An Optimal Algorithm for Online Square Detection

Gen-Huey Chen, Jin-Ju Hong, and Hsueh-I Lu[⋆]

Department of Computer Science and Information Engineering
National Taiwan University

Abstract. A *square* is the concatenation of two identical non-empty strings. Let S be the input string which is given character by character. Let m be the (unknown) smallest integer such that the m-th prefix of S contains a square. The *online square detection* problem is to determine m as soon as the m-th character of S is read. The best previously known algorithm of the online square detection problem, due to Leung, Peng, and Ting, runs in $O(m \log^2 m)$ time. We improve the time complexity to $O(m \log \beta)$, where β is the number of distinct characters in the m-th prefix of the input string. It is not difficult to implement our algorithm to run in expected $O(m)$ time.

1 Introduction

Let $X \circ Y$ denote the concatenation of strings X and Y. A *square* is a non-empty string of the form $X \circ X$. A string does not contain any square is *square free*. Let $S[i, j]$ denote the substring of string S starting from position i and ending at position j. If string X equals $S[i, j]$, we say that X *starts* (or *occurs*) at position i and *ends* at position j in S.

Let S be a length-n string. Observe that there could be $\Omega(n^2)$ squares in S, e.g., when S is an all-one string. There are several $O(n \log n)$-time algorithms for finding compact representations of all squares in S [1, 2, 8]. In particular, each of these algorithms outputs $O(n \log n)$ periodic substrings of S such that any square of S occurs in one or more of those $O(n \log n)$ periodic substrings of S. These algorithms are optimal with respect to the worst-case output size [2].

Whether S is square free or not can be determined in $O(n \log \alpha)$ time [3, 4, 9], where α is the number of distinct characters in S. For example, Crochemore's approach [3, 4] is based upon the following f-factorization (also known as s-factorization [3], which is a variant of LZSS factorization [10]) of S. Let $|S|$ denote the length of string S. The *f-factorization* of S, obtainable in $O(n \log \alpha)$ time, is a partition of S into disjoint segments $B_1, B_2, \ldots, B_p$ for some $p \geq 1$ such that the following conditions hold for each $i = 2, 3, \ldots, p$, where $j = |B_1| + |B_2| + \cdots + |B_{i-1}|$:

[⋆] Corresponding author. Address: 1 Roosevelt Road, Section 4, Taipei 106, Taiwan, R.O.C. `http://www.csie.ntu.edu.tw/~hil/`, `hil@csie.ntu.edu.tw`

Condition 1 If $S[j+1]$ does not occur in $S[1,j]$ (i.e., $B_1 \circ B_2 \circ \cdots \circ B_{i-1}$), then $B_i = S[j+1]$;

Condition 2 Otherwise, B_i is the longest prefix of $S[j+1,n]$ that occurs in S before position $j+1$.

Suppose that $S[1,i]$ is known to the algorithm in $\Theta(i)$ time for each $i = 1,2,\ldots,n$ and the problem is to determine the smallest m such that $S[1,m]$ contains a square. (Once m is known, one can easily identify all squares of $S[1,m]$ in $O(m)$ time using longest common extensions.) Although there does not seem to be any previous work on this problem, it can be solved in $O(m \log \beta)$ time, where β is the number of distinct characters in $S[1,m]$. For example, we can resort to the *f'-factorization* of S, whose definition is the same as that of f-factorization except replacing its Condition 2 with the following.

Condition 2' Otherwise, B_i is the longest prefix of $S[j+1,n]$ that occurs in S and ends before position $j+1$.

Let $B'_1, B'_2, \ldots, B'_p$ be the f'-factorization of S. Suppose that i is the smallest index such that $|B'_1| + |B'_2| + \cdots + |B'_i| \geq m$. It is not difficult to see that the first i blocks of the f'-factorization of S as well as a square in $S[1,m]$ can be obtained in $O(m \log \beta)$ time.

Leung, Peng, and Ting [7] studied the square detection problem in a more restricted setting. Suppose that S is given to the algorithm character by character and the algorithm has to recognize m as soon as it reads $S[m]$. Leung, Peng, and Ting [7] gave an $O(m \log^2 m)$-time algorithm for the online square detection problem. Our contribution, summarized in Theorem 1, is to improve the running time to $O(m \log \beta)$. The $O(\log \beta)$ factor comes from the binary search required by the traversal of a suffix tree. Therefore, the expected running time of our algorithm can easily be reduced to $O(m)$ using hash tables. Our approach is inspired by Crochemore's algorithm [4] using the f-factorization of S.

Theorem 1. *The online detection problem for a string S can be solved in deterministic $O(m \log \beta)$ time, where $S[1,m]$ is the shortest prefix of S that contains a square and β is the number of distinct characters in $S[1,m]$.*

The rest of the paper is organized as follows. Section 2 describes our algorithm. Section 3 gives the implementation of our algorithm, whose time complexity is analyzed in Section 3.2. We conclude the paper with some open questions in Section 4.

2 Our Algorithm

A square $X \circ X$ is *centered* at position i in S if

$$S[i - |X| + 1, i + |X|] = X \circ X.$$

Let i_1, i_2, and i be positions in S with $i_1 < i_2 \leq i$. The following concept is crucial to our algorithm.

- An $L(i_1, i_2, i)$*-square* of S is a square of $S[i_1, i]$ that ends at position i and is centered at a position between i_1 and $i_2 - 1$ in S.
- An $R(i_1, i_2, i)$*-square* of S is a square of $S[i_1, i]$ that ends at position i and is centered at a position between i_2 and i in S.

See Figure 1 for an illustration.

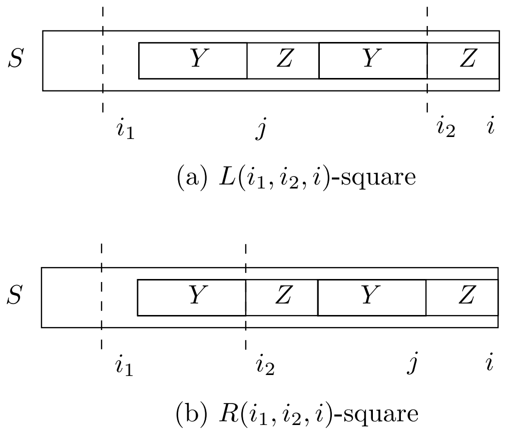

Fig. 1. $L(i_1, i_2, i)$-square and $R(i_1, i_2, i)$-square.

Our algorithm runs iteratively, where the i-th iteration receives $S[i]$ and detects whether there are squares in $S[1, i]$. More specifically, the i-th iteration obtains the f-factorization of $S[1, i]$ from that of $S[1, i - 1]$. Suppose that $S[i]$ belongs to the k-th block of the f-factorization of $S[1, i]$. The algorithm then detects whether there are

- $L(b_{k-1}, b_k, i)$-squares,
- $R(b_{k-1}, b_k, i)$-squares, or
- $R(1, b_{k-1}, i)$-squares,

where for each $j = 1, 2, \ldots, k$, let b_j denote the starting position of the j-th block of the f-factorization of $S[1, i]$. If no square is detected, then the algorithm proceeds to the next iteration. Otherwise, the algorithm outputs i and halts.

Lemma 1. *If the input string S is not square free, then our algorithm correctly outputs the smallest index m such that $S[1, m]$ is not square free.*

Proof. Since $S[1, m-1]$ is square free, the algorithm does not halt before the m-th iteration. It suffices to show that a square of $S[1, m]$ has to be an $L(b_{k-1}, b_k, i)$-square, an $R(b_{k-1}, b_k, i)$-square, or an $R(1, b_{k-1}, i)$-square, where b_j is the starting position of the j-th block B_j in the f-factorization of $S[1, m]$ for each $j = 1, 2, \ldots, k$.

Since $S[b_k, m]$ is in B_k, by definition of f-factorization, $S[b_k, m]$ is a substring of $S[1, m-1]$. Since $S[1, m-1]$ is square free, so is $S[b_k, m]$. If a square of $S[1, m]$ occurs before b_{k-1}, then the square has to be an $R(1, b_{k-1}, m)$-square. If a square of $S[1, m]$ occurs at a position between b_{k-1} and $b_k - 1$, then the square has to be an $L(b_{k-1}, b_k, m)$-square or an $R(b_{k-1}, b_k, m)$-square. □

Comment: Our proof of Lemma 1 is modified from that of Theorem 8.2 in [4].

3 Implementation

Longest common extensions [8] are crucial to the implementation of our algorithm. For positions $i \le j \le k$ in S,

- Let $X_R(i, j, k)$ denote the *longest common right extension* of positions i and j with boundaries k, i.e., the length of the longest common prefix of $S[i, k]$ and $S[j, k]$.
- Let $X_L(j, k, i)$ denote the *longest common left extension* of positions j and k with boundaries i, i.e., the length of the longest common suffix of $S[i, j]$ and $S[i, k]$.

It is not difficult to see the following lemma from Figure 1.

Lemma 2 (Main and Lorentz [8]).

1. *S has an $L(i_1, i_2, i)$-square if and only if there is an index j with $i_1 \le j < i_2$ such that $X_R(j, i_2, i) = |S[i_2, i]|$ and $X_L(j-1, i_2-1, i_1) + X_R(j, i_2, i) \ge |S[j, i_2-1]|$.*
2. *S has an $R(i_1, i_2, i)$-square if and only if there is an index j with $i_2 < j < i$ such that $X_R(i_2, j+1, i) = |S[j+1, i]|$ and $X_L(i_2-1, j, i_1) + X_R(i_2, j+1, i) \ge |S[i_2, j]|$.*

Our implementation uses several suffix trees [6, 11]. Let T' be the suffix tree of a string S'. If W' is a substring of S', then there is a unique path, denoted $P(T', W')$, in T' whose label spells out W'. Moreover, if S' contains β' distinct characters, then, given the ending position of $P(T', W')$ in T', it takes $O(|W''| \log \beta')$ time to determine whether $W' \circ W''$ is also a substring of S'.

3.1 Detecting $L(i_1, i_2, i)$-Squares

Let $i_1 < i_2 \le i_3$ be three given indices. Suppose that the number of distinct characters in $S[i_1, i_3]$ is $O(\beta)$. The following lemma is a key ingredient in the implementation of our algorithm.

Lemma 3. *Let i be the smallest index with $i_2 \le i \le i_3$ such that S has an $L(i_1, i_2, i)$-square. There is an algorithm $A_L(i_1, i_2, i_3)$ that*

- *either determines in $O((i_3 - i_1) \log \beta)$ time that i is undefined without reading any characters in $S[i_3 + 1, n]$*
- *or identifies i in $O((i - i_1) \log \beta)$ time without reading any character of $S[i+1, n]$.*

The rest of the subsection proves Lemma 3.

The preprocessing. The first step is the $O(|S[i_1, i_2]|)$-time preprocessing with which the value of $X_L(j, i_2 - 1, i_1)$ for any index j with $i_1 \leq j < i_2$ can be determined in $O(1)$ time [5, 8]. We then build the suffix tree T_1 of $S[i_1, i_2 - 1] \circ \$$, where \$ is a character not in S [11]. One can easily verify that the preprocessing takes $O((i_2 - i_1) \log \beta)$ time.

Define $z(j) = |S[j, i_2 - 1]| - X_L(j - 1, i_2 - 1, i_1)$ for each index j with $i_1 \leq j < i_2$. Finally, for each node v of T_1, we store an index, denoted $j(v)$, at node v that minimizes $z(j)$ overall all indices j such that $P(T_1, S[j, i_2 - 1])$ passes v. The indices $j(v)$ for all nodes v of T_1 can be computed in $O(i_2 - i_1)$ time in a bottom-up manner.

The iterative procedure. For $i = i_2, i_2 + 1, \ldots, i_3$, the i-th iteration does the following. If $S[i_2, i]$ occurs in $S[i_1, i_2 - 1]$ and $|S[i_2, i]| \geq z(j(v))$, where v is the highest node in T_1 such that the path of T_1 between the root of T_1 and v contains $P(T_1, S[i_2, i])$, then the procedure reports i and halts. Otherwise, the procedure proceeds to the next iteration.

Correctness. The condition $X_R(j, i_2, i) = |S[i_2, i]|$ in Lemma 2(1) is equivalent to the condition that $S[i_2, i]$ is a prefix of $S[j, i_2 - 1]$, which is also equivalent to the condition that $P(T_1, S[j, i_2 - 1])$ contains $P(T_1, S[i_2, i])$. With $X_R(j, i_2, i) = |S[i_2, i]|$, the condition $X_L(j - 1, i_2 - 1, i_1) + X_R(j, i_2, i) \geq |S[j, i_2 - 1]|$ in Lemma 2(1) is equivalent to the condition

$$|S[i_2, i]| \geq |S[j, i_2 - 1]| - X_L(j - 1, i_2 - 1, i_1).$$

Let v be the highest node in T_1 such that the path of T_1 between v and the root of T_1 contains $P(T_1, S[i_2, i])$. By definition of $z(j(v))$, the above condition is equivalent to the condition

$$|S[i_2, i]| \geq z(j(v)).$$

Therefore, by Lemma 2(1), the above iterative procedure does report the smallest index i, if any, with $i_2 \leq i \leq i_3$ such that S has an $L(i_1, i_2, i)$-square.

Time complexity. Suppose that in the previous iteration we already have the ending position of $P(T_1, S[i_2, i - 1])$ in T_1. It takes $O(\log \beta)$ time to determine whether $S[i_2, i]$ is a substring of $S[i_1, i_2 - 1]$. If $S[i_2, i]$ does occur in $S[i_1, i_2 - 1]$, we also keep the ending position of $P(T_1, S[i_2, i])$ to be used in the next iteration. As a result, it is not difficult to verify that the time complexity described in Lemma 3 holds.

3.2 Detecting $R(i_1, i_2, i)$-Squares

Let $i_1 < i_2 < i_3$ be three given indices. Suppose that the number of distinct characters in $S[i_1, i_3]$ is $O(\beta)$. The following lemma is also a key ingredient in the implementation of our algorithm.

Lemma 4. *Let i be the smallest index with $i_2 < i \leq i_3$ such that S has an $R(i_1, i_2, i)$-square. There is an algorithm $A_R(i_1, i_2, i_3)$ such that*

- *if i is undefined, then $A_R(i_1, i_2, i_3)$ reports "i is undefined" in $O((i_3 - i_2)\log\beta)$ time without reading any character of $S[i_3 + 1, n]$;*
- *otherwise, if S has no $L(i_1, i_2, j)$-squares for any $j \in \{i_2, i_2 + 1, \ldots, i\}$, then algorithm $A_R(i_1, i_2, i_3)$ reports i in $O((i - i_2)\log\beta)$ time without reading any character of $S[i + 1, n]$.*

The rest of the subsection proves Lemma 4.

The iterative procedure. For each $i = i_2 + 1, i_2 + 2, \ldots, i_3$, the i-th iteration does the following.

- We first compute the index j_i in $O(1)$ time such that $|S[j_i, i_2 - 1]| = \min(|S[i_1, i_2 - 1]|, |S[i_2, i]|)$.
- We then compute the suffix tree T_2 of $S[j_i, i_2 - 1] \circ \$$ from that of $S[j_{i-1}, i_2 - 1] \circ \$$ in amortized $O(\log\beta)$ time using, e.g., Inenaga's algorithm [6].
- We maintain a data structure for $S[j_i, i_2 - 1]$ from which the value of $X_L(j, i_2 - 1, j_i)$ for any j with $j_i \leq j < i_2$ can be computed in $O(1)$ time. According to [5, 8], such a data structure for $S[j_i, i_2 - 1]$ can be obtained from that of $S[j_{i-1}, i_2 - 1]$ in amortized $O(1)$ time.
- We maintain a data structure for $S[i_2, i]$ from which the value of $X_R(i_2, j, i)$ for any j with $i_2 \leq j \leq i$ can be computed in $O(1)$ time. Similarly, according to [5, 8], such a data structure for $S[i_2, i]$ can be obtained from that of $S[i_2, i - 1]$ in amortized $O(1)$ time.
- Let $F(i)$ denote the longest suffix of $S[i_2, i]$ that is a substring of $S[j_i, i_2 - 1]$. We obtain the ending position of $P(T_2, F(i))$ in T_2 from the ending position of $P(T_2, F(i - 1))$ in amortized $O(\log\beta)$ time. It then takes $O(1)$ time to compute an index $y(i) \leq i_2 - 1$ such that an occurrence of $F(i)$ in $S[i_1, i_2 - 1]$ ends at position $y(i)$. We determine in $O(1)$ time

$$X_L(i_2 - 1, i, i_1) = \begin{cases} |F(i)| & \text{if } y(i) = i_2 - 1; \\ \min(|F(i)|, X_L(y(i), i_2 - 1, j_i)) & \text{otherwise.} \end{cases} \tag{1}$$

- Now we insert in $O(1)$ time the index i to the set $K(e(i))$, where

$$e(i) = i + |S[i_2, i]| - X_L(i_2 - 1, i, i_1).$$

- If there is an index j in $K(i)$, if

$$X_R(i_2, j + 1, i) = |S[j + 1, i]|,$$

  then our procedure reports i; otherwise, the iterative procedure proceeds to the next iteration.

Correctness. First of all, one can see the correctness of Equation (1) by verifying that both sides of the equality are equal to $X_L(i_2-1,i,j_i)$. Observe that in the i-th iteration $K(i)$ has collected all the indices $j<i$ with $e(j)=i$. If $X_R(i_2,j+1,i)=|S[j+1,i]|$, then the condition

$$X_L(i_2-1,j,i_1)+X_R(i_2,j+1,i)\geq|S[i_2,j]|$$

in Lemma 2(2) is equivalent to the condition $e(j)\leq i$. Moreover, the condition

$$X_L(i_2-1,j,i_1)+X_R(i_2,j+1,i)>|S[i_2,j]|,$$

which is equivalent to the condition $e(j)<i$, implies that S has a square ending at position $i-X_L(i_2-1,j,i_1)+X_R(i_2,j+1,i)+|S[i_2,j]|$. Therefore, by Lemma 2(2), our iterative procedure outputs i if and only if S has an $L(i_1,i_2,i)$-square.

Time complexity. According to the above explanation, it is not difficult to see that the time complexity of Lemma 4 holds.

3.3 The Implementation

With subroutines $A_L(i_1,i_2,i_3)$ and $A_R(i_1,i_2,i_3)$, we prove the following lemma.

Lemma 5. *Our algorithm described in Section 2 can be implemented to run in $O(m\log\beta)$ time.*

Proof. The implementation proceeds iteratively for $i=1,2,\ldots,n$, where the i-th iteration reads $S[i]$ and performs the following steps.

- We obtain the suffix tree T of $S[1,i]$ from the suffix tree of $S[1,i-1]$ in amortized $O(\log\beta)$ time. We then determine the index k_i such that $S[i]$ is in the k_i-th block of the f-factorization of S. Observe that with the help of T, one can compute k_i from k_{i-1} in $O(\log\beta)$ time. If $k_i=1$, we proceed to the next iteration.
- Knowing $k_i\geq 2$, we perform
  - the i-th iteration of $A_L(k_{i-1},k_i,k_{i+1})$,
  - the i-th iteration of $A_R(k_{i-1},k_i,k_{i+1})$, and
  - the i-th iteration of $A_R(1,k_{i-1},k_{i+1})$

  in the above order. If any of these three i-th iterations reports i and halts, then our implementation also reports i and halts. Otherwise, our implementation proceeds to the next iteration.

The description of our implementation ignores on purpose the fact that we do not know the value of k_{i+1} in the i-iteration. However, one can verify that this abuse to the interface of subroutines $A_L()$ and $A_R()$ is all right, since each iteration of our implementation calls only the i-th iterations of subroutines $A_L()$ and $A_R()$. It follows from Lemmas 3 and 4 that our implementation correctly outputs m in $O(m\log\beta)$ time. □

Now one can easily see that Theorem 1 is immediate from Lemmas 1 and 5.

4 Concluding Remarks

As we mentioned in the introduction, each of those $O(\log \beta)$ terms comes from the binary search required for choosing the right branch to go while traversing a suffix tree of a string with $O(\beta)$ distinct characters. Using hash tables, one can implement our algorithm to run in expected $O(m)$ time. An immediate open question is to see if it is possible to further reduce the required running time to worst-case $O(m)$ time. It would also be of interest to see if the technique of f-factorization can be extended to detect repeats of the form X^k with $k > 2$ in an online manner.

References

1. A. Apostolico and F. P. Preparata. Optimal off-line dection of repetitions in a string. *Theoretical Computer Science*, 22:294–315, 1983.
2. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
3. M. Crochemore. Recherche linéaire d'un carré dans un mot. *Comptes Rendus des Séances de l'Académie des Sciences. Série I. Mathématique*, 296(18):781–784, 1983.
4. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.
5. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.
6. S. Inenaga. Bidirectional construction of suffix trees. *Nordic Journal of Computing*, 10(1):52–67, 2003.
7. H.-F. Leung, Z. Peng, and H.-F. Ting. An efficient online algorithm for square detection. In K.-Y. Chwa and J. I. Munro, editors, *Proceeings of the 10th Annual International Conference*, Lecture Notes in Computer Science 3106, pages 432–439, Jeju Island, Korea, August 17-20 2004. Springer-Verlag.
8. 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.
9. 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 F12 of *NATO ASI Series*, pages 271–278. Springer-Verlag, 1985.
10. J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.
11. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.