

A sublinear time string prefix square detection algorithm*

Yung-Tsang Chang^{1,2,†}

Jung-Hua Hsu^{2,‡}

S. S. Yu^{2,§}

¹ *Department of Information Management
Hsiuping Institute of Technology
No. 11, Gungye Rd., Da-Li City
Taichung, Taiwan
R.O.C.*

² *Department of Computer Science
National Chung-Hsing University
Taichung, Taiwan 402
R.O.C.*

Abstract

In this paper, an algorithm is proposed to detect the occurrence of a prefix square concurrently when a string is inputted. Some properties of certain variables are studied in order to show the correctness and the time complexity of this algorithm. The time complexity of this algorithm is proved to be not only linear but also sublinear.

Keywords : Prefix square, matching algorithm, string, complexity, sublinear.

1. Introduction

This paper presents a sublinear-time on-line algorithm which concerns the exact string matching problem: checking whether an input string has a prefix repetition. One is referred to [6] for a linear-time on-

*This paper was supported by the National Science Council R.O.C. under Grant NSC 93-2115-M-005-006.

[†]*E-mails*: kenchang@mail.hit.edu.tw, phd9115@cs.nchu.edu.tw

[‡]*E-mail*: s9156016@cs.nchu.edu.tw

[§]*E-mail*: pyu@cs.nchu.edu.tw

line algorithm for finding the smallest prefix palindrome. Exact string matching is one of the most important topics in discrete mathematics and in many other fields. It has extensive applications in data processing, lexical analysis, compiler design, image processing, DNA decoding, etc. In this paper, an exact string matching problem is described as: for a given set of words, called keywords or patterns, and for any other given word, called the subject word or the text, one is going to check whether there is a keyword being a subword of the subject word. Usually, keywords or the subject word are preprocessed in order to speed up the matching process. For example, algorithms proposed in [1] and [4] consist of constructing related matching tries and functions according to the given keywords and then using these tries and functions to accomplish the matching works. Also in [3], an algorithm is applied to preprocess keywords and produce a related matching finite state automaton which is then applied to accomplish the matching works. On the other hand, algorithms introduced in [9] and [10] are used to construct a suffix tree for a given subject word and then apply this suffix tree to detect the occurrences of keywords. The algorithm presented in [9] can be applied to do the longest common substrings matching of two strings in $O(m + n)$ units of time, where m and n are lengths of these two strings, respectively.

Suffix tree structures can also be applied to find repeated subwords from a given word (see [8]). The repeated subword problem is one of the most important problems of string comparing in molecular biology since high biomolecular string similarity often indicates significant functional or structural similarity ([2]). There are many other algorithms concerning finding repetitions in a given word w . For instance, an algorithm which is applied to compute the length of a repeated suffix for each prefix of a given word was proposed in [5].

Let A be a finite alphabet and A^* the free monoid generated by A . Any element $w \in A^*$ is a word, i.e., a finite sequence of symbols taken from A . The length of a word w is denoted as $\lg(w)$. If $u \in A^*wA^*$ then w is called an *infix* or a *subword* of u . If $u = vw$ then v (resp., w) is a *prefix* (resp., a *suffix*) of u . A word v is a *bifix* of another word u if v is a prefix and a suffix of u . If v is a prefix (resp., a suffix, a bifix) of u and $v \neq u$ then v is a *proper prefix* (resp., a *proper suffix*, a *proper bifix*) of u . When designing a neural network with n nodes for a set of data (i.e., input-output pairs), one must check whether there is a solution, i.e.,

the (input, output) results of the n nodes neural network is consistent with the (input, output) pairs of inputted data. If a dependent set of weights of the neural network occurs when an independent set of (input, output) pairs is inputted, called a collision, then some extra nodes must be added into the neural network; otherwise, the neural network can not satisfy the data. One way to check whether there are collisions is by transferring the weights of nodes for each input datum-pair into a vector of letters and checking whether there exists a prefix square of this sequence of vectors with respect to the corresponding input datum-pairs. Once a prefix square is found during the training of a neural network, it means that a collision occurs.

In this paper, we modify the 'next function algorithm' introduced in [4] and present an algorithm to detect whether there exists a prefix square when a sequence is inputted. In fact, the prefix square problem is much easier than other string matching problems. Thus, one does not need to apply most more complex known algorithms, such as algorithms mentioned before, to solve the prefix square matching problem. Moreover, the prefix square matching process can be accomplished concurrently during inputting a data sequence. That is, this work can be done on-line. In the meanwhile, one is referred to [6] and [7] for on-line string matching algorithms.

Through the investigation of properties concerning some variables related to or used by the algorithm given in Section 2, the time complexity of this algorithm is studied. Even though we do not give the least upper bound of the time complexity,

we show that the time complexity of this algorithm is sublinear.

2. Prefix square detection algorithm

This section presents an algorithm to detect whether there is a prefix square when a sequence is inputted. The 'next function algorithm' proposed in [4] was used to find the next state when a matching trie was used to do the string matching. The essential rule of this algorithm is that there is only one initial state and the next state is obtained by comparing the suffixes with the prefixes of a string. This is quite suitable for the prefix square detection problem. Suppose a sequence of letters $a_1, a_2, \dots, a_n, \dots$ is inputted. We now modify the 'next function algorithm' in [4] and present an algorithm to detect whether there is a prefix square as follows.

Algorithm 2.1**Input:** $a_j, j = 1, 2, \dots$ **Variables:** n_k : the length of the longest proper bifix of $a_1 a_2 \dots a_k$; j : an index denotes the letter a_j inputted; $i + 1$: an index denotes the letter a_{i+1} which is compared with the inputted letter a_j .

1. $n_1 := 0, j := 2, i := n_1$.
2. While $a_j \neq \text{Nil}$ Do
3. While $a_j \neq a_{i+1}$ and $i \neq 0$ Do
4. $i := n_i$.
5. End While
6. If $i = 0$
7. Then $n_j := 0$.
8. End If
9. If $a_j = a_{i+1}$
10. Then $n_j := i + 1, i := n_j$.
11. End If
12. If $n_j = \frac{j}{2}$
13. Then Output: "A prefix $a_1 \dots a_{n_j}$ is repeated"; Exit.
14. End If
15. $j := j + 1$
16. End While
17. Output "There is no prefix square".

Output: The repeated prefix $a_1 \dots a_{n_j}$ once it is detected.

Example 2.2. Consider the sequence $w = \text{ATCATGAATCATAATCATGAATCATA}$ over the alphabet $\{A, T, C, G\}$. Let 4_j denote the times of Line 4 executed when a_j is inputted. The number 4_j will be studied in Section 4. Then when the sequence w is inputted, the related variables j, a_j, n_j of Algorithm 2.1 and the number 4_j are listed as follows.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
a_j	A	T	C	A	T	G	A	A	T	C	A	T	A	A	T	C	A	T	G	A	A	T	C	A	T	A
n_j	0	0	0	1	2	0	1	1	2	3	4	5	1	1	2	3	4	5	6	7	8	9	10	11	12	13
4_j	0	0	0	0	0	1	0	1	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0

As $n_{26} = 13$, i.e., $n_j = 13 = \frac{j}{2}$, the output of this example is the repeated prefix $ATCATGAATCAT\bar{A}$. From this example, it could be found that the time complexity of Algorithm 2.1 should be sublinear. It is shown in Section 4 by comparing the number 4_j with the number n_j .

3. Properties of Algorithm 2.1

In this section, some properties of the variable n_j will be derived and the correctness of Algorithm 2.1 will be shown. By virtue of Lines 12-14 of Algorithm 2.1, $n_j = \frac{j}{2}$ means $a_1a_2 \cdots a_{n_j} = a_{n_j+1}a_{n_j+2} \cdots a_j$. The meaning of n_j concerns the correctness of Algorithm 2.1. Thus the variable n_j plays an essential role in this algorithm. Now, we show a property of the number n_j as follows.

Proposition 3.1. *Let $j \geq 1$. The number n_j of Algorithm 2.1 denotes the length of the longest proper prefix v of $a_1a_2 \cdots a_j$ such that v is also a suffix of $a_1a_2 \cdots a_j$.*

Proof. Line 1 sets n_1 to 0 and i to n_1 . This means the only proper such prefix is the empty word λ . Suppose that there is $k \geq 1$ such that for any $1 \leq j \leq k$, n_j denotes the length of the longest proper prefix v of $a_1a_2 \cdots a_j$ such that v is also a suffix of $a_1a_2 \cdots a_j$. Consider n_{k+1} . From Line 7 and Line 10, it is clear that $i = n_k$ when $j = k + 1$ and Line 3 is executed. If $a_{k+1} \neq a_{i+1}$ then $a_1a_2 \cdots a_{i+1} \neq a_{k-i+1} + a_{k-i+2} \cdots a_{k+1}$. As $a_1a_2 \cdots a_i = a_{k-i+1}a_{k-i+2} \cdots a_k$ whenever $i \neq 0$, the longest proper prefix v of $a_1a_2 \cdots a_i$ such that v is a suffix of $a_{k-i+1}a_{k-i+2} \cdots a_k$ is the next proper prefix and suffix of $a_1a_2 \cdots a_k$, which must be considered. Moreover, $a_1a_2 \cdots a_i = a_{k-i+1}a_{k-i+2} \cdots a_k$ yields that v is a suffix of $a_1a_2 \cdots a_i$ and $\lg(v) = n_i$. Thus Line 4 sets i to n_i . If $i = 0$, then we need only check whether $a_1 = a_{k+1}$. Therefore, one of Line 7 and Line 10 will set n_{k+1} to the length of the longest proper prefix v of $a_1a_2 \cdots a_{k+1}$ such that v is also a suffix of $a_1a_2 \cdots a_{k+1}$. By mathematical induction on the index j , the assertion holds. \square

In view of Proposition 3.1 and the line 12 of Algorithm 2.1, one can derive the following result immediately.

Proposition 3.2. *Algorithm 2.1 can find the first prefix repetition of a word.*

Proposition 3.3. *$n_j \leq \frac{j}{2}$ and $n_j < \frac{j}{2}$ except the last n_j . Hence, if $n_j \neq 0$ then $n_{n_j} < \frac{n_j}{2}$.*

Proof. From Proposition 3.1 and the line 12 of Algorithm 2.1, it follows that $n_j \leq \frac{j}{2}$. Since the algorithm ends when $n_j = \frac{j}{2}$ or $a_j = \text{Nil}$, the case $n_j = \frac{j}{2}$ can only hold for the last n_j . It is then clear that if $n_j \neq 0$ then $n_{n_j} < \frac{n_j}{2}$. \square

4. The time complexity of Algorithm 2.1

When we survey Algorithm 2.1, it is not difficult to find that each line will be executed at most once for each input a_j except Lines 3, 4 and 5. Thus, from the time complexity of Line 4, one can derive the time complexity of the algorithm. Let 4_j denote the times of Line 4 of Algorithm 2.1 to be executed when a_j is inputted. The following lemma provides properties concerning variables n_j and 4_j :

Lemma 4.1.

- (1) $4_1 = 0$ and $4_2 = 0$.
- (2) If a_j is processed for some $j \geq 3$, then $n_2 = 0$ and $4_3 = 0$.
- (3) If $n_j = 0$, then $4_{j+1} = 0$ and $n_{j+1} = 0$ or 1.
- (4) If $n_j = 1$, then $4_{j+1} = 0$ or 1.
- (5) If $n_j = n_{j-1} + 1$, then $4_j = 0$.
- (6) If $n_j \neq 0$ and $4_{j+1} = 0$, then $n_{j+1} = n_j + 1$.

Proof. By virtue of Line 1 of Algorithm 2.1, it is clear that $4_1 = 0$ and $4_2 = 0$. If $a_2 = a_1$, then Algorithm 2.1 ends. Thus a_j being processed for an integer $j \geq 3$ means that $a_2 \neq a_1$ and $n_2 = 0$. This yields $4_3 = 0$. From Line 3 of Algorithm 2.1, it is clear that if $n_j = 0$ then $4_{j+1} = 0$; by Lines 7 and 10, $n_{j+1} = 0$ or 1. If $n_j = 1$ and $a_{j+1} = a_{n_j+1}$, then $4_{j+1} = 0$. Let $i = n_j$. As $n_1 = 0$, if $n_j = 1$ and $a_{j+1} \neq a_{n_j+1} = a_2$, then by Line 4 of Algorithm 2.1, $i = n_{n_j} = n_1 = 0$. By Lines 7 and 10 of Algorithm 2.1, $n_{j+1} = 0$ or 1. Lines 9 and 10 of Algorithm 2.1 yield that $n_j = n_{j-1} + 1$ holds only if $a_j = a_{n_{j-1} + 1}$. Thus $n_j = n_{j-1} + 1$ implies $4_j = 0$. If $n_j \neq 0$

and $4_{j+1} = 0$, then by Line 3 of Algorithm 2.1, $a_{j+1} = a_{n_{j+1}}$. From Line 10 of Algorithm 2.1, it follows $n_{j+1} = n_j + 1$. \square

Proposition 4.2. *Let $4_j = r \geq 1$, $i_1 = n_{j-1}$ and $i_{s+1} = n_{i_s}$ for $1 < s < r$. Then $n_j = n_{i_{r+1}} + 1$ or $n_j = 0$.*

Proof. In view of the while loop: Lines 3-5 of Algorithm 2.1, it follows that $i_s \neq 0$ and $a_{n_j} \neq a_{i_{s+1}}$ for $1 \leq s \leq r$. And each time the loop resets i from its to i_{s+1} , where $1 \leq s \leq r$. If $a_j = a_{i_{r+1}+1}$, then by Line 10 of Algorithm 2.1, $n_j = n_{i_r} + 1$. The case $a_j \neq a_{i_{r+1}+1}$ can hold only when $n_{i_r} = 0$. In this case, by virtue of Line 7 of Algorithm 2.1, $n_j = 0$. \square

Furthermore, some more properties concerning the number 4_j are shown in the sequel.

Proposition 4.3. *$4_j < \log_2 j$ for any $j \geq 1$.*

Proof. By Lemma 4.1, $4_1 = 0$, $4_2 = 0$, and $4_3 = 0$ if they are counted. Thus $4_j < \log_2 j$ for $1 \leq j \leq 3$. For $j \geq 4$, clearly if $4_j = 0$ or $4_j = 1$, then $4_j < \log_2 j$. Now let $j \geq 4$ and $4_j = r \geq 2$. Let $i_1 = n_{j-1}$ and $i_{s+1} = n_{i_s}$ for $1 \leq s \leq r$. In view of the while loop: Lines 3-5 of Algorithm 2.1, it follows that $i_s \neq 0$ for $1 \leq s \leq r$. By virtue of Proposition 3.3, $i_{s+1} < i_s/2$ for $1 \leq s \leq r$. Since $i_1 = n_{j-1}$ is not the last one, by Proposition 3.3, $i_1 < \frac{j-1}{2}$. Thus $i_r < \frac{i_1}{2^{r-1}} < \frac{j}{2^r}$. As $i_r \neq 0$, $i_r \geq 1$. This yields that $1 < \frac{j}{2^r}$. Hence $0 < (\log_2 j) - r$, i.e., $r < \log_2 j$. \square

For the given string $w = a_1 a_2 \cdots a_n$ processed in Algorithm 2.1, let

$$m_j = \begin{cases} 4_j - 1 & \text{if } 4_j \neq 0 \\ 0 & \text{if } 4_j = 0, \end{cases}$$

$$0_j = \begin{cases} 0 & \text{if } 4_j \neq 0 \\ 1 & \text{if } 4_j = 0 \end{cases} \quad \text{and} \quad \beta_j = \sum_{i=1}^j (0_i - m_i).$$

The following proposition concerns the execution time of Line 4 which is compared with the number n_j .

Proposition 4.4. *$\beta_j > n_j$ for each $j \geq 1$.*

Proof. By Lemma 4.1, $4_1 = 0$ and $4_2 = 0$, and $4_3 = 0$ if it is counted. Thus, $\beta_1 = 1$, $\beta_2 = 2$ and $\beta_3 = 3$. It is clear that $n_1 = 0$ and $n_2 \leq 1$. And, $n_2 = 0$ when $j \geq 3$ is considered. Now, if $n_3 = 0$, then $4_4 = 0$. Otherwise, $n_3 = 1$

and $a_3 = a_1$. In this case, if $a_4 = a_2$, then $4_4 = 0$, $n_4 = 2$ and Algorithm 2.1 ends. If $a_4 \neq a_2$, then $4_4 = 1$ and $n_4 = a$ or 1. These observations yield $\beta_j > n_j$ for each $1 \leq j \leq 4$. Assume that there is $r \geq 4$ such that for each $1 \leq j \leq r$, $\beta_j > n_j$. Now, consider β_{r+1} and n_{r+1} . We have the following two cases:

- (1) $n_{r+1} = n_r + 1$. Then by Lemma 4.1, $4_{r+1} = 0$. This implies that $\beta_{r+1} = \beta_r + 1$. By assumption, $\beta_{r+1} = \beta_r + 1 > n_r + 1 = n_{r+1}$.
- (2) $n_{r+1} \neq n_r + 1$. By virtue of Lines 9-11, $n_{r+1} \neq n_r + 1$ only if $a_{r+1} \neq a_{n_{r+1}}$. Consider the following two sub cases:
 - (2.1) $n_r = 0$. Then by Lines 6-8, $n_{r+1} = 0$. And, clearly, $4_{r+1} = 0$. This yields $\beta_{r+1} = \beta_r > n_r = n_{r+1}$.
 - (2.2) $n_r \neq 0$. In view of Line 3, it follows that $4_{r+1} \geq 1$. If $4_{r+1} = 1$, then $\beta_{r+1} = \beta_r$ and $n_{r+1} = n_r + 1$ or $n_{r+1} = 0$. By Proposition 3.3, $n_{n_r} < \frac{n_r}{2}$. Thus $n_{r+1} \leq n_r$. By assumption, $\beta_{r+1} = \beta_r > n_r \geq n_{r+1}$. Now, consider the case $4_{r+1} > 1$. From the definition of β , we have $\beta_{r+1} = \beta_r - 4_{r+1} + 1$. Let $k = 4_{r+1}$, $i_1 = n_r$ and $i_s = n_{i_{s-1}}$ for $2 \leq s \leq k$. By Line 3, $i_s \neq 0$ and $a_{r+1} \neq a_{i_s}$ for any $1 \leq s \leq k$. By virtue of Lines 7 and 10, $n_{r+1} = 0$ or $n_{r+1} = n_{i_k} + 1$. Proposition 3.3 implies that $n_{i_s} \leq i_s - 1$ for $1 \leq s \leq k$. By Algorithm 2.1, the condition $k = 4_{r+1} > 1$ derives that $n_{r+1} = 0$ or $n_{r+1} = n_{i_k} + 1$. Thus $n_{r+1} \leq n_r - k + 1$. By assumption, $\beta_{r+1} = \beta_r - 4_{r+1} + 1 > n_r - k + 1 \geq n_{r+1}$.

By mathematical induction on j , the assertion holds. \square

As $n_j \geq 0$ for any $j \geq 1$, Proposition 4.4 derives the following result immediately.

Proposition 4.5. $\sum_{i=1}^j m_i < \sum_{i=1}^j 0_i$.

Let $\gamma_j = \begin{cases} 1 & \text{if } 4_j \geq 1 \\ 0 & \text{if } 4_j = 0 \end{cases}$. Proposition 4.5 derives $\sum_{i=1}^j 4_i = \sum_{i=1}^j m_i +$

$\sum_{i=1}^j \gamma_i < \sum_{i=1}^j 0_i + \sum_{i=1}^j \gamma_i$. Since $\sum_{i=1}^j \gamma_i \sum_{i=1}^j 0_i = j$, we then have the following result concerning the time complexity of Algorithm 2.1.

Proposition 4.6. $\sum_{i=1}^j 4_i < j$ for each $j \geq 1$.

Note that each line of Algorithm 2.1 will be executed at most once for each input a_j except Lines 3-5. This together with Proposition 4.6 yields that the time complexity of Algorithm 2.1 is sublinear, i.e., less than or equal to $\lg(w)$. There is a word w such that $\sum_{j=1}^{\lg(w)} 4_j = \lg(w) - 3$ when w is processed in Algorithm 2.1. For example: Let $w = a_1 a_2 a_1 a_1 \dots a_1$ where $a_1 \neq a_2$ with $\lg(w) < 4$. It is clear that $4_j = 1$ for any $j > 3$. Hence, $\sum_{j=1}^{\lg(w)} 4_j = \lg(w) - 3$. In fact, the number $\lg(w) - 3$ should be the upper bound of the number $\sum_{j=1}^{\lg(w)} 4_j$ when a word w is processed in Algorithm 2.1.

References

- [1] A. V. Aho and M. J. Corasick, Efficient string matching: an aid to biologicraphic search, *Communications of the ACM*, Vol. 18 (6) (1975), pp. 333–340.
- [2] D. Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997.
- [3] J. H. Hsu, T. S. Tsui and S. S. Yu, A concurrent string matching for multiple keywords with infix checking, *submitted*.
- [4] D. E. Knuth, J. H. Morris, Jr. and V. R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.*, Vol. 6 (2) (1977), pp. 323–350.
- [5] A. Lefebvre and T. Lecroq, A heuristic for computing repeats with a factor oracle: application to biological sequences, *Intern. J. Computer Math.*, Vol. 79 (12) (2002), pp. 1303–1315.
- [6] G. Manacher, A new linear-time on-line algorithm for finding the smallest initial palindrome of a string, *J. Assoc. Comput. Math.*, Vol. 22 (1975), pp. 346–351.
- [7] P. D. Michailidis and K. G. Margaritis, On-line string matching algorithms: survey and experimental result, *OPA*, Vol. 76 (2001), pp. 411–434.
- [8] J. Stoye and D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, *Theoret. Computer Sci.*, Vol. 270 (2002), pp. 843–856.
- [9] P. Weiner, Linear pattern matching algorithm, in *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, (1973), pp. 1–11.
- [10] E. Ukkonen, On-line construction of suffix-trees, *Algorithmica*, Vol. 14 (1995), pp. 249–260.

Received January, 2005