|S[b_k - 2p..b_k - 1]| = 2p \geq 2 \times \pi(S[r..b_k - 1])$, we have $\pi(S[b_k - 2p..b_k - 1]) = \pi(S[r..b_k - 1])$ by Lemma 4.4. Moreover, we have $|S[r..b_k - 1]| = \delta(b_k - p) + p \geq p \times (q - 1)$. If $\pi(S[r..b_k - 1]) < p$, then by Lemma 4.3, $\pi(S[r..b_k - 1])$ can divide p, which means that there is a q-repetition of length $\pi(S[r..b_k - 1]) \times q$ in $S[r..b_k - 1]$. This contradicts the assumption that $S[1..i - 1]$ contains no q-repetition. Consequently, $\pi(S[b_k - 2p..b_k - 1]) = \pi(S[r..b_k - 1]) = p$. Then, according to Lemma 4.7, we have $\sum_{b_k \leq j \leq i} \sum_{p \in P_j^A} p < \frac{3}{2} \times |F_{k-1}|$. □

Next, the time complexity is analyzed as follows. Since Step A1 performs at most $|F_k|$ ($\leq p$) character comparisons for each $p \in P_i^A$, detecting a type-A q-repetition whose ending character belongs to F_k requires at most $\sum_{b_k \leq j \leq i} \sum_{p \in P_j^A} |S[b_k..j]| \leq \sum_{b_k \leq j \leq i} \sum_{p \in P_j^A} p$ character comparisons, which is bounded above by $\frac{3}{2} \times |F_{k-1}|$ according to Lemma 4.8. Therefore, we have the following lemma.

Lemma 4.9. *Detecting a type-A q-repetition whose ending character belongs to F_k takes total $O(|F_{k-1}|)$ time.*

4.3. Detection of a type-B q-repetition

The following lemma can be proved similar to Lemma 4.2.

Lemma 4.10. *$S[i - p \times q + 1..i]$ is a type-B q-repetition if and only if $p \in P_i^B$ and $F_k = S[b_k - p..i - p]$.*

For each $p \in P_i^B$, we have $p < |F_k| < 2p$. Refer to Fig. 6, where a type-B 4-repetition ending at i in S is illustrated. It is not difficult to see that $F_k = S[b_k - p..i - p]$ holds if and only if the following two conditions hold:

(C1) $\ell_p(S[b_k + p..i], F_k) = |F_k| - p$;
(C2) $S[b_k - p..b_k - 1] = S[b_k..b_k + p - 1]$.

By the aid of the linear-time on-line LCP preprocessing for $S[b_k..i]$, it takes constant time to compute $\ell_p(S[b_k + p..i], F_k)$. Hence, whether (C1) holds or not can be decided in constant time. On the other hand, (C2) requires that $S[b_k..b_k + p - 1]$,

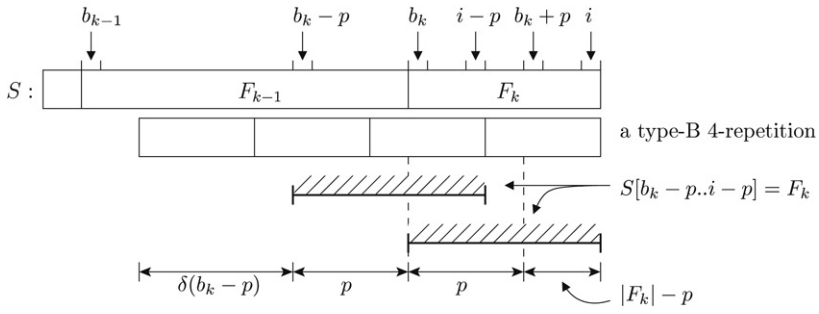

Fig. 6. A type-B 4-repetition.

which is a prefix of F_k, is a suffix of F_{k-1}. Let f_j be the ending position of the leftmost occurrence of $S[b_k..j]$ in F_{k-1}, i.e., $f_j = \min\{x : S[x + b_k - j..x] = S[b_k..j]$ and $b_{k-1} + j - b_k \le x < b_k\}$, where $b_k \le j \le i$. If $S[b_k..j]$ does not occur in F_{k-1}, then set $f_j = b_k$. It is not difficult to see that $S[b_k..j]$ is a suffix of F_{k-1} if and only if $f_j < b_k$ and $\ell_s(S[b_{k-1}..f_j], F_{k-1}) \ge |S[b_k..j]|$. Hence, (C2) holds if and only if the following condition holds:

(C2′) $f_{b_k+p-1} < b_k$ and $\ell_s(S[b_{k-1}..f_{b_k+p-1}], F_{k-1}) \ge p$.

According to Lemma 4.10 and the discussion above, detecting whether or not there is a type-B q-repetition ending at i in S can be accomplished as follows.

Step B1: Compute f_i.
Step B2: Decide whether or not (C1) and (C2′) hold for some $p \in P_i^B$.

If (C1) and (C2′) hold for some $p \in P_i^B$, then $S[i - p \times q + 1..i]$ is a type-B q-repetition. The time complexity of detecting a type-B q-repetition is analyzed below.

When $i = b_k$, computing f_i requires at most $f_i - b_{k-1} + 1$ character comparisons. When $i > b_k$ and $f_{i-1} = b_k$, we set $f_i = b_k$. When $i > b_k$ and $f_{i-1} < b_k$, we have f_i equal to the ending position of the leftmost occurrence of $S[b_k..i]$ in $S[e_{i-1}..b_k - 1]$ ($f_i = b_k$ if $S[b_k..i]$ does not occur in $S[e_{i-1}..b_k - 1]$). The latter can be determined with at most $2f_i - 2f_{i-1} - 1$ character comparisons, if the Knuth–Morris–Pratt algorithm [15] is applied, as detailed below.

For simplicity, let $X = S[b_k..i]$ and $Y = S[e_{i-1}..b_k - 1]$. The Knuth–Morris–Pratt algorithm consists of two phases: a preprocessing phase and a searching phase. The preprocessing phase constructs a table M of $|X|$ entries, where $M[\ell]$ ($2 \le \ell \le |X|$) indicates the length of the longest proper suffix of $X[1..\ell - 1]$ that is also a prefix of $X[1..\ell - 1]$. Then, the searching phase determines the leftmost occurrence of X in Y with at most $2d - |X|$ character comparisons, where $d = f_i - e_{i-1} + 1$ if X occurs in Y, and $d = f_i - e_{i-1}$ if X does not occur in Y.

In fact, it is not necessary for us to perform the preprocessing phase. Notice that each of $\pi(S[b_k..b_k])$, $\pi(S[b_k..b_k + 1])$, …, $\pi(S[b_k..i])$ can be determined in constant time, by the aid of the linear-time on-line LCP preprocessing for $S[b_k..i]$. Consequently, each entry of M can be determined in constant time. Besides, $|X| - 1$ character comparisons can be saved in the searching phase, as a consequence of $X[1..|X| - 1] = S[b_k..i - 1] = S[e_{i-1}..f_{i-1}]$. Therefore, the number of character comparisons needed for computing f_i is at most $2d - 2 \times |X| + 1 \le 2 \times (f_i - e_{i-1} + 1) - 2 \times (i - b_k + 1) + 1 = 2f_i - 2f_{i-1} - 1$, where $e_{i-1} = b_k + f_{i-1} - i + 1$.

On the other hand, since f_{b_k+p-1} and $\ell_s(S[b_{k-1}..f_{b_k+p-1}], F_{k-1})$ (computed by Step P1) are available when $S[i]$ is read, whether (C2′) holds or not can be decided in constant time. Whether (C1) holds or not can be decided in constant time as well. Consequently, Step B2 takes constant time for each $p \in P_i^B$.

The overall time complexity to compute $f_{b_k}, f_{b_k+1}, \ldots, f_i$ is bounded by

$$O(i - b_k + 1) + O(f_{b_k} - b_{k-1} + 1) + O\left(\sum_{b_k < j \le i} (2f_j - 2f_{j-1} - 1)\right) \le O(|F_k|) + O(f_i - b_{k-1}) \le O(|F_k|) + O(|F_{k-1}|),$$

where $O(i - b_k + 1)$ time is required by the on-line LCP preprocessing for F_k ($= S[b_k..i]$). On the other hand, since there are at most $|F_{k-1}| - 1$ integers contained in $P_{b_k}^B, P_{b_k+1}^B, \ldots, P_i^B$ (refer to Step P3), deciding whether or not (C1) and (C2′) hold for all integers in $P_{b_k}^B, P_{b_k+1}^B, \ldots, P_i^B$ requires $O(|F_{k-1}|)$ time. Therefore, we have the following lemma.

Lemma 4.11. *Detecting a type-B q-repetition whose ending character belongs to F_k takes total $O(|F_{k-1}F_k|)$ time.*

4.4. Detection of a type-C q-repetition

If $S[r..i]$ is a type-C q-repetition, then $|F_k| \ge 2 \times \pi(S[r..i])$. By Lemma 4.4, we have $\pi(S[r..i]) = \pi(F_k)$. Let $p = \pi(F_k)$. We first prove the following lemma.

Lemma 4.12. *$S[i - p \times q + 1..i]$ is a type-C q-repetition if and only if $|F_k| \ge 2p$ and $\delta(b_k + p) = p \times q - |F_k|$.*

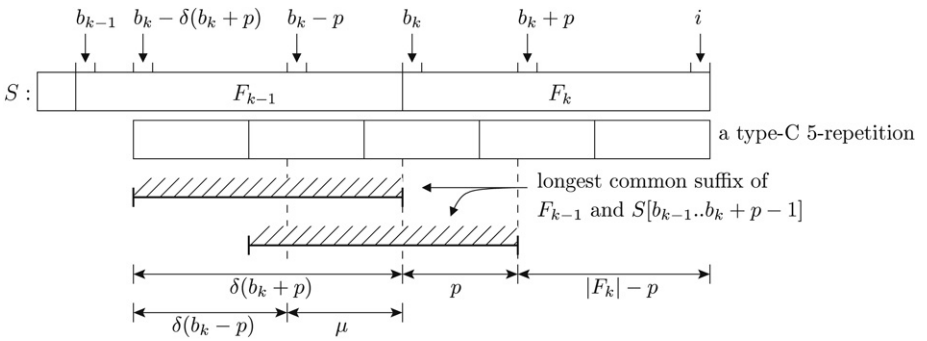

Fig. 7. A type-C 5-repetition, where $\mu = p$.

Proof. ($\Rightarrow$) $|F_k| \geq 2p$ and $\delta(b_k + p) \geq p \times q - |F_k|$ hold as an immediate consequence of a type-C q-repetition (refer to Fig. 7). If $\delta(b_k + p) > p \times q - |F_k|$, then $S[i - p \times q..i - 1]$ is a q-repetition, a contradiction.

($\Leftarrow$) Clearly, p is a period of $S[b_k - \delta(b_k + p)..i]$, a string of length $p \times q$. It is implied that $S[i - p \times q + 1..i]$ is a type-C q-repetition, because $|F_k| \geq 2p$. □

According to Lemma 4.12, detecting a type-C q-repetition can be accomplished as follows.

Step C1: If $|F_k| = 2p$, then compute $\delta(b_k + p)$.
Step C2: Decide whether or not $|F_k| \geq 2p$ and $\delta(b_k + p) = p \times q - |F_k|$ hold.

If $|F_k| \geq 2p$ and $\delta(b_k + p) = p \times q - |F_k|$ hold, then $S[i - p \times q + 1..i]$ is a type-C q-repetition. The time complexity of detecting a type-C q-repetition is analyzed below.

Step C1 is executed whenever $|F_k| = 2p$ holds, and its computed value (i.e., $\delta(b_k + p)$) is to be used in Step C2. By the aid of the linear-time on-line LCP preprocessing for $S[b_k..i]$, the value of p can be determined in constant time. Let $\mu = \ell_s(S[b_k - p..b_k - 1], S[b_k..b_k + p - 1])$, which can be determined with at most p character comparisons. It takes constant time to calculate $\delta(b_k + p)$ as follows:

$$\begin{aligned}
&\text{when } \mu \geq |F_{k-1}|, \quad \delta(b_k + p) = |F_{k-1}|;\\
&\text{when } \mu < |F_{k-1}| \text{ and } \mu < p, \quad \delta(b_k + p) = \mu;\\
&\text{when } \mu < |F_{k-1}| \text{ and } \mu = p, \quad \delta(b_k + p) = \delta(b_k - p) + \mu.
\end{aligned}$$

If $\mu \geq |F_{k-1}|$, then $b_k - p \leq b_{k-1}$ and hence $\delta(b_k + p) = |F_{k-1}|$. If $\mu < |F_{k-1}|$ and $\mu < p$, then $S[(b_k - 1) - \mu] \neq S[(b_k + p - 1) - \mu]$, which implies $\delta(b_k + p) = \mu$. If $\mu < |F_{k-1}|$ and $\mu = p$, then $\delta(b_k + p) = \delta(b_k - p) + \mu$ (refer to Fig. 7 again), where $\delta(b_k - p)$ was calculated by Step P1.

According to Lemma 4.6, the total number of character comparisons needed for Step C1 to calculate $\delta\big(b_k + \pi(S[b_k..j])\big)$'s for all $b_k \leq j \leq i$ with $|S[b_k..j]| = 2 \times \pi(S[b_k..j])$ is bounded above by $\frac{3}{2} \times |S[b_k..i]|$. On the other hand, let $i' = b_k + 2p - 1$ and $p' = \pi(S[b_k..i'])$. By Lemma 4.4, we have $p' = p$, and hence $\delta(b_k + p) = \delta(b_k + p')$. Since $\delta(b_k + p')$ was computed by Step C1 when $S[i']$ was read, Step C2 can be completed in constant time. The execution of Step C2 for $S[b_k]$, $S[b_k + 1]$, ..., $S[i]$ takes total $O(|S[b_k..i]|)$ time. Therefore, we have the following lemma.

Lemma 4.13. *Detecting a type-C q-repetition whose ending character belongs to F_k takes total $O(|F_k|)$ time.*

4.5. Detection of a type-D q-repetition

Since no type-D q-repetition ends in F_1 and F_2, we assume $k \geq 3$. If $S[r..i]$ is a type-D q-repetition, then $b_{k-1} - r < \pi(S[r..i])$ (i.e., $|F_{k-1}F_k| > \pi(S[r..i]) \times (q - 1)$), for otherwise $(b_{k-1} - r \geq \pi(S[r..i]))$, $S[b_{k-1}..b_k] = S[b_{k-1} - \pi(S[r..i])..b_k - \pi(S[r..i])]$, which contradicts the definition of F_{k-1}. By Lemma 4.4, we have $\pi(S[r..i]) = \pi(F_{k-1}F_k)$.

Detecting a type-D q-repetition is similar to detecting a type-C q-repetition. Define $\widetilde{\delta}(x) = \ell_s(S[1..x - 1], F_1F_2 \cdots F_{k-2})$ and $\sigma(x) = \min\{\widetilde{\delta}(x), |S[b_{k-1}..x - 1]|\}$, where $x > b_{k-1}$. Also let $p = \pi(F_{k-1}F_k)$. The following lemma can be proved similar to Lemma 4.12.

Lemma 4.14. *$S[i - p \times q + 1..i]$ is a type-D q-repetition if and only if $|F_{k-1}F_k| > p \times (q - 1)$ and $\sigma(b_{k-1} + p) = p \times q - |F_{k-1}F_k|$.*

Refer to Fig. 8, where $\sigma(b_{k-1} + p)$ is illustrated. According to Lemma 4.14, detecting a type-D q-repetition can be accomplished as follows.

Step D1: If $i = b_k$, then compute $\sigma\big(b_{k-1} + \pi(S[b_{k-1}..j])\big)$ for all $b_{k-1} < j \leq b_k$ with $|S[b_{k-1}..j]| = 2 \times \pi(S[b_{k-1}..j])$.
Step D2: If $i > b_k$ and $|F_{k-1}F_k| = 2p$, then compute $\sigma(b_{k-1} + p)$.
Step D3: Decide whether or not $|F_{k-1}F_k| > p \times (q - 1)$ and $\sigma(b_{k-1} + p) = p \times q - |F_{k-1}F_k|$ hold.

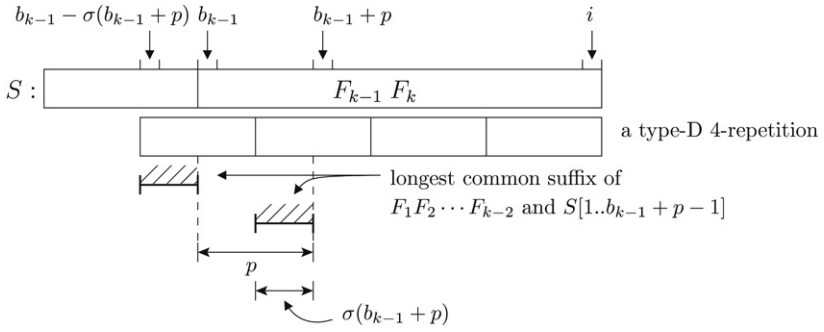

Fig. 8. A type-D 4-repetition.

If $|F_{k-1}F_k| > p \times (q-1)$ and $\sigma(b_{k-1}+p) = p \times q - |F_{k-1}F_k|$ hold, then $S[i-p\times q+1..i]$ is a type-D q-repetition. The time complexity of detecting a type-D q-repetition is analyzed below.

In fact, Step D1 and Step D2 perform the same operation $\big(F_{k-1}F_k = S[b_{k-1}..i]$ and $p = \pi(F_{k-1}F_k)\big)$. Their computed values are to be used in Step D3. By the aid of the linear-time on-line LCP preprocessing for $S[b_{k-1}..i]$, each $\pi(S[b_{k-1}..j])$ for $b_{k-1} \le j \le i$ can be determined in constant time. Step D1 requires at most $\pi(S[b_{k-1}..j])$ character comparisons for each $\sigma\big(b_{k-1}+\pi(S[b_{k-1}..j])\big)$. Step D2 requires at most $\pi(S[b_{k-1}..i])$ character comparisons. According to Lemma 4.6, the total number of character comparisons needed for Step D1 and Step D2 to calculate $\sigma\big(b_{k-1}+\pi(S[b_{k-1}..j])\big)$'s for all $b_{k-1} < j \le i$ with $|S[b_{k-1}..j]| = 2 \times \pi(S[b_{k-1}..j])$ is bounded above by $\frac{3}{2} \times |S[b_{k-1}..i]|$.

Step D3 needs to compute $\sigma(b_{k-1}+p)$ only when $|F_{k-1}F_k| > p\times(q-1)$ holds. Let $i' = b_{k-1}+2p-1$ and $p' = \pi(S[b_{k-1}..i'])$. By Lemma 4.4, we have $p' = p$, and hence $\sigma(b_{k-1}+p) = \sigma(b_{k-1}+p')$. The latter was computed by Step D1 (when $S[b_k]$ was read) if $b_{k-1} < i' \le b_k$, and computed by Step D2 (when $S[i']$ was read) if $i' > b_k$. Therefore, Step D3 can be completed in constant time. The execution of Step D3 for $S[b_k]$, $S[b_k+1]$, ..., $S[i]$ takes total $O(|S[b_k..i]|)$ time. We have the following lemma.

Lemma 4.15. *Detecting a type-D q-repetition whose ending character belongs to F_k takes total $O(|F_{k-1}F_k|)$ time.*

4.6. Time complexity

The time complexity of detecting a q-repetition in S is analyzed below. Suppose that the execution halts after reading $S[m]$, i.e., there is a q-repetition ending at m or $m = |S|$, and $S[m]$ belongs to F_s, where $s \ge 2$. Computing the f-factorization of $S[1..m]$ on-line takes total $O(m \log \beta)$ time, where β is the number of distinct characters in $S[1..m]$. According to Lemma 4.1, the preprocessing takes total $\sum_{2\le k\le s} O(|F_{k-1}|) = O(m)$ time. According to Lemmas 4.9, 4.11, 4.13 and 4.15, the detection of four-type q-repetitions takes total $\sum_{2\le k\le s} O(|F_{k-1}F_k|) = O(m)$ time, where $q \ge 3$. Therefore, we have the following theorem, which summarizes the main result of this paper.

Theorem 4.16. *The on-line repetition detection problem can be solved in $O(m \log \beta)$ time, where m is the number of characters read and β is the number of distinct characters among them.*

5. Discussion and conclusion

Detecting repetitions in a string S plays an important role in many areas such as combinatorics, automata and formal language theory, data compression, bioinformatics, *etc*. Previously, there were off-line algorithms that could detect repetitions in S in $O(|S| \log \alpha)$ time, where α is the number of distinct characters in S. It was shown in [21] that detecting a square in S off-line over a general alphabet requires $\Omega(|S| \log |S|)$ time. Apparently, Theorem 3.5 is time-optimal.

We also presented an $O(m \log \beta)$ time algorithm that could detect a q-repetition on-line, where $q \ge 3$. There was no off-line algorithm proposed before for the problem. Notice that the $O(m \log \beta)$ time required for computing the f-factorization on-line is mainly due to the construction and traversal of the suffix tree. It was suggested in [6] as an open problem whether or not the f-factorization can be computed on-line in linear time. A positive answer to this problem would reduce the time complexity of Theorem 4.16 from $O(m \log \beta)$ to $O(m)$.

We assumed a general alphabet throughout this paper. For an integer alphabet, both building a suffix tree [8] and computing the f-factorization [2,6] can be achieved in linear time in an off-line manner. It is an interesting problem to investigate the time complexity of building a suffix tree and computing the f-factorization on-line over an integer alphabet.

In [12], Gusfield and Stoye proposed an $O(|S| \log \alpha)$ time off-line algorithm to report all distinct repetitions in S. Like Crochemore's algorithm, the algorithm also computed the f-factorization and the longest common prefix (and suffix). It seems possible that the algorithm can be adapted to an on-line manner to find all repetitions.

References

[1] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, Theoretical Computer Science 22 (1983) 297–315.
[2] G. Chen, S.J. Puglisi, W.F. Smyth, Fast and practical algorithms for computing all the runs in a string, in: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, CPM, in: Lecture Notes in Computer Science, vol. 4580, Springer-Verlag, 2007, pp. 307–315.
[3] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Information Processing Letters 12 (5) (1981) 244–250.
[4] M. Crochemore, Recherche linéaire d'un carré dans un mot, Comptes Rendus des Séances de l'Académie des Sciences. Série I. Mathématique 296 (18) (1983) 781–784.
[5] M. Crochemore, Transducers and repetitions, Theoretical Computer Science 45 (1) (1986) 63–86.
[6] M. Crochemore, L. Ilie, Computing longest previous factor in linear time and applications, Information Processing Letters 106 (2) (2008) 75–80.
[7] M. Crochemore, W. Rytter, Squares, cubes, and time–space efficient string searching, Algorithmica 13 (5) (1995) 405–425.
[8] M. Farach, Optimal suffix tree construction with large alphabets, in: Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS, 1997, pp. 137–143.
[9] M. Farach, M. Thorup, String matching in Lempel-Ziv compressed strings, Algorithmica 20 (4) (1998) 388–404.
[10] N.J. Fine, H.S. Wilf, Uniqueness theorem for periodic functions, Proceedings of the American Mathematical Society 16 (1) (1965) 109–114.
[11] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[12] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, Journal of Computer and System Sciences 69 (4) (2004) 525–546.
[13] M.A. Harrison, Introduction to Formal Language Theory, Addison-Wesley, 1978.
[14] J.-J. Hong, Analysis of repetitions in a string, Ph.D. Thesis, National Taiwan University, July 2007.
[15] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (2) (1977) 323–350.
[16] R.M. Kolpakov, G. Bana, G. Kucherov, mreps: Efficient and flexible detection of tandem repeats in DNA, Nucleic Acids Research 31 (13) (2003) 3672–3678.
[17] R.M. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS, 1999, pp. 596–604.
[18] S.R. Kosaraju, Computation of squares in a string, in: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, CPM, in: Lecture Notes in Computer Science, vol. 807, Springer-Verlag, 1994, pp. 146–150.
[19] H.-F. Leung, Z. Peng, H.-F. Ting, An efficient algorithm for online square detection, Theoretical Computer Science 363 (1) (2006) 69–75.
[20] M. Lothaire, Applied Combinatorics on Words, Cambridge University Press, 2005.
[21] 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.
[22] M.G. Main, R.J. Lorentz, Linear time recognition of squarefree strings, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series, vol. F12, Springer-Verlag, 1985, pp. 271–278.
[23] V. Parisi, V.D. Fonzo, F. Aluffi-Pentini, STRING: Finding tandem repeats in DNA sequences, Bioinformatics 19 (14) (2003) 1733–1738.
[24] R. Ross, R. Winklmann, Repetitive strings are not context-free, Technical Report CS-81-070, Washington State University, Pullman, WA, 1981.
[25] J. Stoye, D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, Theoretical Computer Science 270 (1–2) (2002) 843–856.
[26] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[27] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.