Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 363 (2006) 69–75
Theoretical Computer Science
www.elsevier.com/locate/tcs

An efficient algorithm for online square detection

H.F. Leung[a], Z.S. Peng[b], H.F. Ting[b,*]

[a] *Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong*
[b] *Department of Computer Science, The University of Hong Kong, Hong Kong*

Abstract

A square is a string that can be divided into two identical substrings. The problem of square detection has found applications in areas such as bioinformatics and data compression. There are many offline algorithms for the problem. In this paper, we give the first online algorithm for deciding whether a string contains a square. Our algorithm runs in total $O(h \log^2 h)$ time where h is the length of the longest prefix of the input string that does not contain a square.

Keywords: Online algorithms; Squares; Square-free detection; String processing

1. Introduction

Given a string $T = t_1 t_2 \dots t_n$ of characters from some alphabet Σ, we say that T is a *square* if it is the concatenation of two identical substrings, or more precisely, if $t_1 t_2 \dots t_{\lfloor n/2 \rfloor} = t_{\lfloor n/2 \rfloor + 1} \dots t_{n-1} t_n$. We say that T contains a square if it has a substring $t_i t_{i+1} \dots t_j$ that is a square. T is *square-free* if it has no such substring. Square-free strings, as well as their detection, have found important applications in areas such as bioinformatics [4,6,13], data compression [24] and formal language theory [5,23,22,16,17].

The problem of detecting squares in a string has been studied extensively in the literature. There are several $O(n \log n)$-time offline algorithms for determining whether a string of n characters is square-free [18,8,20]. When the alphabet Σ is fixed (i.e., when Σ has a constant number of characters), the time complexity can be reduced to $O(n)$ [8,20]. However, it was proved in [19] that for general Σ, any algorithm for the problem must run in $\Omega(n \log n)$ time. Besides determining whether a string is square-free, there are also algorithms that find all the squares in a string. Apostolico and Preparata [3] and Crochemore [7] gave $O(n \log n)$-time algorithms for finding all squares in a string of length n; their algorithms assume that the alphabet Σ is totally ordered. Main and Lorentz [19] later showed that for general Σ, we can find all the squares in the same time bound. More recently, Stoye and Gusfield gave a much simpler $O(n \log n)$-time algorithm for the problem [25]. They also considered the case when the alphabet Σ is fixed and gave a linear time algorithm that finds all the (essentially different) squares in a string [12]. Both of their algorithms use suffix trees. There are also randomized algorithm [21] and parallel algorithms [1,9,10,2] for square detection.

* Corresponding author.
E-mail address: hfting@cs.hku.hk (H.F. Ting).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.06.011

In this paper, we initiate the study of online algorithms for square detection and give the first online algorithm for determining whether a string is square-free. The study is motivated by our research on using local search to solve some classical constraint satisfaction problems [14,15,26]. Local search is efficient in general. However, there is no guarantee that it stops; the search can be trapped in some infinite loop. It can be shown that such loop corresponds to a square in some string whose characters encode the states of the search. To avoid wasting time in an infinite loop, we need to detect squares in the string online. More precisely, every time the search makes a move and a new character is added to the string, we need to determine if there is any square in the new string. We stop if there is a square.

Our online algorithm for determining whether a string is square-free works for general alphabet Σ. In other words, we do not make any assumption on Σ and given any two characters in Σ, we can only decide whether they are equal or not (i.e., $=$ or $\neq$). Given any input string T, our algorithm reads T character by character. After reading a new character, it checks immediately whether there is a square in the string. If the first square in T ends at the hth position of T, our algorithm stops after reading the first h characters and reports the square. The whole process runs in $O(h \log^2 h)$ time. Recall that for general alphabet, any offline algorithm for deciding whether a string is square-free runs in $\Omega(n \log n)$ time where n is the length of T. In general, n can be much larger than h.

The paper is organized as follows. Section 2 gives the definitions and notations necessary for our discussion. Section 3 describes the main procedure of our algorithm, and Section 4 shows how to use this procedure to detect the first square of the input string. Section 5 gives concluding remarks.

2. Preliminaries

For any string X, we let $|X|$ denote the length of X, which is defined to be the number of characters in X. Given another string X', let XX' denote the concatenation of X and X', which is the string obtained by appending X' at the end of X. We say that a string is a *square* if it is the concatenation of two identical strings. We say that a string is a *pseudo-square with center G* if it is equal to XGX for some string X. Note that a square is a pseudo-square with an empty string as its center. Furthermore, a pseudo-square may have different centers. For example, the string *aabbbbbaa* is a pseudo-square with center *bbbb*, and is also a pseudo-square with center *abbbba*. In particular, every string is a pseudo-square with center equal to itself.

The input of our algorithm is a string $T = t_1 t_2 t_3 \dots t_n$ of characters. For any integer $1 \leqslant i \leqslant j \leqslant n$, we let $T_{i\dots j}$ denote the substring $t_i t_{i+1} \cdots t_j$. For any $i \leqslant \ell \leqslant j$, we say that the substring $T_{i\dots \ell}$ is a *prefix* of $T_{i\dots j}$. We say that the substring $T_{\ell\dots j}$ is a *suffix* of $T_{i\dots j}$. A suffix is a *square suffix* if it is a square, and is a *pseudo-square suffix* if it is a pseudo-square. Note that to solve our square detection problem, it suffices to check, after reading each character T_h, whether $T_{1\dots h}$ has a square suffix. The following fact suggests an approach for checking suffix square.

Fact 1. *Let $T_{i\dots j}$ be any substring of T. Suppose that $T_{i\dots j}$ has a pseudo-square suffix YGY. If the string of the next $|G|$ characters $T_{(j+1)\dots(j+|G|)}$ is equal to the center G, then $T_{1\dots(j+|G|)}$ has a square suffix XX where $X = YG$.*

In the next section, we describe an efficient procedure `Dcenter`(i, j) for the following task: after reading character T_h for each $h > j$, determine whether $T_{(j+1)\dots h}$ is equal to the center of some pseudo-square suffix of $T_{i\dots j}$. Note that if there is such pseudo-square suffix YGY, then by Fact 1, we conclude that $T_{1\dots h}$ has a square suffix $YGYG$. In Section 4, we describe an online algorithm `Dsquare` for finding the first square of T (and thus determining whether T is square-free). Roughly speaking, `Dsquare` invokes a small, but sufficient number of instances of `Dcenter` such that if $T_{1\dots h}$ has a suffix square, then there must be an instance `Dcenter`(i, j) running where $T_{(j+1)\dots h}$ is equal to the center of some pseudo-square suffix of $T_{i\dots j}$. Then, `Dcenter`(i, j) will detect the corresponding suffix square.

3. The procedure `Dcenter`

In this section, we describe the procedure `Dcenter`(i, j), which checks the centers in the substrings $T_{i\dots j}$. In our discussion, we assume that $T_{i\dots j}$ is square-free. When we describe our algorithm `Dsquare` in the next section, we will verify that for each instance `Dcenter`(i, j) invoked by `Dsquare`, $T_{i\dots j}$ is indeed square-free.

For any pair (a, b) where $i \leqslant a \leqslant b \leqslant j$, we say that (a, b) is a *center boundary for* $T_{i\ldots j}$ if $T_{a\ldots b}$ is the center of some pseudo-square suffix of $T_{i\ldots j}$. Below, we give a high level description of `Dcenter`(i, j):

> Build a list L of center boundaries for $T_{i\ldots j}$. Then, after reading character T_h for each $j + 1 \leqslant h \leqslant 2j - i + 1$, check if there is a center boundary $(a, b) \in L$ such that $T_{a\ldots b} = T_{(j+1)\ldots h}$. If there is such (a, b), stop and report that $T_{a\ldots b} = T_{(j+1)\ldots h}$.

Note that `Dcenter`(i, j) can stop after reading T_{2j-i+1} because for any $h > 2j - i + 1$, the length of $T_{i\ldots j}$ is smaller than that of $T_{(j+1)\ldots h}$ and $T_{(j+1)\ldots h}$ cannot possibly be equal to the center of any pseudo-square of $T_{i\ldots j}$. Since L may have as many as $\Theta((j - i)^2)$ entries, a straightforward implementation of the procedure is not efficient. To reduce the running time, we show in Section 3.1 that we only need to store $O(j - i)$ center boundaries in L. Furthermore, we show that we can construct this shorter list of center boundaries in $O(j - i)$ time. Then, in Section 3.2, we explain how to check the entries in L efficiently so that the procedure `Dcenter`(i, j) runs in total $O((j - i)\log(j - i))$ time.

3.1. A short list of center boundaries

Given any center boundary (a, b) for $T_{i\ldots j}$, we say that (a, b) is *minimal* if there is no shorter center boundary for $T_{i\ldots j}$ that starts at a, or more precisely, if for any $a \leqslant \ell < b$, (a, ℓ) is not a center boundary for $T_{i\ldots j}$. Note that there are only $O(j - i)$ minimal center boundaries for $T_{i\ldots j}$. The following lemma suggests that it should be no use for `Dcenter`(i, j) to check those center boundaries that are not minimal.

Lemma 1. *Suppose that* `Dcenter`(i, j) *stops and reports that* $T_{a\ldots b} = T_{(j+1)\ldots h}$ *after reading* T_h. *Then*, (a, b) *must be a minimal center boundary*.

Proof. Suppose to the contrary that the center boundary (a, b) is not minimal. By definition, there is a center boundary (a, ℓ) where $a \leqslant \ell < b$, and from the facts that $T_{a\ldots\ell}$ is a prefix of $T_{a\ldots b}$ and $T_{a\ldots b} = T_{(j+1)\ldots h}$, we conclude that $T_{(j+1)\ldots h}$ has a prefix $T_{(j+1)\ldots k}$ that is equal to $T_{a\ldots\ell}$. Since $\ell < b$, we have $k < h$. Note that the execution of `Dcenter`(i, j) would have stopped after reading T_k (because $T_{(j+1)\ldots k}$ is equal to the center $T_{a\ldots\ell}$) and T_h would not be read; a contradiction. □

Hence, `Dcenter`$(i \ldots j)$ only needs to check the $O(j - i)$ minimal center boundaries for $T_{i\ldots j}$. Note that (i, j) and (j, j) are the minimal center boundaries for $T_{i\ldots j}$ starting at i and j, respectively. From the assumption that $T_{i\ldots j}$ is square-free, we can use the following lemma to identify the other minimal center boundaries.

Lemma 2. *Suppose that* $T_{i\ldots j}$ *is square-free. Then, for any* $i < a \leqslant b < j$, (a, b) *is a minimal center boundary for* $T_{i\ldots j}$ *if and only if* $T_{(b+1)\ldots j}$ *is the longest suffix of* $T_{i\ldots j}$ *that is equal to a suffix of* $T_{i\ldots(a-1)}$.

Proof. It can be verified from the definition that (a, b) is a minimal center boundary for $T_{i\ldots j}$ if and only if (i) the suffix $T_{(b+1)\ldots j}$ of $T_{i\ldots j}$ is equal to a suffix of $T_{i\ldots(a-1)}$ and (ii) for any ℓ where $a \leqslant \ell < b$, $T_{(\ell+1)\ldots j}$ is not a suffix of $T_{i\ldots(a-1)}$. Therefore, to prove that $T_{(b+1)\ldots j}$ is the longest suffix of $T_{i\ldots j}$ that is equal to a suffix of $T_{i\ldots(a-1)}$, we only need to prove that for any $i \leqslant \ell < a$, $T_{(\ell+1)\ldots j}$ is not equal to any suffix of $T_{i\ldots(a-1)}$.

Suppose to the contrary that there is an integer $i \leqslant \ell < a$ such that $T_{(\ell+1)\ldots j}$ is equal to some suffix $T_{h\ldots(a-1)}$ of $T_{i\ldots(a-1)}$. Since $\ell < a$, we have $Y = T_{h\ldots\ell}$ is a prefix of $T_{h\ldots(a-1)}$, and together with the fact that $T_{h\ldots(a-1)} = T_{(\ell+1)\ldots j}$, we conclude that $T_{(\ell+1)\ldots j}$ has a prefix, namely $T_{(\ell+1)\ldots(\ell+|Y|)}$, that is equal to $T_{h\ldots\ell}$. Therefore, $T_{h\ldots(\ell+|Y|)} = T_{h\ldots\ell}T_{(\ell+1)\ldots(\ell+|Y|)} = YY$ is a square in $T_{i\ldots j}$, which contradicts the assumption that $T_{i\ldots j}$ is square-free. □

The next lemma shows that we can find the longest suffixes for identifying the minimal center boundaries of $T_{i\ldots j}$ in linear time.

Lemma 3. *We can find for all* $i < a < j$, *the longest suffix of* $T_{i\ldots j}$ *that is equal to some suffix of* $T_{i\ldots(a-1)}$ *in total* $O(j - i)$ *time*.

Proof. Note that given any string $P_{1\ldots n}$, there is a linear time algorithm that finds, for all $1 \leqslant \ell \leqslant n$, the longest prefix of $P_{1\ldots n}$ that is equal to some prefix of $P_{(\ell+1)\ldots n}$ (e.g., [11, pp. 7–10]). By reversing $T_{i\ldots j}$ and applying this algorithm,

we can find the longest suffix of $T_{i\ldots j}$ that is equal to some suffix of $T_{i\ldots(a-1)}$ for all a between i and j in $O(j-i)$ time. □

3.2. Checking the list of minimal center-boundaries

After constructing the list L of minimal center boundaries for $T_{i\ldots j}$, the procedure `Dcenter`(i, j) checks for each $j+1 \leqslant h \leqslant 2j-i+1$, whether there is a center boundary $(a, b) \in L$ such that $T_{a\ldots b} = T_{(j+1)\ldots h}$. Note that checking the list L in brute force may take quadratic time. Below, we describe a more efficient checking procedure. Our idea is to remove the center boundaries (a, b) from L as soon as we find that $T_{a\ldots b}$ cannot be equal to $T_{(j+1)\ldots h}$ for any h.

Observe that for any $(a, b) \in L$, if we find that $T_a \neq T_{j+1}$, we can remove (a, b) from L immediately because for any h, $T_{a\ldots b} \neq T_{(j+1)\ldots h}$ (the first characters of the two substrings are already different). In general, for any $0 \leqslant \ell \leqslant b-a$, if we find that $T_{a+\ell} \neq T_{j+\ell+1}$, we can remove (a, b) from L immediately because $T_{a\ldots b} \neq T_{(j+1)\ldots h}$ for any h. Based on this observation, `Dcenter`(i, j) checks the list L as follows.

> After reading T_h for each $h \in \{j+1, j+2, \ldots, 2j-i+1\}$, or equivalently, after reading $T_{j+\ell+1}$ for each $\ell \in \{0, 1, \ldots, j-i\}$, do the following:
>
> - Remove all the center boundaries (a, b) from L where $T_{a+\ell} \neq T_h$ $(= T_{j+\ell+1})$.
> - Check whether there is a center boundary (a, b) remaining in L such that the length of $T_{a\ldots b}$ is equal to that of $T_{(j+1)\ldots h}$. If such (a, b) exists, `Dcenter`(i, j) stops and reports that $T_{a\ldots b} = T_{(j+1)\ldots h}$.

To see that the above procedure is correct, note that if `Dcenter`(i, j) stops without reporting anything, then there must be no minimal center boundary (a, b) for $T_{i\ldots j}$ such that $T_{a\ldots b} = T_{(j+1)\ldots h}$ for some h; if there is such (a, b), it will not be removed from L and `Dcenter`(i, j) can at least report $T_{a\ldots b} = T_{(j+1)\ldots h}$ after reading h. On the other hand, if `Dcenter`(i, j) stops after reading T_h and reports that $T_{a\ldots b} = T_{(j+1)\ldots h}$, then by the design we have (i) $T_{a\ldots b}$ and $T_{(j+1)\ldots h}$ have the same length and (ii) $T_a = T_{j+1}, T_{a+1} = T_{j+2}, \ldots, T_b = T_h$ (because (a, b) remains in L after reading $T_{j+1}, T_{j+2}, \ldots, T_h$). It follows that $T_{a\ldots b}$ is indeed equal to $T_{(j+1)\ldots h}$.

We now estimate its total running time. Suppose that T_{j+r} is the last character that `Dcenter`(i, j) reads. For any $1 \leqslant k \leqslant r$, let P_k be the set of center boundaries in L just before `Dcenter`(i, j) reads the character T_{j+k}. Since `Dcenter`(i, j) scans the list L once after reading each character, the total running time for the checking is $O(|P_1| + |P_2| + \cdots + |P_r|)$. Note that P_1 is the set of all minimal center boundaries for $T_{i\ldots j}$ and thus $|P_1| = O(j-i)$. The next lemma gives a bound on each $|P_k|$.

Lemma 4. *For each $1 \leqslant k \leqslant r$, we have $|P_k| = O((j-i)/k)$.*

Proof. Note that for any center boundary $(a, b) \in P_k$, the length of $T_{a\ldots b}$ is at least k. Below, we prove that for any two center boundaries (a, b) and (c, d) in P_k, the substrings $T_{a\ldots(a+k-1)}$ and $T_{c\ldots(c+k-1)}$ cannot overlap. In other words, we have either $c > a+k-1$ or $a > c+k-1$. This implies that there are at most $(j-i+1)/k$ center boundaries in P_k and the lemma follows.

Assume to the contrary that $T_{a\ldots(a+k-1)}$ and $T_{c\ldots(c+k-1)}$ overlap. Suppose without loss of generality that $a \leqslant c$. Then, we have $a \leqslant c \leqslant a+k-1$. Recall that P_k is the set of entries in L just before `Dcenter`(i, j) reading T_{j+k}. Since $(a, b) \in P_k$, `Dcenter`(i, j) does not remove it after reading $T_{j+1}, T_{j+2}, \ldots, T_{j+k-1}$ and hence $T_a = T_{j+1}$, $T_{a+1} = T_{j+2}, \ldots, T_{a+k-2} = T_{j+k-1}$, or equivalently, $T_{a\ldots(a+k-2)} = T_{(j+1)\ldots(j+k-1)}$. Similarly, we have $T_{c\ldots(c+k-2)} = T_{j\ldots(j+k-1)}$ and thus $T_{a\ldots(a+k-2)} = T_{c\ldots(c+k-2)}$.

Since $c \leqslant a+k-1$, the substring $Y = T_{a\ldots(c-1)}$ is a prefix of $T_{a\ldots(a+k-2)}$. Together with the fact that $T_{a\ldots(a+k-2)} = T_{c\ldots(c+k-2)}$, we conclude that $T_{c\ldots(c+k-2)}$ has a prefix, namely $T_{c\ldots(c+|Y|-1)}$, that is equal to $T_{a\ldots(c-1)}$. Thus, $T_{a\ldots(c+|Y|-1)} = T_{a\ldots(c-1)}T_{c\ldots(c+|Y|-1)} = YY$ is a square in $T_{i\ldots j}$, contradicts our assumption that $T_{i\ldots j}$ is square-free. □

Based on Lemma 4, we conclude that the total running time for the checking procedure is $O(|P_1| + |P_2| + \cdots + |P_r|) = O((j-i)(1 + 1/2 + \cdots + 1/r))$. Now, we are ready to summarize our discussion and establish an upper bound on the time complexity of `Dcenter`(i, j).

Theorem 5. `Dcenter`(i, j) *runs in total* $O((j-i)\log(j-i))$ *time.*

Proof. By Lemmas 2 and 3, `Dcenter`(i, j) constructs the list L of necessary minimal center boundaries in $O(j - i)$ time. The total time for checking the list L (and possibly remove some of its entries) after reading each new character is $O(|P_1| + |P_2| + \cdots + |P_r|) = O((j - i)(1 + 1/2 + \cdots + 1/r))$. Note that $1 + 1/2 + \cdots + 1/r = O(\log r)$ and $r \leqslant j - i$. The theorem follows. □

4. The algorithm `Dsquare`

The basic task of `Dsquare` is to invoke a small, but sufficient number of instances of `Dcenter` so that after reading every new character T_h, it can correctly determine whether $T_{1\ldots h}$ has a square suffix. The following lemma gives some hint on identifying this small set of instances.

Lemma 6. *Suppose that $T_{1\ldots h}$ has a square suffix $T_{g\ldots h} = XX$. Then, for any integers i, j where $1 \leqslant i \leqslant g$ and $h - |X| < j < h$, $T_{i\ldots j}$ has a pseudo-square suffix whose center is equal to $T_{(j+1)\ldots h}$.*

Proof. Suppose that $T_{g\ldots h} = T_{g\ldots m}T_{(m+1)\ldots h}$ where $T_{g\ldots m} = T_{(m+1)\ldots h} = X$. Note that $m = h - |X|$ and thus $m < j < h$. Let $T_{(m+1)\ldots j} = Y$ and $T_{(j+1)\ldots h} = G$. Since $T_{g\ldots m} = T_{(m+1)\ldots h} = T_{(m+1)\ldots j}T_{(j+1)\ldots h} = YG$, we have $T_{g\ldots j} = T_{g\ldots m}T_{(m+1)\ldots j} = YGY$. Therefore, $T_{i\ldots j}$ has a pseudo-square suffix $T_{g\ldots j} = YGY$ whose center is equal to $T_{(j+1)\ldots h} = G$. □

Lemma 6 suggests that there should be many substrings $T_{i\ldots j}$ that can be used to check whether $T_{1\ldots h}$ has a square suffix. More importantly, it implies that one such $T_{i\ldots j}$ can be used to check many different $T_{1\ldots h}$ and thus we do not need to invoke too many instances of `Dcenter`. To specify a small set of instances that guarantees the correctness of `Dsquare`, we need the following definition: for any positive integers i, j, ℓ, we say that the pair (i, j) is an ℓ-pair if

- the second component j is equal to $k2^\ell$ for some integer k, and
- the first component i is equal to $\max\{1, j - 2^{\ell+2} + 1\}$.

For example, $(1, 2)$, $(1, 4)$, $(1, 6)$, $(1, 8)$, $(3, 10)$, $(5, 12)$ are the first six 1-pairs, and $(1, 4)$, $(1, 8)$, $(1, 12)$, $(1, 16)$, $(5, 20)$, $(9, 24)$ are the first six 2-pairs.

We are now ready to give the details of `Dsquare`. Roughly speaking, `Dsquare` invokes the set of instances `Dcenter`(i, j) where (i, j) is an ℓ-pair. For simplicity, our algorithm is presented as a parallel algorithm. It should be clear that the algorithm can be sequentialized and its running time is just the total running time of all instances of `Dcenter` invoked by `Dsquare`.

After reading each T_h, do the following steps:
if ((there is a square in $T_{(h-3)\ldots h}$) or (any of the running `Dcenter`
detects a square in $T_{1\ldots h}$)), report the square and stop.
$\ell := 1$;
while $(2^\ell \leqslant h)$ **do**
begin
if $(h \bmod 2^\ell = 0)$
begin
$g := \max\{1, h - 2^{\ell+2} + 1\}$;
start `Dcenter`(g, h) for the ℓ-pair (g, h);
end;
$\ell := \ell + 1$;
end;
The following lemma asserts that `Dsquare` determines whether T is square-free correctly.

Lemma 7. *Suppose that h is the smallest integer such that $T_{1\ldots h}$ contains a square suffix $T_{g\ldots h} = XX$. Then,* `Dsquare` *detects a square and stops after reading T_h.*

Proof. The lemma is obviously true if $|XX| \leqslant 4$. Suppose that $|XX| > 4$. Then,

- $2^\ell < |X| \leqslant 2^{\ell+1}$ for some integer $\ell > 1$, and
- $k2^\ell < h \leqslant (k + 1)2^\ell$ for some integer $k \geqslant 1$.

Consider the ℓ-pair (i, j) where $i = \max\{1, k2^\ell - 2^{\ell+2} + 1\}$ and $j = k2^\ell$. Note that $j < h$ and by design, `Dsquare` starts `Dcenter`(i, j) before T_h is read. Observe that

$$i = \max\{1, j - 2^{\ell+2} + 1\} \leqslant \max\{1, h - 2^{\ell+2} + 1\} \leqslant \max\{1, h - 2|X| + 1\} \leqslant g.$$

Furthermore, since $|X| > 2^\ell$ and $h \leqslant (k+1)2^\ell$, we have $j = k2^\ell > h - |X|$. By Lemma 6, we conclude that $T_{i\ldots j}$ has a pseudo-square suffix whose center is equal to $T_{(j+1)\ldots h}$. After T_h is read, `Dcenter`(i, j) reports this center and `Dsquare` stops. □

Based on Lemma 7, it can be verified that for every instance `Dcenter`(i, j) invoked by `Dsquare`, the corresponding substring $T_{i\ldots j}$ must be square-free; if $T_{i\ldots j}$ contains square $T_{a\ldots b}$, then `Dsquare` will stop after reading T_b and will not invoke `Dcenter`(i, j). Together with the implementation of `Dcenter`(i, j) given in Section 3, we can estimate the total running time of `Dsquare`. Let h be the smallest integer such that $T_{1\ldots h}$ has a square suffix. If T is square-free, let h be length of T.

Theorem 8. `Dsquare` *runs in* $O(h \log^2 h)$ *time.*

Proof. For any ℓ, let τ_ℓ be the total running time of the instances of `Dcenter`(i, j) invoked by `Dsquare` where (i, j) is an ℓ-pair. Note that `Dsquare` stops before reading $T_{2^{\lceil \log h \rceil}+1}$ and it will not invoke `Dcenter`(i, j) for any ℓ-pair (i, j) with $j > 2^{\lceil \log h \rceil}$. It follows that $\ell \leqslant \lceil \log h \rceil$ and the running time of `Dsquare` is $O(\sum_{1 \leqslant \ell \leqslant \lceil \log h \rceil} \tau_\ell)$. Note that for a fixed ℓ,

- there are $O(h/2^\ell)$ different ℓ-pairs (i, j) with $j \leqslant h$, and
- for each ℓ-pair (i, j), `Dcenter`(i, j) runs in $O(\ell 2^\ell)$ time (Theorem 5).

Hence, $\tau_\ell = O(\ell h)$ and $\sum_{1 \leqslant \ell \leqslant \lceil \log h \rceil} \tau_\ell = O(h \log^2 h)$. □

5. Conclusion

In this paper, we give the first online algorithm for determining whether a string T is square-free. Our algorithm works for general alphabet. If T is square-free, our algorithm runs in $O(n \log^2 n)$ time where n is the length of T. Note that for general alphabet, any offline algorithm for determining whether T is square-free runs in $\Omega(n \log n)$ time. It is interesting to find out if the running time of our algorithm can be reduced to $O(n \log n)$. Another interesting open question is to study how efficiently the online version of cube detection problem can be solved.

Acknowledgements

We are grateful to the anonymous referees for helpful suggestions on improving the presentation of the paper.

References

[1] A. Apostolico, Optimal parallel detection of squares in strings, Algorithmica 8 (4) (1992) 285–319.
[2] A. Apostolico, D. Breslauer, An optimal $O(\log \log n)$-time parallel algorithm for detecting squares in a string, SIAM J. Comput. 25 (6) (1996) 1318–1331.
[3] A. Apostolico, E.P. Preparata, Optimal off-line detection of repetitions in a string, Theoret. Comput. Sci. 22 (1983) 297–315.
[4] G. Benson, Tandem Repeats Finder: a program to analyze DNA sequences, Nucl. Acids Res. 27 (2) (1999) 573–580.
[5] J.A. Brzozowski, K. Culik, A. Gabrielian, Classification of noncounting events, J. Comput. System Sci. 5 (1971) 41–53.
[6] A. Castelo, W. Martins, G. Gao, TROLL–tandem repeat occurrence locator, Bioinformatics 18 (4) (2002) 634–636.
[7] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Inform. Process. Lett. 12 (5) (1981) 244–250.
[8] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 45 (1986) 63–86.
[9] M. Crochemore, W. Rytter, Efficient parallel algorithms to test squarefreeness and factorize strings, Inform. Process. Lett. 38 (2) (1991) 57–60.
[10] M. Crochemore, W. Rytter, Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays, Theoret. Comput. Sci. 88 (1991) 59–82.
[11] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, 1997.
[12] D. Gusfield, J. Stoye, Linear-time algorithms for finding and representing all tandem repeats in a string, J. Comput. System Sci. 69 (2004) 525–546.
[13] R. Kolpakov, G. Bana, G. Kucherov, `mreps`: efficient and flexible detection of tandem repeats in DNA, Nucl. Acids Res. 31 (13) (2003) 3672–3678.

[14] J.H.M. Lee, H.F. Leung, H.W. Won, Extending GENET for non-binary constraint satisfaction problems, in: Proc. Seventh Internat. Conf. on Tools with Artifical Intelligence, 1995, pp. 338–343.
[15] J.H.M. Lee, H.F. Leung, H.W. Won, Performance of a comprehensive and efficient constraint library based on local search. in: Proc. 11th Australian Join Conf. on Artifical Intelligence, 1998, pp. 13–17.
[16] M. Main, An infinite square-free co-CFL, Inform. Process. Lett. 20 (2) (1985) 105–107.
[17] M. Main, W. Bucher, D. Haussler, Applications of an infinite square-free co-CFL, Theoret. Comput. Sci. 49 (1987) 113–119.
[18] M. Main, R. Lorentz, An $O(n \log n)$ algorithm for recognizing repetition, Technical Report CS-79-056, Washington State University, 1979.
[19] M. Main, R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, J. Algorithms 5 (3) (1984) 422–432.
[20] M. Main, R. Lorentz, Linear time recognition of squarefree strings, Combin. Algorithms Words 12 (1985) 271–278.
[21] M.O. Rabin, Discovering repetitions in strings, Combin. Algorithms Words, (1985) 279–288.
[22] R. Ross, R. Winklmann, Repetitive strings are not context-free, Technical Report CS-81-070, Washington State University, Pullman, WA, 1981.
[23] H.J. Shyr, A strongly primitive word of arbitrary length and its applications, Internat. J. Comput. Math. 6 (1977) 165–170.
[24] J. Storer, Data Compression: Methods and Theory, Computer Science Press, New York, NY, 1987.
[25] J. Stoye, D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, Theoret. Comput. Sci. 270 (2002) 843–850.
[26] J.H.Y. Wong, H.F. Leung, Solving fuzzy constraint satisfaction problems with fuzzy GENET, in: Proc. 10th IEEE Internat. Conf. on Tools with Artifical Intelligence, 1998, pp. 180–191.