Vestnik of Don State Technical University. 2019. Vol. 19, no. 3, pp. 290–300. ISSN 1992-5980 eISSN 1992-6006
Вестник Донского государственного технического университета. 2019. Т. 19, № 3. С. 290–300. ISSN 1992-5980 eISSN 1992-6006

INFORMATION TECHNOLOGY, COMPUTER SCIENCE, AND MANAGEMENT
ИНФОРМАТИКА, ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА И УПРАВЛЕНИЕ

UDC 519.689 https://doi.org/10.23947/1992-5980-2019-19-3-290-300

Accelerated preprocessing in task of searching substrings in a string *

A. V. Mazurenko[1], N. V. Boldyrikhin[2]**
[1] DDoS-GUARD LLC, Rostov-on-Don, Russian Federation
[2] Don State Technical University, Rostov-on-Don, Russian Federation

Ускоренный препроцессинг в задаче поиска подстрок в строке***

А. В. Мазуренко[1], Н. В. Болдырихин[2]**
[1] ООО «ДДОС-Гвард», г. Ростов-на-Дону, Российская Федерация
[2] Донской государственный технический университет, г. Ростов-на-Дону, Российская Федерация

Introduction. A rapid development of the systems such as Yandex, Google, etc., has predetermined the relevance of the task of searching substrings in a string, and approaches to its solution are actively investigated. This task is used to create database management systems that support associative search. Besides, it is applicable in solving information security issues and creating antivirus programs. Algorithms of searching substring in a string are used in signature-based discovery tasks.

Materials and Methods. The solution to the problem is based on the Aho-Corasick algorithm which is a typical technique of searching substrings in a string. At the same time, a new approach regarding preprocessing is employed.

Research Results. The possibility of constructing the transition function and suffix references through suffix arrays and special mappings, is shown. The relationship between the prefix tree and suffix arrays was investigated, which provided the development of a fundamentally new method of constructing the transition and error functions. The results obtained enable to substantially shorten the time intervals spent on the preelection processing of a set of pattern strings when using an integer alphabet. The paper lists eight algorithms. The developed algorithms are evaluated. The results obtained are compared to the formerly known. Two theorems and eight lemmas are proved. Two examples illustrating features of the practical application of the developed preprocessing procedure are given.

Discussion and Conclusions. The preprocessing procedure proposed in this paper is based on the communication between the suffix array built on the ground of a set of pattern strings and the construction of transition and error functions at the

Введение. Бурное развитие таких систем, как Yandex, Google и пр., предопределило актуальность задачи поиска подстрок в строке. На сегодняшний день активно исследуются подходы к ее решению. Эта задача используется при создании систем управления базами данных, поддерживающих ассоциативный поиск. Кроме того, она применима при решении вопросов информационной безопасности, создании антивирусных программ. Алгоритмы поиска подстрок в строке используются в задачах обнаружения, основанного на сигнатурах.

Материалы и методы. Решение задачи базируется на алгоритме Ахо — Корасик, который представляет собой классический способ осуществления поиска подстрок в строке. Вместе с тем применен новый подход в части, касающейся предварительной обработки.

Результаты исследования. Показана возможность построения функции перехода и суффиксных ссылок при помощи суффиксных массивов и специальных отображений. Исследована взаимосвязь между префиксным деревом и суффиксными массивами. Это дало возможность разработать принципиально новый способ построения функций перехода и ошибок.

Полученные результаты позволяют существенно сократить время, затрачиваемое на предвыборную обработку множества строк образцов при использовании целочисленного алфавита.

В статье приведено восемь алгоритмов. Оценены разработанные алгоритмы. Полученные результаты сопоставлены с ранее известными. Доказаны две теоремы и восемь лемм. Приведены два примера, иллюстрирующие особенности практического применения разработанной процедуры препроцессинга.

Обсуждение и заключения. Предложенная в данной статье процедура препроцессинга основывается на связи между суффиксным массивом, созданным на основе множества строк образцов, и построением функций перехода и ошибок на начальных этапах работы алгоритма Ахо — Корасик. Такой подход отличен от традиционного и требует

* The research is done within the frame of the independent R&D.
** E-mail: mazurencoal@gmail.com, boldyrikhin@mail.ru
*** Работа выполнена в рамках инициативной НИР.

initial stages of the Aho-Corasick algorithm. This approach differs from the traditional one and requires the use of algorithms providing a suffix array in linear time. Thus, the algorithms that enable to significantly reduce the time for preprocessing of a set of pattern strings under the condition of using a certain type of alphabet in comparison to the known approach proposed in the Aho- Corasick algorithm are described. The research results presented in the paper can be used in antivirus programs that apply searching for signatures of malicious data objects in the memory of a computer system. In addition, this approach to solving the problem on searching substrings in a string will significantly speed up the operation of database management systems using associative search.

Keywords: string searching, Aho-Corasick algorithm, prefix tree, suffix array, information search, error function, transition function

For citation: A.V. Mazurenko, N.V.Boldyrikhin. Accelerated preprocessing in task of searching substrings in a string. Vestnik of DSTU, 2019, vol. 19, no. 3, pp. 290–300. https://doi.org/10.23947/1992-5980-2019-19-3-290-300

использования алгоритмов, позволяющих построить суффиксный массив за линейное время. Таким образом, описаны алгоритмы, позволяющие существенно сократить время на предварительную обработку множества строк образцов при условии использования определенного типа алфавита по сравнению с известным подходом, предложенным А. Ахо и М. Корасик.

Результаты исследований, приведенные в статье, могут быть применены в антивирусных программах, использующих поиск сигнатур вредоносных информационных объектов в памяти вычислительной системы. Кроме того, данный подход к решению задачи поиска подстроки в строке позволяет значительно ускорить работу систем управления баз данных, применяющих ассоциативный поиск.

Ключевые слова: поиск подстроки, алгоритм Ахо — Корасик, префиксное дерево, суффиксный массив, поиск информации, функция ошибок, функция перехода.

Образец для цитирования: Мазуренко, А.В. Ускоренный препроцессинг в задаче поиска подстрок в строке / А. В. Мазуренко Н. В. Болдырихин // Вестник Дон. гос. техн. ун-та. — 2019. — Т. 19, № 3. — С. 290–300. https://doi.org/10.23947/1992-5980-2019-19-3- 290-300

Introduction. Nowadays, awareness of the cybersecurity of distributed information systems and individual computing facilities is growing essentially [1]. A range of such tasks is wide enough [1–10]. Of special interest is the creation of powerful antivirus software (SW). One of the most important tasks solved through such SW is searching substrings in a string [1, 5, 6, 10–13].

Materials and Methods. The task of substring searching is to find all the lines in the text T with a total length m matching any pattern from a given set of patterns P. Suppose that the sum of the lengths of all elements P consisting of characters of the alphabet I is n. A solution to this problem was proposed by A. Aho and M. Corasick [6, 10]. In their algorithm, the pre-election processing time is $O(n|I|)$, and the search time is $O(m|I|+k)$. Here, k is a number of matches found in the text with lines belonging to a set of samples.

Currently, the task of finding a substring in a string is being intensely investigated for two reasons:

- search engines are rapidly developing [11];
- the detection process in antivirus software products is based on signatures [1].

In this regard, algorithms have been created that have to be selected due to specific needs of the user. The latest results obtained under solving the problem of searching a set of substrings are described in [13].

The results presented in this paper are based on the relationship between the suffix array created from a set of pattern strings and the construction of transition and error functions at the initial stages of the Aho - Corasick algorithm. This approach differs from the traditional one and requires using algorithms to construct a suffix array in linear time. So, the paper describes the algorithms by which the pre-election processing time is reduced to $O(n)$.

Given the alphabet I, a set of patterns $P=\{P_1,P_2,...,P_k\}$ where $P_i \in I^*$, $i=\overline{1,k}$. Let us denote by $n=\sum_{i=1}^{k}|P_i|$. Assume that the alphabet I is a limited range of integers. The boundary may depend on the length of the string in question $s \in I^*$ or may involve an interval $[0,c]$ where c is a positive integer: $c \geq |s|$. Let $\varepsilon \in I$ be an empty string.

Let *goto* be a transition function and a *failure* — an error function. These modifications are concerned with the methods for constructing the mentioned functions used in the Aho – Corasick algorithm [6, 10].

Suppose $SuffArr(s)$ is a certain algorithm for constructing a suffix array for a string $s \in I^*$ in linear time. A description of such algorithms can be found, for example, in [12–15].

Suppose $x, y \in I^*$. Then, $lcp(x,y)$ is the largest common prefix of the strings x and y.

Consider the string $s \in I^*$, $s = s[s[0]s[1]...s[n-1]]$. Let $s[s[i]s[i+1]...s[j]]$ be a substring s including characters from i to j where $i \le j$, $i, j = \overline{0, n-1}$. Let us denote p_s by the suffix array corresponding to the string s. Suppose

$$p_s = p_s[p_s[0]p_s[1]...p_s[n-1]],$$

that is $s[s[p_s[0]]...s[n-1]] < s[s[p_s[1]]...s[n-1]] < ... < s[s[p_s[n-1]]...s[n-1]]$.

To construct a suffix array, the algorithm described in [15] will be used.

Suppose $\alpha_i \notin I$, $\alpha_i \ne \alpha_j$, $1 \le i < j \le k+1$, $\alpha_1 < \alpha_2 < ... < \alpha_{k+1}$. Let $\forall b \in I \ \ \alpha_i < b$, where $1 \le i \le k+1$. Granting $P \ne \varnothing$, $alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$.

Suffix Array Processing Algorithm p_s

Here, $s \in I^*$: $s = \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$, $P_i \in I^*$, $1 \le i \le k$.

$Adaptation(s, p_s, alpha)$

1. $new\_array \leftarrow \varepsilon$
2. $for\ (i \leftarrow |alpha|; i < |s|; i++)$ {
3. $j \leftarrow 0$
4. $while\ (s[s[p_s[i]]...s[|s|-1]][j] \notin alpha)$ {
5. $new\_array[i][j] \leftarrow s[s[p_s[i]]...s[|s|-1]][j]$
6. $j \leftarrow j + 1$
7. }
8. }
9. $ordered\_list[0] \leftarrow new\_array[0]$
10. $for\ (i \leftarrow 1; i < |s| - |alpha|; i++)$ {
11. $j \leftarrow 0$
12. $if\ (new\_array[i] \ne new\_array[i-1])$ {
13. $ordered\_list[i] \leftarrow new\_array[i]$
14. $j \leftarrow j + 1$
15. }
16. }
17. *return ordered_list*

Lemma 1. Let $P = \{P_1, P_2, \ldots, P_k\}$, $s = \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$. Then the *Adaptation* algorithm builds an array of lexicographically ordered suffixes of the patterns belonging to P over the time $O(|s| - |alpha|)$.

Proof. In the loop of 2–8, the construction of the *new_array* is performed, whose *i*-th element is a prefix of the corresponding suffix s which includes all the characters of this suffix starting with the zero position to its first element belonging to the set *alpha*. In this case, using the suffix array p_s, all suffixes s are looped over according to their lexicographic order. Thus, the *new_array* consists of all suffixes of the patterns belonging to P according to their lexicographic sequencing, and the recurrence of some suffixes is possible.

Note that all strings starting with characters belonging to the *alpha* array, that is, the first $|alpha|$ suffixes, are excluded from consideration. Then, in the loop of 10–16, using the *new_array*, the *ordered_list* array is constructed through eliminating repetition strings. To do this, due to the lexicographic sequence of the strings, it is sufficient to check whether the string in question coincides with the previous one.

The loop of 2–8 is executed over the time $O(|s| - |alpha|)$ since all strings starting with characters belonging to the *alpha* array are excluded from consideration. In the loop 10–16, $|s| - |alpha|$ of string matches occur. Thus, we obtain an asymptotic estimate of $O(|s| - |alpha|)$ algorithm running time. The lemma is proved.

Partitioning algorithm according to lexicographic sequencing

Here, s is an array of lexicographically ordered strings.

$DandC(s)$

1. $sub[0] \leftarrow 0$
2. $j \leftarrow 0$
3. $for\ (i \leftarrow 0; i < |s| - 1; i++)\ \{$
4. $if\ (s[i] \neq s[i+1])\ \{$
5. $\text{sub}[j] \leftarrow i+1$
6. $j \leftarrow j+1$
7. $\}$
8. $sub[j] \leftarrow |s|$
9. *return sub*

Lemma 2. The *DandC* algorithm based on an array of lexicographically ordered strings *s* constructs a *sub* array consisting of positive integers that show the indices corresponding to the first strings among the strings with the first characters equal over the time $O(|s|)$.

Proof. The boundary corresponding to the first character begins with 0, which corresponds to the assignment performed in step 1. In the loop of 3–7, the first characters of the *i*-th and (*i* + 1)-th strings are sequentially compared where $i = \overline{0, |s| - 2}$. If the characters are not equal, then the beginning of the boundary corresponding to the next character is written to the *sub* array. Otherwise, the loop execution continues. The right boundary of the last character corresponds to the number of strings in the *s* array (step 8).

The comparison in step 4 occurs over the time $O(1)$, as the recording in step 5 and the increment in step 6 do. Thus, the loop of 3–7 is performed over the time $O(|s|)$. The lemma is proved.

First link algorithm

Here, *tree* is a tree, $lex\_words \in I^*$, $link\_num$ is the number of some character in lex_words string, v is a serial number of a new node that joins the node with the serial number $node\_number$.

$BuildFirstLink(tree\&, lex\_words\&, v\&, link\_num, node\_number)$

1. $new\ tree.node[v]$
2. $tree.node[v].state \leftarrow lex\_words[lex\_words[0]...lex\_words[link]]$
3. $new\ tree.node[node\_number].link \leftarrow tree.node[v]$
4. $tree.node[node\_number].link.symbol \leftarrow lex\_words[link\_num]$
5. $v \leftarrow v+1$

Lemma 3. The *BuildFirstLink* algorithm constructs a new node with the sequence number *v* and an arc leading from *node_number* to a new node *v*, in the *tree* over the time $O(1)$.

Substring link algorithm

Here, *tree* is a tree, $lex\_words \in I^*$, v is a serial number of a new node that joins the node with the serial number $start$.

$BuildSubstringLink(tree\&, lex\_words\&, v\&, start)$

1. $for\ (k \leftarrow start; k < |lex\_words|; k++)\ \{$
2. $new\ tree.node[v]$
3. $tree.node[v].state \leftarrow lex\_words[lex\_words[0]...lex\_words[k]]$
4. $new\ tree.node[v-1].link \leftarrow tree.node[v]$
5. $tree.node[v-1].link.symbol \leftarrow lex\_words[k]$
6. $v \leftarrow v+1$
7. $\}$

Lemma 4. The *BuildSubstringLink* algorithm constructs new nodes in *tree* matching all prefixes of the string *lex_words* starting with the prefix $lex\_words[lex\_words[0]...lex\_words[start]]$ over the time $O(|lex\_words| - start)$.

Last link algorithm

Here, *tree* is a tree, $lex\_words \in I^*$, v is a serial number of a new arc, I is an alphabet.

$BuildLastLink(tree\&, lex\_words, v, I)$

1. $new\ tree.node[0].link[v] \leftarrow tree.node[0]$
2. $symbols \leftarrow \varnothing$
3. $for\ (i \leftarrow 0; i < |lex\_words|; i++)$ {
4. $symbols[i] \leftarrow lex\_words[i][0]$
5. $j \leftarrow 0$
6. $for\ (i \leftarrow 0; i < |I|; i++)$ {
7. if $(I[i] \notin symbols)$ {
8. $tree.node[0].link[v].symbol[j] \leftarrow I[i]$
9. $j \leftarrow j + 1$
10. }
11. }

Lemma 5. The *BuildLastLink* algorithm builds a loop at the root node. Its marking corresponds to a set of symbols by which it is impossible to go to other nodes of the *tree* from the root node over the time $O(|lex\_words| + |I|)$.

Transition Algorithm

Here, lex_words is an array of lexicographically ordered strings.

$CreateLink(lex\_words)$

1. $str \leftarrow \varnothing$
2. $sub \leftarrow \text{DandC}(lex\_words)$
3. $v \leftarrow 1$
4. $tree \leftarrow \varnothing$
5. $tree.node[0].state \leftarrow \varepsilon$
6. $for\ (i \leftarrow 0; i < |sub| - 1; i++)$ {
7. $BuildFirstLink(tree, lex\_words[sub[i]], v, 0, 0)$
8. $BuildSubstringLink(tree, lex\_words[sub[i]], v, 1)$
9. $for\ (j \leftarrow sub[i] + 1; j < sub[i+1]; j++)$ {
10. $temp \leftarrow |lcp(lex\_words[j-1], lex\_words[j])| + 1$
11. $z \leftarrow tree.getStateNumber(lcp(lex\_words[j-1], lex\_words[j]))$
12. $BuildFirstLink(tree, lex\_words[j], v, temp, z)$
13. $BuildSubstringLink(tree, lex\_words[j], v, temp)$
14. }
15. }
16. $BuildLastLink(tree, lex\_words, v, lex\_words)$
17. *return tree*

Lemma 6. The *CreateLink* algorithm builds a prefix *tree* with a loop at the root node over the time $O\left(\sum_{i=0}^{|lex\_words|-1} |lex\_words[i]|\right)$.

Proof. In step 2, the *DandC* algorithm is executed (see Lemma 2), after which, in step 5, the root node with the serial number 0 of the *tree* is created. Its state is taken equal to a blank string ε. Consider the loop of 6–15 at the *i*-th step.

In step 7, using the *BuildFirstLink* algorithm, a node is created whose state corresponds to the first character of the string $lex\_words[sub[i]]$. Given the construction of the *sub* array, it can be argued that such a character has not occurred before among the first characters of the previous strings. Then, in step 8, the implementation of the *BuildSubstringLink* algorithm sequentially creates nodes whose state matches all prefixes of the string $lex\_words[sub[i]]$ excluding the prefix built in the previous string.

In the loop of 9–14, using *BuildFirstLink* and *BuildSubstringLink* algorithms, we perform the same actions with strings lying in an integer space $[sub[i]+1, sub[i+1]-1]$. Since each such string has a common non-zero prefix with the previous string, the algorithm immediately switches to the state corresponding to the largest common prefix, starting with which, it is required to build new nodes. In step 16, using the *BuildLastLink* algorithm, a loop at the root node is created.

Steps 12 and 13 are performed over the time

$$O(1)+O\left(\left|lex\_words[j]\right|-\left|lcp\left(lex\_words[j-1], lex\_words[j]\right)\right|-1\right)=$$
$$=O\left(\left|lex\_words[j]\right|-\left|lcp\left(lex\_words[j-1], lex\_words[j]\right)\right|\right).$$

Thus, it follows from Lemmas 2, 3, and 4 that the loop of 9-14 is executed over the time

$$O\left(\sum_{j=sub[i]+1}^{sub[i+1]-1}\left|lex\_words[j]\right|-\left|lcp\left(lex\_words[j-1], lex\_words[j]\right)\right|\right).$$

The loop of 6–14 is executed over the time

$$O\left(\sum_{i=0}^{|sub|-2}\left|lex\_words[sub[i]]\right|\right)+$$
$$+O\left(\sum_{i=0}^{|sub|-2}\sum_{j=sub[i]+1}^{sub[i+1]-1}\left|lex\_words[j]\right|-\left|lcp\left(lex\_words[j-1], lex\_words[j]\right)\right|\right)=$$
$$=O\left(\sum_{i=0}^{|sub|-1}\left|lex\_words[sub[i]]\right|-\left|lex\_words\right|\right).$$

It follows from Lemma 5, that step 16 is performed over the time $O\left(\left|lex\_words\right|+\sum_{i=0}^{|lex\_words|-1}\left|lex\_words[i]\right|\right)$. Thus, we obtain an asymptotic estimate of the running time of the algorithm

$$O\left(\left|lex\_words\right|\right)+O\left(\sum_{i=0}^{|sub|-1}\left|lex\_words[sub[i]]\right|-\left|lex\_words\right|\right)+$$
$$+O\left(\left|lex\_words\right|+\sum_{i=0}^{|lex\_words|-1}\left|lex\_words[i]\right|\right)=O\left(\sum_{i=0}^{|lex\_words|-1}\left|lex\_words[i]\right|\right).$$

The lemma is proved.

The *goto* function algorithm

Here, P is a set of pattern strings.

$ConstructGoto(P)$

1. $s \leftarrow \alpha_1 P_1 \alpha_2 P_2 \ldots \alpha_k P_k \alpha_{k+1}$
2. $p_s \leftarrow SuffArr(s)$
3. $alpha \leftarrow \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$
4. $ordered\_list \leftarrow Adaptation(s, p_s, alpha)$
5. $j \leftarrow 0$
6. $lex\_words \leftarrow \varnothing$
7. $P\_length \leftarrow \{|P_1|, |P_2|, ..., |P_k|\}$
8. $for\ (i \leftarrow 0; i < |ordered\_list|; i++)$

9. *if* $\left(\left(\left|ordered\_list[i]\right| \in P\_length\right)\ and\ \left(ordered\_list[i] \in P\right)\right)$

10. $lex\_words[j++] \leftarrow ordered\_list[i]$

11. $goto \leftarrow CreateLink(lex\_words)$

12. *return goto*

We should remind that $n = \sum_{i=1}^{k} \left|P_i\right|$ for a set of patterns $P = \{P_1, P_2, \ldots, P_k\}$.

Theorem 1. The *ConstructGoto* algorithm develops the *goto* function over the time $O(n)$.

Proof. In step 2, a suffix array p_s for the string *s* is constructed. In step 4, using the *Adaptation* algorithm, all suffixes of the strings belonging to a set of patterns *P* are written to the *ordered_list* array. In this case, recurrences are not excluded. In the loop of 8–10, an array *lex_words* containing suffixes belonging to *P* and arranged in lexicographic sequence without recurrences is constructed. In step 11, a prefix tree is built with a loop at the root node based on the strings contained in the *lex_words* array. The data structure returned by the *CreateLink* algorithm defines exactly the *goto* function.

Step 2 is completed over the time $O(n+k+1)$ [12]. From Lemma 1, it follows that step 4 is completed over the time $O(n+k+1-k-1) = O(n)$. In the loop of 8–10, only strings whose length is equal to the length of any pattern are considered.

Thus, no more than $O(n)$ checks are needed to find patterns of *P*. From Lemma 6, it follows that step 11 is completed over the time $O\left(\sum_{i=0}^{|lex\_words|-1} \left|lex\_words[i]\right|\right) = O(n)$. Since $k \le n$, we obtain an asymptotic estimate of the running time of the $O(n)+O(n+k+1) = O(n)$ algorithm. The theorem is proved.

Research Results

Example 1.

Suppose $P = \{one, on, once, cell, lull, eye, near\}$. Then

$$s = \alpha_1 one \alpha_2 on \alpha_3 once \alpha_4 cell \alpha_5 lull \alpha_6 eye \alpha_7 near \alpha_8 . \quad (1)$$

Table 1 shows the result of the *goto* function algorithm on the entry of the string *s* (1).

Table 1

Prefix tree structure

node number	*node* state	*link* branched states from *node*	symbols on *link* branches from *node*
0	ε	1. *c*; 2. *e*; 3. *l*; 4. *n*; 5. *o*	1. *c*; 2. *e*; 3. *l*; 4. *n*; 5. *o*
1	*c*	1. *ce*	1. *e*
2	*ce*	1. *cel*	1. *l*
3	*cel*	1. *cell*	1. *l*
4	*cell*	–	–
5	*e*	1. *ey*	1. *y*
6	*ey*	1. *eye*	1. *e*
7	*eye*	–	–
8	*l*	1. *lu*	1. *u*
9	*lu*	1. *lul*	1. *l*
10	*lul*	1. *lull*	1. *l*
11	*lull*	–	–
12	*n*	1. *ne*	1. *e*
13	*ne*	1. *nea*	1. *a*
14	*nea*	1. *near*	1. *r*
15	*near*	–	–
16	*o*	1. *on*	1. *n*
17	*on*	1. *onc;* 2. *one*	1. *c*; 2. *e*
18	*onc*	1. *once*	1. *e*
19	*once*	–	–
20	*one*	–	–

Suppose $\tilde{s} = \alpha_{k+1}\tilde{P}_k\alpha_k \ldots \alpha_2\tilde{P}_1\alpha_1\tilde{P}_0\alpha_0$ is mirroring of the string *s*.

***Failure* function algorithm**

Here, P is a set of pattern strings.

FalseSuff(P)

1. $\tilde{s} \leftarrow \alpha_{k+1}\tilde{P}_k\alpha_k \ldots \alpha_2\tilde{P}_1\alpha_1\tilde{P}_0\alpha_0$
2. $p_{\tilde{s}} \leftarrow SuffArr(\tilde{s})$
3. $alpha \leftarrow \{\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_{k+1}\}$
4. ordered_list $\leftarrow Adaptation(\tilde{s}, p_{\tilde{s}}, alpha)$
5. $link \leftarrow \varnothing$
6. $for\ (i \leftarrow 0; i < |\tilde{s}|; i{+}{+})$
7. $inLink[i] \leftarrow \varepsilon$
8. sub $\leftarrow DandC(ordered\_list)$
9. $str \leftarrow \varnothing$
10. $for\ (i \leftarrow 0; i < |sub| - 1; i{+}{+})$
11. $for\ (j \leftarrow sub[i]; j < sub[i+1] - 1; j{+}{+})$
12. $str[j] \leftarrow |lcp(ordered\_list[j], ordered\_list[j+1])|$
13. $for\ (i \leftarrow 0; i < |sub| - 1; i{+}{+})$
14. $for\ (k \leftarrow sub[i+1] - 1; k > sub[i]; k{-}{-})$ {
15. $for\ (j \leftarrow sub[i]; j < k; j{+}{+})$
16. $min\_element \leftarrow \min(str[k-1], str[k-2], \ldots, str[j])$
17. $if\ (\min\_element = |ordered\_list[j]|)$
18. $\min\_temp[j - sub[i]] \leftarrow min\_element$
19. }
20. $max\_element \leftarrow \max(\min\_temp[0], \min\_temp[1], \ldots, \min\_temp[w])$
21. найти *max_index*: $\min\_temp[\max\_index] = \max\_element$
22. $inLink[k] \leftarrow ordered\_list[\max\_index + sub[i]]$
23. }
24. $for\ (i \leftarrow 0; i < |inLink|; i{+}{+})$ {
25. $link[i][0] \leftarrow ordered\_list[i]^{\sim}$; //string mirroring
26. $link[i][1] \leftarrow inLink[i]^{\sim}$; // string mirroring
27. }
28. *return link*

Remark. In string 20, $w \le sub[i+1] - sub[i] - 1$.

Theorem 2. The *FalseSuff* algorithm constructs the *failure* function over the time $O(n)$. Proof. In step 1, we construct an array of characters that contains mirror images of strings belonging to a set of patterns *P* and some unique characters. In step 2, we construct a suffix array $p_{\tilde{s}}$ for the string $\tilde{s}$. In step 4, using the *Adaptation* algorithm, all suffixes of the strings belonging to a set of patterns $\tilde{P}$ (a set of patterns consisting of mirrored strings *P*) are written to the *ordered_list* array, and recurrences are not excluded.

In step 8, the *DandC* algorithm is executed (see Lemma 2), after which, in the loop of 10–12, we find the length of the largest common prefix between the strings that match the first character. We write the result to the *str* array. Note that this value is zero for the strings for which this condition is not satisfied. In the loop of 13–23, a special mapping is constructed between the strings for which the first character matches. We describe this mapping. Indicate some string

$$s \in ordered\_list[ordered\_list[sub[i]], \ldots, ordered\_list[sub[i+1] - 1]].$$

Consider a set of strings belonging to $ordered\_list\left[ordered\_list\left[sub[i]\right],...,ordered\_list\left[sub[i+1]-1\right]\right]$. Their length is equal to the length of the largest common prefix with s excluding s itself. From this set, we find the string s' that has an overall length, and assign it to s. Obviously, the constructed mapping is a bijection under the condition of $s' \neq \varepsilon$. The result is written to the *inLink* array. In the loop of 24–27, using the *inLink* array, we explicitly indicate the constructed mapping while mirroring each of the strings. Thus, we assign the $\tilde{s}'$ node to the $\tilde{s}$ node of the prefix tree constructed on the basis of the array of patterns P. Its state is equal to the largest proper suffix $\tilde{s}$ that occurs among the many states of the considered prefix tree. But according to the definition of the *failure* function, this is the desired result.

Suppose $n=\sum_{i=1}^{k}\left|P_i\right|$. Step 2 is completed over the time $O(n+k+1)$ [12]. From Lemma 1, it follows that step 4 is performed over the time $O(n+k+1-k-1)=O(n)$. The loop of 6–7 is executed over the time $O\left(\left|\tilde{s}\right|\right)=O(n)$. From Lemma 2, it follows that step 8 is performed over the time $O(n)$. The loop of 10–12 is executed over the time $O\left(\sum_{i=0}^{|sub|-2}\sum_{j=sub[i]+1}^{sub[i+1]-1}\gamma_j\right)=O(k),\ \forall j\ \ \gamma_j=1$. The loop of 14–23 is completed over the time

$$O\left(\sum_{k=sub[i]+1}^{sub[i+1]-1}\sum_{j=sub[i]}^{k-1}\gamma_j\right)=O\left(\sum_{j=0}^{sub[i+1]-sub[i]-2}\gamma_j\right)=O\left(sub[i+1]-sub[i]-1\right),\ \forall j\ \ \gamma_j=1.$$

Then the loop of 13–23 is executed over the time

$$O\left(\sum_{i=0}^{|sub|-2}\left(sub[i+1]-sub[i]-1\right)\right)=O\left(sub\left[|sub|-1\right]\right)=O(k).$$

Since $\left|inLink\right|<n$, then the loop of 24–27 is executed over the time $O\left(\left|inLink\right|\right)=O(n)$. Thus, considering that $k\le n$, we obtain an asymptotic estimate of the running time of the $O(n)+O(n+k+1)+O(k)=O(n)$ algorithm. The theorem is proved.

Example 2.

Suppose, as in example 1, $P=\{one,on,once,cell,lull,eye,near\}$. Then

$$\tilde{s}=\alpha_8 rae n\alpha_7 eye\alpha_6 llul\alpha_5 llec\alpha_4 ecno\alpha_3 no\alpha_2 eno\alpha_1. \quad (2)$$

Table 2 shows the result of the *failure* function algorithm on the entry of the string $\tilde{s}$ (2).

Table 2

False links between nodes

inLink array	*link* array	*inLink* array	*link* array
0. ε	0. 0. *nea*; 1. ε	10. *l*	10. 0. *cel*; 1. *l*
1. ε	1. 0. *c*; 1. ε	11. *l*	11. 0. *cell*; 1. *l*
2. *c*	2. 0. *onc*; 1. *c*	12. *l*	12. 0. *lull*; 1. *l*
3. ε	3. 0. *e*; 1. ε	13. *l*	13. 0. *lul*; 1. *l*
4. *e*	4. 0. *ce*; 1. *e*	14. ε	14. 0. *n*; 1. ε
5. *ec*	5. 0. *once*; 1. *ce*	15. *n*	15. 0. *on*; 1. *n*
6. *e*	6. 0. *ne*; 1. *e*	16. ε	16. 0. *o*; 1. ε
7. *en*	7. 0. *one*; 1. *ne*	17. ε	17. 0. *near*; 1. ε
8. *e*	8. 0. *eye*; 1. *e*	18. ε	18. 0. *lu*; 1. ε
9. ε	9. 0. *l*; 1. ε	19. ε	19. 0. *ey*; 1. ε

For all nodes for which Fig. 1 does not show false links, we believe that a false link leads to a root node.

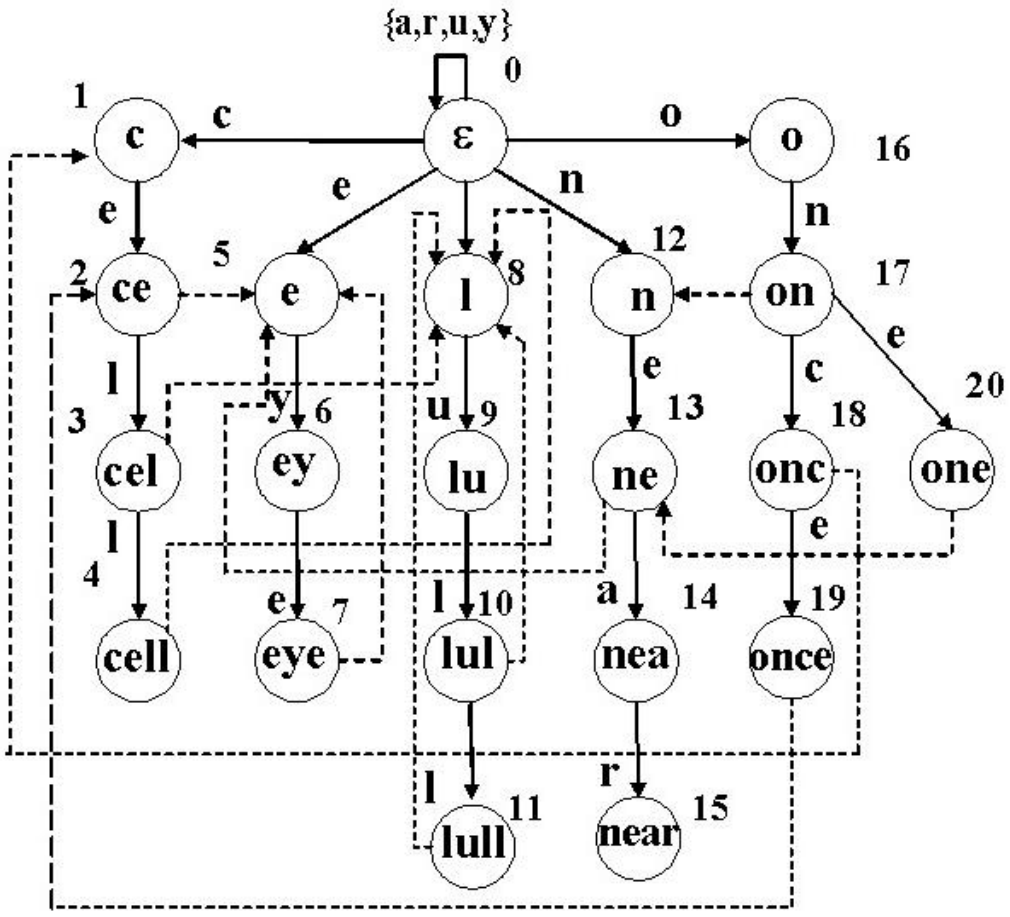

Fig. 1. Prefix tree with false links

Discussion and Conclusions. A new preprocessing procedure in the Aho-Corasick algorithm is described. It runs in the linear time $O(n)$. The connection between the suffix arrays and a prefix tree was investigated which allowed us to propose a different way of constructing transition and failure functions. The results obtained provide reducing the time on the pre-election processing of a set of pattern strings when using the integer alphabet.

References

1. Stallings, W. Computer security: principles and practice. Boston: Pearson, 2012, 182 p.

2. Chernyshev, Y.O., et al. Issledovanie vozmozhnosti primeneniya geneticheskikh algoritmov dlya realizatsii kriptoanaliza blochnykh kriptosistem. [Feasibility study of genetic algorithms application for implementation of block cryptosystem cryptanalysis.] Vestnik of DSTU, 2015, vol. 15, no. 3 (82), pp. 65–72 (in Russian).

3. Sadovoy, N.N., Kosolapov, Yu.V., Mkrtichyan, V.V. Programmnye utility dlya kontrolya i predotvrash-cheniya setevykh atak na urovne dostupa k seti. [Software utilities to control and prevent network attacks at the network access level.] Vestnik of DSTU, 2005, vol. 5, no. 2 (24), pp. 173–178 (in Russian).

4. Mogilevskaya, N.S., Kulbikayan, R.V., Zhuravlev, L.A. Porogovoe razdelenie faylov na osnove bitovykh masok: ideya i vozmozhnoe primenenie. [Threshold file sharing based on bit masks: concept and possible use.] Vestnik of DSTU, 2011, vol. 11, no. 10, pp. 1749–1755 (in Russian).

5. Sheludko, A.A., Boldyrikhin, N.V. Poisk informatsionnykh ob"ektov v pamyati komp'yutera pri reshenii zadach obespecheniya kiberbezopasnosti. [Search of information objects in computer memory solving the problems of cyber security provision.] Young Researcher of the Don, 2018, no. 6 (15), pp. 81–86 (in Russian).

6. Mazurenko, A.V., Boldyrikhin, N.V. Obnaruzhenie, osnovannoe na signaturakh, s ispol'zovaniem algoritma Akho — Korasik. [Signature-based detection using the Aho-Corasick algorithm.] Proc. of North Caucasus Branch of Moscow Tech. University of Communications and Informatics, 2016, no. 1, pp. 339–344 (in Russian).

7. Mazurenko, A.V., Arkhangelskaya, N.S., Boldyrikhin, N.V. Algoritm proverki podlinnosti pol'zovatelya, osnovannyy na graficheskikh klyuchakh. / [User authentication algorithm based on pattern locks.] Young Researcher of the Don, 2016, no. 3 (3), pp. 92–95 (in Russian).

8. Mazurenko, A.V., Stukopin, V.A. Geometricheskaya realizatsiya metoda provedeniya elektronnykh vyborov, osnovannogo na porogovom razdelenii sekreta. [Geometric realization of electronic elections based on threshold secret sharing.] Vestnik of DSTU, 2018, vol. 18, no. 2, pp. 246–255 (in Russian).

9. Cherkesova, L.V., et al. Algoritmicheskaya otsenka slozhnosti sistemy kodirovaniya i zashchity informatsii, osnovannoy na porogovom razdelenii sekreta, na primere sistemy elektronnogo golosovaniya. [Complexity calculation of coding and information security system based on threshold secret sharing scheme used for electronic voting.] Vestnik of DSTU, 2017, vol. 17, no. 3, pp. 145–155 (in Russian).

10. Antonov, E.S. Kak nayti million. Sravnenie algoritmov poiska mnozhestva podstrok. [How to find a million. Substring set searching algorithms comparison.] RSDN Magazine, 2011, no. 1, pp. 60–67 (in Russian).

11. Tarakeswar, M.K., Kavitha, M.D. Search Engines: A Study. Journal of Computer Applications, 2011, vol. 4, no. 1, pp. 29–33.

12. Karkkainen, J., Sanders, P., Burkhardt, S. Linear work suffix array construction. Journal of the ACM, 2006, vol. 53, no. 6, pp. 918–936.

13. Baklanovsky, M.V. Khanov, A.R. Povedencheskaya identifikatsiya program. [Identification of programs based on the dehavior.] Modeling and Analysis of Information Systems, 2014, vol. 21, no. 6, pp. 120–130 (in Russian).

14. Becher, V., et al. Efficient repeat finding in sets of strings via suffix arrays. Discrete Mathematics and Theoretical Computer Science, 2013, vol. 15, no. 2, pp. 59–70.

15. Shrestha, A.M.S., Frith, M.C., Horton, P. A bioinformatician's guide to the forefront of suffix array construction algorithms. Briefings in Bioinformatics, 2014, vol. 15, no. 2, pp. 138–154.

Submitted 22.01.2019
Scheduled in the issue 12.04.2019

Authors:

Mazurenko, Alexander V.
mathematician-programmer, DDoS-GUARD LLC (62/2, Budenovsky Pr., Rostov-on-Don, 344002, RF)
ORCID: http://orcid.org/0000–0001–9541–3374
mazurencoal@gmail.com

Boldyrikhin, Nikolay V.
associate professor of the Cybersecurity of IT Systems Department, Don State Technical University
(1, Gagarin Square, Rostov-on-Don, 344000, RF), Cand.Sci. (Eng.)
ORCID: http://orcid.org/0000–0002–9896–9543,
boldyrikhin@mail.ru