Online and Dynamic Recognition of Squarefree Strings

Jesper Jansson and Zeshan Peng

Department of Computer Science,
The University of Hong Kong,
Pokfulam Road, Hong Kong
{jjansson, zspeng}@cs.hku.hk

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. Its running time is $O(n \log n)$, where n is the ending position of the first square, which matches the running times of the fastest known algorithms for the analogous offline problem. We also present a very 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 contains a square and if so, report it and stop.

1 Introduction

A classic problem in computer science is to determine whether a given string T contains a *square*, defined as a substring of T which can be split into two identical parts. Since a square is one of the simplest possible types of patterns in a string, methods for detecting squares efficiently have a wide range of applications in diverse areas such as string algorithms and combinatorics [2, 6, 8, 9, 11, 16, 18], automata and formal language theory [6, 12], data compression [4, 8, 17], coding theory [4], and computational biology [3, 5, 11].

Many people have studied this problem and its variants (see, e.g., [1, 2, 4, 6–9, 11, 12, 15–18] and the numerous references therein). However, previous work has mainly focused on the *offline* version in which the entire string T is available at once. This offline property is not desirable in certain applications. For example, suppose we need to determine whether a string of one million characters contains a square. If we use an offline algorithm, we will have to scan through all the one million characters, which may be very inefficient if a square appears at the very beginning of the string. In some applications, such as online data compression, the offline property is even unacceptable; we need to be able to report a square whenever a new character arrives. The online squarefree recognition problem is also motivated by the local search method for solving the constraints satisfaction problem in [13, 14, 19]; 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 [15].

Our main result is an efficient algorithm for the online squarefree recognition problem over an arbitrarily ordered alphabet. This is a reasonable assumption for most applications because when the symbols are encoded as binary numbers in a computer, this will induce a lexicographical ordering among them. Our algorithm is based on the work of Leung, Peng, and Ting [15]. We also introduce and study a dynamic version of the problem.

1.1 Problem Definitions

For any string T, let $|T|$ be the length of T. For every $1 \le i \le j \le |T|$, denote the substring of T starting at position i and ending at position j by $T[i..j]$, and define $T[i] = T[i..i]$. A substring of the form $T[i..(i + 2k - 1)]$ 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.*

We distinguish between the *offline*, *online*, and *dynamic* versions of the squarefree recognition problem. In the offline version, the 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 it 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 on position q of T by the symbol `x`", and the objective is to decide after each update if the resulting T contains a square and if so, report it and stop. In this paper, we also consider a combination of the online and dynamic versions of the problem that also allows updates of the form "append the symbol `x` to the end of T".

The alphabet of the input string determines how efficiently the various 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.* [1] 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.*

[1] As an example to illustrate the difference between general and ordered alphabets, consider the element uniqueness problem which has a lower bound of $\Omega(n^2)$ for general alphabets but admits an $O(n \log n)$-time solution for ordered alphabets (see [4]).

1.2 Previous Results

For the offline and general alphabet case, Main and Lorentz [17] 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 [17]. (For the offline and non-general alphabet case, other efficient algorithms for finding squares were presented earlier in [1] and [7].) However, it is still not known if the lower bound $\Omega(n \log n)$ for determining squarefreeness holds for ordered alphabets. For the offline and constant alphabet case, there exist algorithms that determine if T is squarefree in optimal $O(n)$ time [8, 18]. Parallel algorithms for finding squares offline have also been developed (see [4]).

For the online case, the only previously known result is the algorithm by Leung, Peng, and Ting [15] for general alphabets 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 3.1.

1.3 Our Results

We first present an algorithm for the online squarefree recognition problem over arbitrarily ordered alphabets. It reads the successive characters of T until a square has been formed, then reports the occurrence of this square and stops. The 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 $h \in \{1, 2, \ldots, n\}$. Note that this matches the running times of the fastest known offline algorithms for determining squarefreeness of strings over ordered alphabets [1, 7, 17].

Next, we give a very simple algorithm for the dynamic version of the squarefree recognition problem. It works for general alphabets and uses $O(n)$ time per update, where n is the length of the input string. The algorithm can easily be extended to also solve the combination of the online and dynamic versions of the problem in which every update either modifies an existing character or adds a new character to the end of T.

The table below summarizes our results.

Alphabet type	Online algorithm	Dynamic algorithm	Online + dynamic
General	$O(n \log^2 n)$ (See [15])	$O(n)$ per update (Theorem 3, Section 4)	$O(n)$ per update (Theorem 4, Section 4)
Ordered	$O(n \log n)$ (Theorem 2, Section 3)	$O(n)$ per update (Theorem 3, Section 4)	$O(n)$ per update (Theorem 4, Section 4)

2 Preliminaries

2.1 Suffix Trees

Let A be a string of length k. A *suffix of* A is a substring of A of the form $A[x..k]$, where $x \in \{1, 2, \ldots, k\}$. A *suffix tree for* A (see, e.g., [10, 11]) is a rooted tree with $O(k)$ nodes which represents each suffix of A as a unique path from the root to a leaf. Every edge in the suffix tree for A encodes a particular substring of A whose starting and ending positions in A are specified by two integers which label that edge. For any two leaves x and y, the unique path from the root to the lowest common ancestor of x and y encodes the longest common prefix of the two suffixes represented by x and y.

3 An Efficient Online Squarefree Recognition Algorithm for Arbitrarily Ordered Alphabets

In this section, we present an algorithm for the online squarefree recognition problem for arbitrarily ordered alphabets. Our algorithm is based on the algorithm of Leung, Peng, and Ting [15] for the general alphabet case, but faster.

3.1 LPT: The Algorithm of Leung, Peng, and Ting

Here, we briefly review the algorithm of Leung, Peng, and Ting [15], henceforth referred to as LPT. LPT reports the first square in the online input string T in $O(n \log^2 n)$ time, where n is the position in T where the square ends.

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 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

```
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 running DHangSq$(i, j)$
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. Algorithm LPT

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 `DHangSq`(i,j) processes running (for various values of i and j). Refer to [15] for more details as well as correctness proofs for the algorithm.

For any $1 \le i \le j$, the pair (i,j) is called a *level-ℓ pair* if there exists an integer q such that $j = q \cdot 2^\ell$ and $i = \max\{1,\, q \cdot 2^\ell - 4 \cdot 2^\ell + 1\}$. (Hence, $j - i + 1 \le 4 \cdot 2^\ell$.) The analysis in [15] of Algorithm LPT can be summarized and expressed as:

Theorem 1. *[15] 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* `DHangSq`(i,j) *for a level-ℓ pair (i,j).*

Leung, Peng, and Ting [15] described how to implement `DHangSq`(i,j) for general alphabets to run in $O((j-i+1)\cdot\log(j-i+1))$ 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 $O(n \log^2 n)$.

3.2 Speeding Up `DHangSq`

Recall that `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 [15] shows that this problem can in fact be reduced to the following problem (stated slightly differently in [15]) at an additional cost of $O(j-i+1)$ time, where the parameter k in the new problem 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 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 $1 \le s \le e(s) \le k$. 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. More precisely:

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

3.3 Solving MSCC for Integer Alphabets

We now 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 $m \le k$.

(In the next section, we show how to deal with this extra constraint efficiently for ordered alphabets by using an input alphabet mapping technique.)

The main idea of our algorithm for MSCC for integer alphabets is to store the given A in a suffix tree T_A, and match the successive characters of B along a unique path from the root in T_A until either enough characters match so that $A[s..e(s)]$ equals a prefix of B for some s, or the current character of B fails to match any outgoing edge at the current position in T_A. Our algorithm consists of a preprocessing phase and a matching phase:

Phase I (Preprocessing Phase): Construct a suffix tree T_A for A. For convenience, let s for any $s \in \{1, 2, \ldots, k\}$ also refer to the leaf in T_A that represents the suffix $A[s..k]$. Augment T_A with additional information as follows. For every edge f in T_A, define $v(f)$ as the minimum value of $e(s) - s + 1$ taken over all leaves s belonging to the subtree of T_A below f (note that $v(f) \leq k$). Obtain and store $v(f)$ for every edge f in T_A by doing a bottom-up traversal of T_A.

Phase II (Matching Phase): For successive values of $h \in \{1, 2, \ldots, k\}$, check if $B[1..h]$ equals $A[s..e(s)]$ for any $(s, e(s)) \in L$ with the following method. Match the successive characters in B along the unique path in T_A starting at the root by following edges labeled by $B[1], B[2], \ldots$ (to traverse an edge in T_A that represents x characters, we need to match it to x characters from B) until either h reaches the value $v(f)$ for the edge f being traversed (success; return h), or the current character in B does not match any edge at the current position in T_A (failure; return *fail*).

Correctness: In Phase I, the algorithm builds a suffix tree T_A for A. In Phase II, the algorithm starts at the root of T_A and follows a path whose labels match the successive characters of B. Suppose that the algorithm has received $B[1..h]$ for any $h \in \{1, 2, \ldots, k\}$. By the properties of a suffix tree, the set of leaves descending from the current location in T_A encode all prefixes of suffixes (i.e., all substrings) of A having length h that are identical to the string $B[1..h]$ received so far. Now, if there is such a substring $A[s..(s + h - 1)]$ that also satisfies $(s, s + h - 1) \in L$, then the edge f being traversed will have $v(f) = h$, and since the length of the path from the root is exactly h, the algorithm will succeed and return h.

To see that the algorithm will stop for the smallest possible h, suppose $B[1..h] = A[s..(s + h - 1)]$ as well as $B[1..h'] = A[t..(t + h' - 1)]$ for some $h < h'$ and $(s, (s + h - 1))$, $(t, (t + h' - 1)) \in L$. Then the algorithm must have terminated after $B[1..h]$ has been processed because the corresponding path of length h in T_A from the root will have reached the lowest common ancestor of the two leaves s and t, and the edge f leading to that node satisfies the stopping condition $v(f) \leq \min\{h, h'\} = h$.

Running Time: To implement the algorithm above, we use the method of Farach-Colton *et al.* [10] for constructing suffix trees over integer alphabets to build T_A in $O(k)$ time. Next, the bottom-up traversal to compute $v(f)$ for every edge f in T_A takes $O(k)$ time. Then, in the matching phase, the total time for finding which outgoing edges to follow in T_A from internal nodes is upper-bounded by the number of edges in T_A since each edge is examined at most once; thus, these computations take $O(k)$ time. The rest of the computations in the matching phase 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 $O(k)$.

Lemma 2. *MSCC for integer alphabets can be solved in $O(k)$ time.*

3.4 LPT*: An Online Squarefree Recognition Algorithm for Arbitrarily Ordered Alphabets

Our solution for the subproblem MSCC in Section 3.3 requires the alphabet of the input string A to be an integer alphabet $\{1, 2, \ldots, m\}$, where $m \le |A|$. However, the input T to the online squarefree recognition problem for an arbitrarily ordered alphabet does not necessarily meet this requirement. 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 arbitrarily 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 a string T' such that for each positive integer h, the alphabet of $T'[1..h]$ is precisely $\{1, 2, \ldots, |\Sigma_h|\}$. To do this, it stores the distinct symbols read from T so far in a balanced binary search tree $\mathcal{B}$ and associates a unique integer with each symbol inserted into $\mathcal{B}$. Since the number of nodes in $\mathcal{B}$ while reading $T[1..h]$ is always less than or equal to h and because Σ_h is ordered, the total time used to translate $T[1..h]$ to $T'[1..h]$ is $O(h \log h)$.

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 $O(j - i + 1)$ time.

Next, for any `DHangSq`(i, j) process started by LPT*, say that it is *on level* ℓ if (i, j) is a level-ℓ pair. We make the following crucial observation:

Lemma 3. *At any point during the execution of LPT*, there are at most four active* `DHangSq` *processes on each level.*

Proof. Suppose LPT* has just read $T[h]$. Consider any level $\ell \leq \log h$. Let a be the largest multiple of 2^ℓ which is less than h, and write $a = q \cdot 2^\ell$, i.e., $q \cdot 2^\ell < h \leq (q + 1) \cdot 2^\ell$. If $q < 4$ then less than four `DHangSq` processes on level ℓ have been started and the lemma follows directly. Hence, assume $q \geq 4$. Each `DHangSq`(i, j) is active while at most $j - i + 1 = 4 \cdot 2^\ell$ positions of T are being read. This means that right after $T[h]$ is read, the only active `DHangSq`(i, j) processes on level ℓ are those that were started for $j \in \{(q-3)\cdot 2^\ell, (q-2)\cdot 2^\ell, (q-1)\cdot 2^\ell, q\cdot 2^\ell\}$. $\square$

By Lemma 3, we only need to keep track of four $F_{i..j}$ arrays for each level reached. This means 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 1 and 2, 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.*

4 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 a squarefree string T of length n, followed by a series of updates of the form $T[q] :=$ `'x'` (where $1 \leq q \leq n$) which means "replace the symbol on position q of T by the symbol x". After each update, our algorithm uses $O(n)$ time to check if the modified T contains a square, and if so, reports it and stops.

The key observation is that after each update $T[q] :=$ `'x'`, any newly formed square in T must include the position q along with a (possibly empty) substring ending immediately before q and a (possibly empty) substring starting immediately after 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. *Suppose that T is a squarefree string of length n and we perform an update $T[q] :=$ 'x', where $1 \leq q \leq n$. The resulting string T contains a square 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 the resulting T contains a square $S = T[p..(p+2k-1)]$. Then we know by the key observation above that $p \leq q \leq p+2k-1$. Define the *twin of* 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 $q < q'$. Define $p = q - |LCSu^{-1}(q,q')|$ and $r = q' - |LCSu^{-1}(q,q')|$. By the definition of $LCSu^{-1}$, we have $T[p..(q-1)] = T[r..(q'-1)]$. Next, we rewrite the inequality 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 $LCPr^{+1}$. Putting everything together, we have $T[p..(r-1)] = T[r..(q'-q+r-1)]$, i.e., T contains a square. See Fig. 2 for an illustration. The case $q > q'$ is symmetric. $\square$

Now, to determine if T contains a square after performing an update $T[q] :=$ `'x'`, apply Lemma 4. More precisely: for each $q' \in \{1, 2, \ldots, n\}$ with $q' \neq q$,

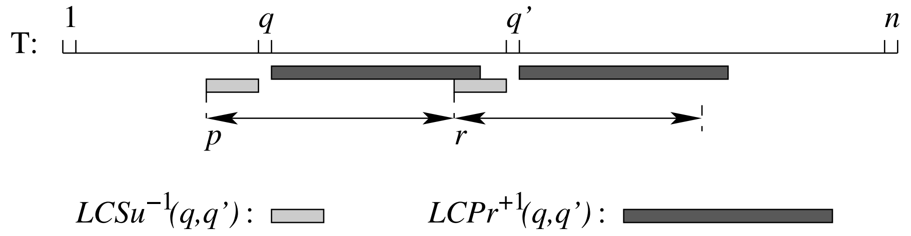

Fig. 2. Illustrating the second part of the proof of Lemma 4

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, then T contains a square; report it and stop. If no, then T is still squarefree.

To implement the above, we use an $O(n)$-time method to obtain the values of $|LCSu^{-1}(q, q')|$ 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, for all $j \in \{1, 2, \ldots, (n - 1)\}$, compute the length of the longest common prefix of $S[j..(n-1)]$ and $S[1..(n-q)]$ in $O(n)$ total time based on the method on p. 8 in [11] 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 length of the input string.*

We end this section by describing how the above algorithm can be extended to the *online dynamic* squarefree recognition problem that also allows characters to be appended to the current T. Given any update $T[q]$:= `'x'`, where $q \in \{1, 2, \ldots, (n + 1)\}$, if $1 \leq q \leq n$ then perform the same steps as above. If $q = n+1$ then position $n+1$ must be the endpoint of any possible newly formed square according to the key observation. In this case, calculate $|LCSu^{-1}(n+1, j)|$ for all $j \in \{\lceil \frac{n+1}{2} \rceil, \ldots, n\}$ and note that the resulting T contains a square if and only if $T[n+1] = T[j]$ and $|LCSu^{-1}(n+1, j)| \geq n-j$ for some j as in Lemma 4; use this fact to report any newly formed square. As above, the time needed for one update is $O(n)$, where n is the length of the current T. We obtain:

Theorem 4. *The online dynamic squarefree recognition problem for general alphabets can be solved in $O(n)$ time per update, where n is the current length of the input string.*

5 Concluding Remarks

We have presented an efficient algorithm for the online version of the squarefree recognition problem for arbitrarily ordered alphabets which runs in $O(n \log n)$

time. In comparison, the fastest known offline algorithms for determining if a string of length n over an ordered alphabet is squarefree [1,7,17] also run in $O(n \log n)$ time. Moreover, we have provided a simple algorithm for a dynamic version of the problem for general alphabets with $O(n)$ time per update.

Some interesting open questions are:

- Is the running time of our algorithm optimal, i.e., does there exist a lower bound of $\Omega(n \log n)$ for determining squarefreeness of strings over ordered alphabets? Note that the $\Omega(n \log n)$ bound in [17] assumes a *general* alphabet; for ordered alphabets, no lower bound (except for the trivial $\Omega(n)$ bound) has been proved for the offline case.
- Can the online squarefree recognition problem for *constant* alphabets be solved in $O(n)$ time?
- Can the running time of the LPT algorithm [15] be reduced to $O(n \log n)$ for general alphabets?
- How efficiently can the online and dynamic versions of the *cube* (and higher orders of repetitions) detection problem be solved?

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. R. Bean, A. Ehrenfeucht, and G. F. McNulty. Avoidable patterns in strings of symbols. *Pacific Journal of Mathematics*, 85(2):261–294, 1979.
3. G. Benson. Tandem repeats finder: A program to analyze DNA sequences. *Nucleic Acids Research*, 27(2):573–580, 1999.
4. D. Breslauer. *Efficient String Algorithmics.* PhD thesis, Columbia University, 1992.
5. A. T. Castelo, W. Martins, and G. R. Gao. TROLL – tandem repeat occurrence locator. *Bioinformatics*, 18(4):634–636, 2002.
6. C. Choffrut and J. Karhumäki. Combinatorics of words. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 1, pages 329–438. Springer-Verlag, 1997.
7. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
8. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
9. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific Publishing, 2002.
10. 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.
11. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.
12. J. Karhumäki. Automata on words. In *Proceedings of the 8th International Conference on Implementation and Application of Automata* (CIAA 2003), volume 2759 of *LNCS*, pages 3–10. Springer, 2003.
13. V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 12(1):32–44, 1992.

14. 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.
15. 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 *LNCS*, pages 432–439. Springer, 2004.
16. M. G. Main, W. Bucher, and D. Haussler. Applications of an infinite square-free co-CFL. *Theoretical Computer Science*, 49(2–3):113–119, 1987.
17. 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.
18. 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.
19. 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.