International Journal of Foundations of Computer Science
Vol. 18, No. 2 (2007) 401–414
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

ONLINE AND DYNAMIC RECOGNITION OF SQUAREFREE STRINGS*

JESPER JANSSON[†]
Department of Computer Science and Communication Engineering,
Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
jj@tcslab.csce.kyushu-u.ac.jp

and

ZESHAN PENG
Department of Computer Science, The University of Hong Kong,
Pokfulam Road, Hong Kong
zspeng@cs.hku.hk

Received 16 December 2005
Accepted 21 November 2006
Communicated by J. Karhumaki

ABSTRACT

The online squarefree recognition problem is to detect the first occurrence of a square in a string whose characters are provided as input one at a time. We present an efficient algorithm to solve this problem for strings over arbitrarily ordered alphabets in $O(n \log n)$ time, where n is the ending position of the first square. We also note that the same technique yields an $O(n \cdot (|\Sigma_n| + \log n))$-time algorithm for general alphabets, where $|\Sigma_n|$ is the number of different symbols in the first n positions of the input string. (This is faster than the previously fastest method for general alphabets when $|\Sigma_n| = o(\log^2 n)$.) Finally, we present a simple algorithm for a dynamic version of the problem over general alphabets in which we are initially given a squarefree string, followed by a series of updates, and the objective is to determine after each update if the resulting string is still squarefree.

Keywords: Square detection; squarefree string; online algorithm.

1. Introduction

1.1. *Problem definitions*

For any string T, let $|T|$ be the length of T. For any positive integers i, j satisfying $1 \leq i \leq j \leq |T|$, denote the substring of T starting at position i and ending at

*A preliminary version of this paper has appeared in *Proceedings of the Thirtieth International Symposium on Mathematical Foundations of Computer Science* (MFCS 2005), volume 3618 of *Lecture Notes in Computer Science*, pages 520–531, Springer Verlag, 2005.
[†]Supported in part by the Graduate School of Informatics, Kyoto University, Japan.

position j by $T[i..j]$, and define $T[i] = T[i..i]$. A substring of the form $T[i..(i+2k-1)]$, where k is a positive integer, is called a *square* (also known in the literature as a *tandem repeat*) if for every $x \in \{0, 1, \ldots, (k-1)\}$, it holds that $T[i+x] = T[i+k+x]$. If T does not contain a square, then T is *squarefree.*

In this paper, we study *the squarefree recognition problem.* We distinguish between the *offline*, *online*, and *dynamic* versions of this problem. In the offline version, an entire string T is provided as input directly, and the objective is to determine whether or not T contains a square. In the online version, the characters of the string T arrive one at a time in sequential order, and the objective is to determine after receiving each character if the string obtained so far contains a square; if so, report this fact and stop. Finally, in the dynamic version, a squarefree string T is provided as the initial input and then followed by a series of updates of the form "replace the symbol at position q of T by the symbol x", "insert the symbol x into T at position q", or "delete the symbol at position q from T", and the objective is to decide after each update if the resulting T contains a square and if so, report that T is no longer squarefree and stop.

The alphabet of the input string affects how quickly the different squarefree recognition problems can be solved. Under the least restrictive assumption, the symbols in T cannot be relatively ordered; a comparison between two symbols only tells us if they are equal or not. We call this type of alphabet a *general* alphabet. If the symbols in T admit some arbitrary lexicographical ordering so that any comparison between two symbols yields one of the three outcomes $<$, $=$, and $>$, then the alphabet is called *ordered.*[b] Next, in an *integer* alphabet, all symbols are integers in the range $\{1, 2, \ldots, |T|\}$. Finally, if the size of the alphabet is bounded by a constant, then we say that the alphabet is *constant.* Note that these four alphabet types are decreasingly restrictive in the sense that an algorithm for, e.g., general alphabets will also work for ordered alphabets (but not necessarily the other way around).

1.2. *Motivation*

The online squarefree recognition problem has applications in diverse areas such as string algorithms, bioinformatics, and online data compression [10] where we may stop scanning the input as soon as a square has been formed (this can save a lot of time if a square appears at the beginning of a very long input string). It is also motivated by the local search method for solving the constraints satisfaction problem in [8, 9, 14]; to guarantee that the method will not be trapped in some infinite loop, one can encode the successive states of the search as characters in a growing string and terminate the method if a square is formed at the end of this string [10]. See [10] for additional references.

One of our main results in this paper is a fast algorithm for the case of arbitrarily ordered alphabets. This is a reasonable assumption for most applications because

[b]As an example to illustrate the difference between general and ordered alphabets, consider the element uniqueness problem which has a lower bound of $\Omega(n^2)$ time for general alphabets but admits an $O(n \log n)$-time solution for ordered alphabets (see [2]).

when the symbols of the input string are encoded as binary numbers in a computer, this will induce a lexicographical ordering among them.

1.3. *Previous results*

For the offline and general alphabet case, Main and Lorentz [11] gave an algorithm that can be used to report all s occurrences of squares in a string T of length n in $O(n \log n + s)$ time, or just the longest square in T in $O(n \log n)$ time. This is optimal because to determine if T is squarefree takes $\Omega(n \log n)$ time for general alphabets [11]. For the offline and non-general alphabet case, other efficient algorithms for finding squares were presented earlier in [1] and [4]. For the offline and constant alphabet case, there exist algorithms that determine if T is squarefree in optimal $O(n)$ time [5, 12]. Parallel algorithms for finding squares offline have also been developed (see [2]).

For the online and general alphabet case, Leung, Peng, and Ting [10] gave an algorithm which has a running time of $O(n \log^2 n)$, where n is the ending position in T of the first square. (This is just a factor of $O(\log n)$ worse than the optimal offline algorithm for general alphabets mentioned above.) The algorithm of Leung, Peng, and Ting is outlined in Section 2.1.

The dynamic version of the problem has not been studied before.

Recently and independently of this paper, Chen, Hong, and Lu [3] claimed to have an $O(n \log |\Sigma_n|)$-time algorithm for the online squarefree recognition problem for ordered alphabets, where $|\Sigma_n|$ is the number of distinct characters in the first n positions of T, and that the expected running time of their algorithm can easily be reduced to $O(n)$ using hash tables. There are two problems with this claim. Firstly, in the subroutine named $A_R(i_1, i_2, i_3)$ described in Section 3.2 of [3], the authors maintain a data structure from [11] (also described in [7]) for the substring $T[i_2, i]$ while i ranges from $i_2 + 1$ to i_3 such that the length of the longest common prefix of $T[i_2..i]$ and $T[j..i]$ for any value of j with $i_2 \leq j \leq i$ can be obtained in $O(1)$ time, and such that the data structure for $T[i_2..i]$ can be obtained from that of $T[i_2..(i-1)]$ in amortized $O(1)$ time. However, the $O(i_3 - i_2)$-time method in [11] is not an online method since it might need to look at symbols $T[k]$ with $k > i$, i.e., symbols which have not been received yet. If we modify the method so that it never looks past the current position i then it may have to update $\Omega(i_3 - i_2)$ longest common prefix-values when it reads a new symbol for $\Omega(i_3 - i_2)$ iterations, and then the amortized update time will not be $O(1)$ per iteration. Secondly, since the algorithm does not know the input alphabet or even the values of $|\Sigma_n|$ or n in advance, it seems difficult to predict a suitable hash table size or hashing function for any new internal node created when constructing the suffix tree for $T[1..i]$ from the suffix tree for $T[1..(i-1)]$ in the main loop. On the other hand, in case the hash tables are updated or rebuilt during the algorithm's execution ([3] does not mention if this is what their algorithm is supposed to do), it is important to make sure that these operations are not too costly while the resulting hash tables still guarantee that not too many collisions between symbols of the current input alphabet will occur. We do not see how to do this within the required time bound.

1.4. *Our results*

We first present an efficient algorithm for the online squarefree recognition problem over arbitrarily ordered alphabets. The algorithm reads the successive characters of T until a square has been formed, then reports the occurrence of this square and stops. Its running time is $O(n \log n)$, where n is the ending position in T of the square; in other words, if n is the smallest integer such that $T[1..n]$ contains a square, our algorithm correctly determines whether $T[1..h]$ contains a square after reading $T[h]$ for every $h \in \{1, 2, \ldots, n\}$. Our algorithm is based on the algorithm of Leung, Peng, and Ting [10], but is faster. Our improvement comes from a speedup for ordered alphabets of the key subroutine named `DHangSq` in [10].

Next, we show how to apply the new speedup technique to the general alphabet case. We obtain an $O(n \cdot (|\Sigma_n| + \log n))$-time algorithm for general alphabets, where $|\Sigma_n|$ is the number of distinct symbols in $T[1..n]$. This is asymptotically faster than the original algorithm of Leung, Peng, and Ting [10] if $|\Sigma_n|$ is small compared to n (more precisely, if $|\Sigma_n| = o(\log^2 n)$). Note that long squarefree strings are possible even for small alphabets; in fact, a classical result by Thue [13] states that a constant alphabet of size 3 suffices to create infinitely long squarefree strings [11, 12].

Finally, we give a very simple algorithm for the dynamic squarefree recognition problem. It works for general alphabets and uses $O(n)$ time per update, where n is the current length of T.

The table below summarizes our results in this paper for the online and dynamic versions of the squarefree recognition problem.

	Online version	Dynamic version
Ordered alphabet	$O(n \log n)$ time (Theorem 2, Section 2.4)	$O(n)$ time per update (Theorem 3, Section 3)
General alphabet	$O(n \cdot (\lvert\Sigma_n\rvert + \log n))$ time (Corollary 1, Section 2.5)	$O(n)$ time per update (Theorem 3, Section 3)

2. An Efficient Algorithm for Online Squarefree Recognition over Ordered Alphabets and Small General Alphabets

In this section, we present a fast algorithm for the online squarefree recognition problem for arbitrarily ordered alphabets and small general alphabets.

2.1. *LPT: The algorithm of Leung, Peng, and Ting*

First, we briefly review the algorithm of Leung, Peng, and Ting [10], henceforth referred to as LPT. Please refer to [10] for details as well as correctness proofs for their algorithm.

Algorithm LPT is listed in Fig. 1. It reads the string T one character at a time, starting with $T[1]$. After reading a new position h, LPT immediately checks if $T[1..h]$ contains a square; if so then it reports the square and stops. Otherwise, $T[1..h]$ is squarefree, and the algorithm proceeds to read the character at the next

```
Algorithm LPT:
For $h \in \{1, 2, \ldots\}$, after reading $T[h]$, do the following:
    if there is a square in $T[(h-3)..h]$, or any of the active DHangSq$(i,j)$
    processes detects a square in $T[1..h]$, then report it and stop.
    $j = h$; $\ell = 1$;
    while ($j \geq 2^\ell$) do
        if $j = q \cdot 2^\ell$ for some integer $q$ then
            $i = \max\{1,\ q \cdot 2^\ell - 4 \cdot 2^\ell + 1\}$;
            start DHangSq$(i,j)$;
        $\ell = \ell + 1$;
```

Fig. 1. The algorithm of Leung, Peng, and Ting [10].

position from T. To efficiently do the checking, LPT makes use of a procedure called `DHangSq`(i,j) which solves the following subproblem: for every $h \in \{(j+1), (j+2), \ldots, (2j-i+1)\}$, after $T[h]$ is read, determine if T has a square ending at position h whose first half lies entirely in the interval $T[i..j]$ (such a square is said to be "hanging in $T[i..j]$"). When LPT reaches certain values of h, it starts a new `DHangSq` process so that at any point of its execution, it will have a number of active `DHangSq`(i,j) processes running, for various values of i and j with $i < j$.

To be more precise, for any $i < j$ and positive integer ℓ, the pair (i,j) is called a *level-ℓ pair* if there exists an integer q such that $j = q \cdot 2^\ell$ and $i = \max\{1,\ q \cdot 2^\ell - 4 \cdot 2^\ell + 1\}$. (In this case, $j - i \leq 4 \cdot 2^\ell - 1$.) Whenever LPT reaches a position j of the form $j = q \cdot 2^\ell$ where ℓ and q are positive integers, it starts a `DHangSq`(i,j) process for the level-ℓ pair (i,j), where $i = \max\{1, j - 4 \cdot 2^\ell + 1\}$, which will remain active until reaching position $2j - i + 1$ or a square has been detected. See Fig. 2 for an example.

For any `DHangSq`(i,j) process started by LPT, say that it is *on level ℓ* if (i,j) is a level-ℓ pair. Here, we make the following crucial observation which will be used in Section 2.4:

Lemma 1 *At any point during the execution of LPT, there are at most four active* `DHangSq` *processes on each level.*

Proof. Suppose LPT has read $T[h]$ but not yet $T[h+1]$. Consider any level $\ell \leq \log h$. We claim that there are always at most four active `DHangSq`(i,j) processes on level ℓ. Let a be the largest multiple of 2^ℓ which is less than h, and write $a = q \cdot 2^\ell$, i.e., $q \cdot 2^\ell < h \leq (q+1) \cdot 2^\ell$. For any $j \in \{2^\ell,\ 2 \cdot 2^\ell,\ \ldots,\ (q-4) \cdot 2^\ell\}$, the `DHangSq`$(i,j)$ process on level ℓ started at position j will have finished at position $2j - i + 1 = j + (j - i + 1) \leq (q-4) \cdot 2^\ell + (4 \cdot 2^\ell) = q \cdot 2^\ell < h$. Therefore, the only `DHangSq`(i,j) processes on level ℓ which may possibly be active at the time when $T[h]$ is read are those that were started for $j \in \{(q-3) \cdot 2^\ell,\ (q-2) \cdot 2^\ell,\ (q-1) \cdot 2^\ell,\ q \cdot 2^\ell\}$.

In case $h < (q+1) \cdot 2^\ell$, no new `DHangSq`(i,j) process needs to be started and the claim follows directly. Hence, assume $h = (q+1) \cdot 2^\ell$. If $q \leq 3$ then at most three `DHangSq` processes on level ℓ were started previously, and the claim follows.

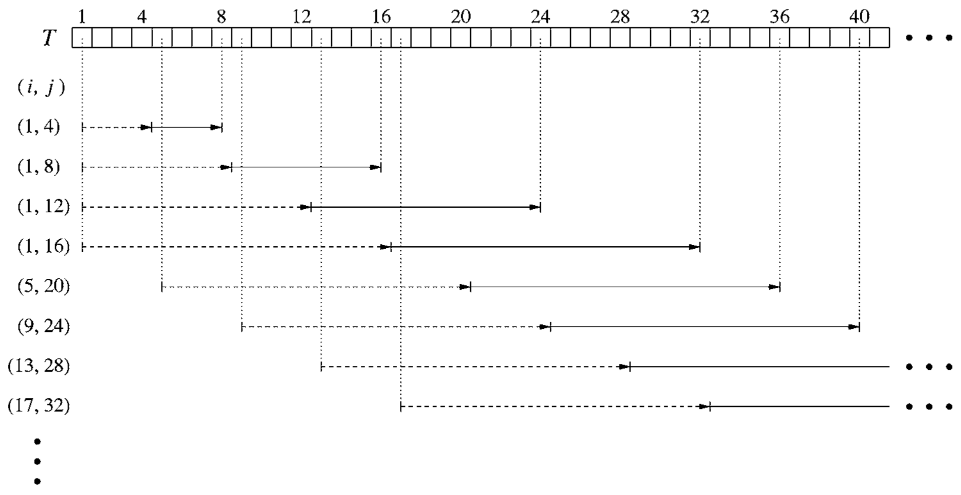

Fig. 2. The scope of $\texttt{DHangSq}(i, j)$ for some level-2 pairs. Each such $\texttt{DHangSq}(i, j)$ process is started right after reading $T[j]$, where j is a multiple of 2^2 and $i = \max\{1, j - 4 \cdot 2^2 + 1\}$, and remains active until reaching $T[2j - i + 1]$ or a square has been detected.

If $q > 3$, observe that the $\texttt{DHangSq}(i, j)$ process on level ℓ which was started at position $j = (q-3) \cdot 2^\ell$ ends at position $j + (j - i + 1) \leq (q-3) \cdot 2^\ell + (4 \cdot 2^\ell) = h$, where in case of equality, the new DHangSq process is started after the old one terminates according to the description of LPT. Thus, there are always at most four active $\texttt{DHangSq}(i, j)$ processes on level ℓ. □

The analysis in [10] of Algorithm LPT can be summarized and expressed as:

Theorem 1 [10] *Suppose n is the smallest integer such that $T[1..n]$ contains a square. For every $h \in \{1, 2, \ldots, n\}$, LPT correctly determines whether $T[1..h]$ contains a square after reading $T[h]$. The total running time of LPT is $\sum_{\ell=1}^{\lceil \log n \rceil} O(\frac{n}{2^\ell}) \cdot t(\ell)$, where $t(\ell)$ is the running time of $\texttt{DHangSq}(i, j)$ for a level-ℓ pair (i, j).*

Leung, Peng, and Ting [10] described how to implement $\texttt{DHangSq}(i, j)$ for general alphabets to run in $O((j - i) \cdot \log(j - i))$ time, i.e., $t(\ell) = O(2^\ell \cdot \ell)$ above. Using this implementation, it follows from Theorem 1 that the total running time of LPT is $O(n \log^2 n)$.

2.2. *Speeding up DHangSq*

Recall that $\texttt{DHangSq}(i, j)$ needs to solve the following problem: for every $h \in \{(j+1), (j+2), \ldots, (2j - i + 1)\}$, after $T[h]$ is read, determine if T has a square ending at position h whose first half lies entirely in the interval $T[i..j]$. Section 4 in [10] shows that this problem can be reduced to the following problem at an additional cost of $O(j - i)$ time, where the new parameter k is equal to $j - i + 1$:

The minimum suffix-centers checking problem (MSCC):

Let A be a given string of length k and let L be a given list of k pairs of integers of the form $(1, e(1)), (2, e(2)), \ldots, (k, e(k))$, where for each $s \in \{1, 2, \ldots, k\}$ it holds that $1 \leq s \leq e(s) \leq k$. Next, let B be a string of length k which arrives online, one character at a time. Return the smallest possible $h \in \{1, 2, \ldots, k\}$ for which there is a pair $(s, e(s))$ in L such that $A[s..e(s)]$ is equal to $B[1..h]$; if no such h exists then return *fail.*

This means that if we could solve MSCC in $O(k)$ time then we could improve the running time of DHangSq and hence Algorithm LPT; see Theorem 1. In particular:

Lemma 2 *If we have an $O(k)$-time algorithm for MSCC then $t(\ell) = O(2^{\ell})$ in Theorem 1.*

2.3. *Solving MSCC for integer alphabets*

Here, we give an algorithm for solving MSCC in $O(k)$ time under the additional constraint that A is a string over an integer alphabet $\{1, 2, \ldots, m\}$ with $m \leq k$. (In the next subsections, we show how to deal with this extra constraint for ordered alphabets and small general alphabets by applying an input alphabet mapping technique.)

The basic idea is to check, after receiving each $B[h]$, if the string $B[1..h]$ received so far equals the first $e(s) - s + 1$ characters of the suffix $A[s..k]$ for any $s \in \{1, 2, \ldots, k\}$. We use a suffix tree[b] to encode A so that we can match the successive characters of B to all of $A[1..e(1)]$, $A[2..e(2)]$, $\ldots$, $A[k..e(k)]$ efficiently until enough characters match or no more matches are possible. For convenience, let s for any $s \in \{1, 2, \ldots, k\}$ also refer to the unique leaf that represents the suffix $A[s..k]$ in the suffix tree for A. We rely on the following property of suffix trees:

Fact 1 (See, e.g., Sections 5.2–5.3 in [7].) Let p be any node or split point on an edge in the suffix tree $\mathcal{T}_A$ for A, and let $C(p)$ be the concatenation of all edge labels along the path in $\mathcal{T}_A$ starting at the root and ending at p. For any leaf s in $\mathcal{T}_A$, it holds that s is a descendant of p or equal to p if and only if the first $|C(p)|$ characters of suffix $A[s..k]$ are identical to $C(p)$.

We now describe the details of our algorithm. It consists of two phases. Phase I (the preprocessing phase) is performed after receiving A and L but before receiving any characters from B, and Phase II (the matching phase) is performed while receiving B.

Phase I (preprocessing phase): Construct the suffix tree $\mathcal{T}_A$ for A. Augment $\mathcal{T}_A$ with additional information as follows. For every edge f in $\mathcal{T}_A$, define $v(f)$ as the minimum value of $e(s) - s + 1$ taken over all leaves s belonging to the subtree of $\mathcal{T}_A$

[b]The *suffix tree* for a string A is a compacted trie storing all the suffixes of A. See, e.g., [6, 7] for a formal definition of a suffix tree.

below f. Obtain and store $v(f)$ for every edge f in $\mathcal{T}_A$ by a bottom-up traversal of $\mathcal{T}_A$.

Phase II (matching phase): Check if $B[1..h]$ equals $A[s..e(s)]$ for any $(s, e(s)) \in L$ for successive values of $h \in \{1, 2, \ldots, k\}$ with the following method. Start at the root of $\mathcal{T}_A$ and traverse edges according to the received $B[1], B[2], \ldots$, where to traverse an edge that represents x characters we require that its label is equal to x consecutive characters from B, until either the total number of matched characters h reaches the value $v(f)$ for the edge f being traversed (success; return h) or the current character $B[h]$ does not match any edge label at the current position in $\mathcal{T}_A$ (failure; return *fail*).

Finally, we prove the correctness and analyze the time complexity of the above algorithm.

Correctness: Suppose the algorithm has just received $B[h]$, where $h \in \{1, 2, \ldots, k\}$. Let p be the node or split point on an edge in $\mathcal{T}_A$ reached by starting at the root and traversing edges according to $B[1..h]$, and let f be the lowest edge belonging to this path. (If such a path does not exist then the algorithm will correctly return *fail*.) First of all, according to the description above, the algorithm returns the current value of h if and only if $v(f) = h$. Next, since the algorithm did not terminate previously, there is no leaf s descending from p with $e(s) - s + 1 < h$. Thus, by the definition of $v(f)$, we have $v(f) = h$ if and only if there exists a leaf s descending from p satisfying $e(s) - s + 1 = h$. Lastly, by Fact 1, each leaf s descending from p represents a suffix $A[s..k]$ whose h first characters equal the string $B[1..h]$ received so far. To conclude, there exists a suffix $A[s..k]$ whose first $e(s) - s + 1$ characters are identical to $B[1..h]$ if and only if the algorithm returns h.

Running time: For Phase I, we use the algorithm of Farach-Colton *et al.* [6] for constructing suffix trees over integer alphabets to build $\mathcal{T}_A$ in $O(k)$ time. Next, the bottom-up traversal to compute $v(f)$ for every edge f in $\mathcal{T}_A$ takes $O(k)$ time. Then, in Phase II, the total time used for finding which outgoing edge to follow in $\mathcal{T}_A$ from an internal node is upper-bounded by the total number of edges in $\mathcal{T}_A$ because each outgoing edge of $\mathcal{T}_A$ is examined at most once; since $\mathcal{T}_A$ has $O(k)$ edges, these computations take $O(k)$ time. The rest of the computations in Phase II take $O(1)$ time per read character and the algorithm reads at most k characters from B. Therefore, the total running time of our algorithm is $O(k)$.

We thus have:

Lemma 3 *MSCC for integer alphabets can be solved in $O(k)$ time.*

2.4. *An efficient algorithm for online squarefree recognition over ordered alphabets*

Our solution for the subproblem MSCC in Section 2.3 requires the alphabet of the input string A to be an integer alphabet $\{1, 2, \ldots, m\}$, where $m \leq |A|$. Therefore, we will modify Algorithm LPT so that before starting `DHangSq` for any required pair of indices (i, j), it translates $T[i..j]$ into an equivalent string $T''_{i..j}$ over the alphabet $\{1, 2, \ldots, (j - i + 1)\}$. Similarly, when a symbol is read from T, the algorithm will translate that symbol into the corresponding integer alphabet for each currently active DHangSq for checking. For this purpose, the modified LPT will translate the input string T online to a string T' over a growing integer alphabet that is subsequently used to construct all the necessary $T''_{i..j}$-strings. In this section, we demonstrate how these extra steps can be performed without increasing the overall asymptotic running time of LPT. Below, the new version of LPT is referred to as LPT*.

For any positive integer h, denote the set of symbols occurring in $T[1..h]$ by Σ_h. By our assumptions, each Σ_h is ordered; except for this fact, we have no information about the alphabet of T in advance.

Translating T to T': As the characters of T arrive online, LPT* first translates them to obtain an equivalent string T' such that for every positive integer h, the alphabet of $T'[1..h]$ is exactly $\{1, 2, \ldots, |\Sigma_h|\}$. To do this, it stores the distinct symbols read from T so far in a balanced binary search tree $\mathcal{B}$ with the following method: Let c be a counter, initially set to 0. When position h of T is read, first check if the symbol $T[h]$ is already contained in $\mathcal{B}$: if no then insert $T[h]$ into $\mathcal{B}$, increment c by one, and associate the integer c with this new symbol; if yes then simply retrieve its associated integer. In both cases, let $T'[h]$ be the integer associated with $T[h]$. Since the number of nodes in $\mathcal{B}$ while reading $T[1..n]$ is always less than or equal to $|\Sigma_n|$ and because Σ_n is ordered, the total time used to translate $T[1..n]$ to $T'[1..n]$ is $O(n \log |\Sigma_n|) = O(n \log n)$.

Translating T' to $T''_{i..j}$: Next, whenever LPT* starts DHangSq for some pair of indices (i, j), it also constructs an injective mapping $f_{i..j}$ from the set of symbols occurring in $T'[i..j]$ to the set $\{1, 2, \ldots, (j - i + 1)\}$ and applies $f_{i..j}$ to each position in $T'[i..j]$ to obtain a string $T''_{i..j}$ over $\{1, 2, \ldots, (j - i + 1)\}$. Furthermore, for each such (i, j), until DHangSq(i, j) is terminated, LPT* keeps track of $f_{i..j}$ so that it can translate online the characters in $T'[(j + 1)..(2j - i + 1)]$ to the same alphabet.

The mapping $f_{i..j}$ is implemented as an array $F_{i..j}$ such that for any $x \in \{1, 2, \ldots, j\}$ occurring as a symbol in $T'[i..j]$, the entry x in $F_{i..j}$ contains the value $f_{i..j}(x)$; the other entries of $F_{i..j}$ are left undefined. For efficiency reasons explained below, LPT* will reuse the array $F_{i..j}$ for a terminated DHangSq, and therefore also associates a "timestamp" of the form (i, j) with each entry of $F_{i..j}$ to directly tell whether an entry is valid or contains old information. Suppose LPT* needs to start a new DHangSq(i, j) for some i immediately after reading a character $T[j]$ and translating it to $T'[j]$. Let c be a counter, initially set to 0, and scan the substring $T'[i..j]$. For each $s \in \{i, (i + 1), \ldots, j\}$, first check if entry $T'[s]$ in $F_{i..j}$ already has been set by checking its timestamp: if no then increment c by one, set entry $F_{i..j}(T'[s])$ to c, and update the timestamp of $f_{i..j}(T'[s])$. Clearly, this takes only $O(j - i)$ time.

By Lemma 1, there are at most four active DHangSq(i, j) processes on each level at any time, so we only need to keep track of four $F_{i..j}$-arrays for every level reached.

This means that we can reuse the array $F_{i..j}$ used for storing $f_{i..j}$ after `DHangSq`(i,j) terminates to store $f_{i'..j'}$ for another `DHangSq`(i',j') on the same level. By using timestamps, we do not need to reinitialize all the positions of the array. However, note that for any such (i',j'), the array $F_{i..j}$ might not be large enough to store j' entries. To handle this issue, whenever LPT* reaches a position of the input string which equals a power of two, we let it double the size of every existing $F_{i..j}$, (e.g., for each existing $F_{i..j}$, initialize a new array with twice as many entries and copy the contents of the old $F_{i..j}$ into the first half of the new array). Thus, after reading h characters from T, every $F_{i..j}$ contains $O(h)$ entries.

Supposing that LPT* terminates after reading $T[1..n]$ for some positive integer n, the time needed for all these operations is bounded by $\sum_{r=1}^{\lfloor \log n \rfloor} O(r) \cdot 4 \cdot O(2^r) = O(n \log n)$. (LPT* doubles the arrays after reaching position 2^r of T for every integer r, i.e., not more than $\lfloor \log n \rfloor$ times. Every time, there are $O(r)$ levels and at most four active `DHangSq` on each level, and the doubling of an array uses time proportional to the number of positions read from T so far.)

Total running time of LPT*: Suppose n is the smallest integer such that $T[1..n]$ contains a square. The total running time of LPT* is equal to the time needed to do all the string translation operations to integer alphabets plus the running time of LPT using the faster `DHangSq` for integer alphabets. By the above, the translation operations take a total of $O(n \log n)$ time. By Theorem 1, the running time of LPT is given by $\sum_{\ell=1}^{\lceil \log n \rceil} O(\frac{n}{2^\ell}) \cdot t(\ell)$, and according to Lemmas 2 and 3, we have $t(\ell) = O(2^\ell)$. Adding everything together yields:

Theorem 2 *The online squarefree recognition problem for arbitrarily ordered alphabets can be solved in $O(n \log n)$ time, where n is the ending position of the first square.*

2.5. *An efficient algorithm for online squarefree recognition over small general alphabets*

For general alphabets, we can use the algorithm in Section 2.4 after replacing the method that translates the input string T into an equivalent string T' over an integer alphabet. (Translating T' to $T''_{i,j}$ is then done in the same way as before.) The new method is similar to the previous one, but instead of storing the distinct symbols read from T in a balanced binary search tree, we just store the position of the first occurrence of each distinct symbol in a list B so that $|B|$ will always equal the cardinality of the current Σ_h. More precisely: Initially, let B be the empty list. After reading each $T[h]$, scan the current B to check if $T[h]$ is equal to $T[x]$ for any x in B. If yes, let $T'[h]$ be the index in B of this x. If no, let $T'[h]$ be $|B|+1$, and insert the integer h at the end of B.

The time used to translate $T[1..n]$ to $T'[1..n]$ is thus $O(n \cdot |\Sigma_n|)$, and as before, the rest of the operations take $O(n \log n)$ time.

Corollary 1 *The online squarefree recognition problem for general alphabets can be solved in $O(n \cdot (|\Sigma_n| + \log n))$ time, where n is the ending position of the first* *square and $|\Sigma_n|$ is the number of distinct symbols in $T[1..n]$.*

According to Corollary 1, the running time of the new method is asymptotically less than that of the original LPT algorithm if the cardinality of Σ_n is $o(\log^2 n)$.

3. An Algorithm for Dynamic Squarefree Recognition over General Alphabets

We now present a simple algorithm for the dynamic squarefree recognition problem over general alphabets. Its input is an initially squarefree string T, followed by a series of updates to T. After each update, our algorithm checks if the modified T is still squarefree, and if not, reports this fact and stops.

Let n be the current length of T. The allowed updates are of the following form:

- `replace(q,'x')`, where $1 \leq q \leq n$, which means "replace the symbol at position q of T by the symbol x". The modified T has length n.
- `insert(q,'x')`, where $1 \leq q \leq n+1$, which means "insert the symbol x into T at position q" (if $q = n+1$ then append x to the end of T). The modified T has length $n+1$.
- `delete(q)`, where $1 \leq q \leq n$, which means "delete the symbol at position q from T". The modified T has length $n-1$.

The key observation is that after each update, any newly formed square in T must include the position q, which limits the total number of comparisons we need to make.

For any two positions i, j of T with $1 \leq i, j \leq n$, define $LCSu^{-1}(i,j)$ as the longest common suffix of $T[1..(i-1)]$ and $T[1..(j-1)]$ and $LCPr^{+1}(i,j)$ as the longest common prefix of $T[(i+1)..n]$ and $T[(j+1)..n]$. We have the following.

Lemma 4 *Let T be a string of length n. For any $q \in \{1,2,\ldots,n\}$, there is a square in T which includes position q if and only if there exists a $q' \in \{1,2,\ldots,n\}$ with $q' \neq q$ such that $T[q] = T[q']$ and $|LCSu^{-1}(q,q')| + |LCPr^{+1}(q,q')| + 1 \geq |q-q'|$.*

Proof. $\Longrightarrow$) Suppose T contains a square $S = T[p..(p+2k-1)]$, where $p \leq q \leq p+2k-1$. Define q' as $q' = q+k$ if $q \leq p+k-1$ and as $q' = q-k$ if $p+k \leq q$. It is easy to see that $q \neq q'$, $T[q] = T[q']$, and $|LCSu^{-1}(q,q')| + |LCPr^{+1}(q,q')| \geq k-1 = |q-q'|-1$.

$\Longleftarrow$) Suppose there exists a $q' \in \{1,2,\ldots,n\}$ with $q' \neq q$ such that $T[q] = T[q']$ and $|LCSu^{-1}(q,q')| + |LCPr^{+1}(q,q')| + 1 \geq |q-q'|$. Assume without loss of generality that $q < q'$ (the case $q > q'$ is symmetric). Define $p = q - |LCSu^{-1}(q,q')|$ and $r = q' - |LCSu^{-1}(q,q')|$. By the definition of $LCSu^{-1}$, we have $T[p..(q-1)] = T[r..(q'-1)]$. Observe that $p \leq q$ and $p < r$. There are two possibilities:

- $q < r$: We rewrite the inequality above as $|LCPr^{+1}(q,q')| \geq -q+r-1$, which yields $T[(q+1)..(r-1)] = T[(q'+1)..(q'-q+r-1)]$ by the definition of $LCPr^{+1}$.

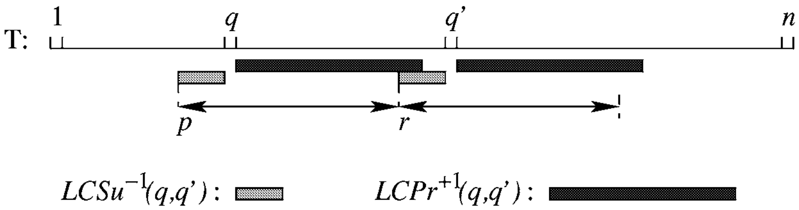

Fig. 3. Illustrating the second part of the proof of Lemma 4 for the case $q < r$.

After concatenation of the respective substrings, we obtain $T[p..(r-1)] = T[r..(q'-q+r-1)]$, i.e., T contains a square, and moreover, this square includes position q since $p \leq q \leq r-1$. See Fig. 3 for an illustration.

- $r \leq q$: We know that $T[p..q] = T[r..q']$. Thus, $T[(p+i)..q] = T[(r+i)..q']$ for any $i \in \{0,1,\ldots,(q-p)\}$, and in particular, $T[(p+q-r+1)..q] = T[(r+q-r+1)..q']$ since $1 \leq q-r+1 \leq q-p$, which gives us $T[(q-(q'-q)+1)..q] = T[(q+1)..q']$. Hence, T contains a square which includes position q.

□

Now, to determine if T contains a square after performing an update of the form `replace(q,'x')`, `insert(q,'x')`, or `delete(q)`, we apply Lemma 4. The resulting algorithm is as follows:

Algorithm Check-if-still-squarefree(q): Let n be the length of the modified T, and for each $q' \in \{1,2,\ldots,n\}$ with $q' \neq q$, check if the two conditions $T[q] = T[q']$ and $|LCSu^{-1}(q,q')| + |LCPr^{+1}(q,q')| + 1 \geq |q-q'|$ hold. If yes for some q', then T has at least one square; report "T is no longer squarefree" and stop. If no for all q', then T is still squarefree (this is because any new square in T must include position q by the key observation above).

To implement Algorithm Check-if-still-squarefree, we use an $O(n)$-time method to obtain $|LCSu^{-1}(q,q')|$ and $|LCPr^{+1}(q,q')|$ for all $q' \in \{1,2,\ldots,n\}$ with $q' \neq q$ as follows. First create a string $S = T[(q+1)..n] \circ T[1..(q-1)]$, where $\circ$ denotes concatenation, of length $n-1$. Then, compute the length of the longest common prefix of $S[j..(n-1)]$ and $S[1..(n-q)]$ for all $j \in \{1,2,\ldots,(n-1)\}$ in $O(n)$ total time based on the method from Section 2 in [11] (alternatively, see Section 1.4 in [7]) for computing the length of the longest common prefix of $S[j..(n-1)]$ and $S[1..(n-1)]$ for every j. Clearly, this will give us all the values of $|LCPr^{+1}(q,q')|$ for $q' \neq q$. To compute the $|LCSu^{-1}(q,q')|$-values, we repeat the above steps but create $S = T[1..(q-1)]^R \circ T[(q+1)..n]^R$ instead, where A^R means the reverse of string A.

Theorem 3 *The dynamic squarefree recognition problem for general alphabets can be solved in $O(n)$ time per update, where n is the current length of the string.*

Algorithm Check-if-still-squarefree can be extended to also return a list of all newly formed squares' starting and ending positions.

4. Concluding Remarks

Some interesting open questions are:

- Can the running time of the LPT algorithm [10] be reduced to $O(n \log n)$ for general alphabets?
- Can the online squarefree recognition problem for *constant* alphabets be solved in $O(n)$ time?
- If we allow $O(n \log n)$ time preprocessing, can the query time after each update for the dynamic version of the problem be reduced to $o(n)$?
- How efficiently can the online and dynamic versions of the *cube* (and higher orders of repetitions) detection problem be solved?

Acknowledgment

We thank Kunihiko Sadakane for some helpful discussions.

References

1. A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22(3):297–315, 1983.
2. D. Breslauer. *Efficient String Algorithmics.* PhD thesis, Columbia University, 1992.
3. G.-H. Chen, J.-J. Hong, and H.-I. Lu. An optimal algorithm for online square detection. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching* (CPM 2005), volume 3537 of *Lecture Notes in Computer Science*, pages 280–287. Springer-Verlag, 2005.
4. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
5. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
6. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.
8. V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 12(1):32–44, 1992.
9. J. H. M. Lee, H.-F. Leung, and H. W. Won. Performance of a comprehensive and efficient constraint library based on local search. In *Proceedings of the 11th Australian Joint Conference on Artificial Intelligence*, pages 191–202, 1998.
10. H.-F. Leung, Z. Peng, and H.-F. Ting. An efficient online algorithm for square detection. In *Proceedings of the 10th International Computing and Combinatorics Conference* (COCOON 2004), volume 3106 of *Lecture Notes in Computer Science*, pages 432–439. Springer-Verlag, 2004.
11. M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
12. M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F 12 of *NATO ASI Series*, pages 271–278. Springer-Verlag, 1985.

13. A. Thue. Über unendliche Zeichenreihen. *Norske Videnskabers Selskabs Skrifter, Mat.-Nat. Kl.*, 7:1–22, 1906.
14. J. H. Y. Wong and H.-F. Leung. Solving fuzzy constraint satisfaction problems with fuzzy GENET. In *Proceedings of the 10th IEEE International Conference on Tools with Artificial Intelligence*, pages 184–191, 1998.