KMP Based Pattern Matching Algorithms for Multi-Track Strings

Diptarama, Yohei Ueki, Kazuyuki Narisawa, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University, Japan
{diptarama@shino.,yohei_ueki@shino.,narisawa@,ayumi@}ecei.tohoku.ac.jp

Abstract. Multi-track string is an N-tuple strings of length n. For two multi-track strings $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ of length n and $\mathbb{P} = (p_1, p_2, ..., p_M)$ of length m, permuted pattern matching is a problem to find all positions i such that $\mathbb{P}$ is permuted match with $\mathbb{T}[i : i+M]$. We propose three new algorithms for permuted pattern matching based on the KMP algorithm. The first algorithm is an exact matching algorithm that runs in $O(nN)$ time after $O(mM)$-time preprocessing. The second and third algorithms are filtering algorithms that run in $O(n(N + \sigma))$ after $O(m(M + \sigma))$-time preprocessing, where σ is the size of alphabet. Experiments show that our algorithms run faster than AC automaton based algorithm that proposed by Katsura *et al.* [5].

Keywords: string matching, multi-track, KMP algorithm

1 Introduction

Pattern matching problem on strings is defined as finding all occurences of a pattern string in a text string. Many of pattern matching algorithms can find occurences of the pattern fast by preprocessing the pattern such as AC automaton [1], and KMP algorithm [7], or by constructing data structure from the text such as suffix tree [13], suffix array [9], and position heap [2].

Katsura *et al.* [5, 6] defined a set (or tuple) of strings be a *multi-track string*, and formulated the pattern matching problem on multi-track strings, called *permuted pattern matching*. In order to solve this problem, they proposed permuted pattern matching algorithms by constructing data structure from text such as multi-track suffix tree [5], multi-track position heap [6], or by preprocessing the the pattern such as AC automaton based algorithm [5].

In this paper, we propose three new algorithms for multi-track pattern matching problem, based on KMP algorithm [7][1]. The first one is a full-permuted pattern matching algorithm, that we call it MTKMP for short. Second and third are filtering algorithms for multi-track full- and sub- permuted matching problem, respectively. For short, we call them Filter-MTKMP-Full and Filter-MTKMP, respectively. Different with AC automaton based algorithm [5] that constructs

[1] We implement MP algorithm for the first algorithm and KMP algorithm for the second and third algorithms.

failure functions for each track of the pattern, our algorithms only construct one failure function for a multi-track pattern. Also, our algorithms match the whole tracks of the pattern to the text instead of matching each track of the text independently.

We show that MTKMP runs in $O(mM)$ time for constructing the failure function, and $O(nN)$ for matching. Both Filter-MTKMP-Full and Filter-MTKMP run in $O(m(M+\sigma))$ time for constructing the failure function, $O(n(N+\sigma))$ for filtering, and $O(mM)$ to verify each candidate. Moreover, the experiment results show that proposed algorithms run faster than AC automaton based multi-track pattern matching algorithm [5] for full-permuted matching, and Filter-MTKMP works faster on sub-permuted-matching when the alphabet size and the track count of the pattern are large.

2 Preliminaries

Let $w \in \Sigma^n$ be a string of length n over an alphabet Σ, and $\sigma = |\Sigma|$ be the alphabet size. $|w|$ denotes the length of w and $w[i]$ denotes i-th character of w. The substring of w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \le i \le j \le |w|$. In addition, $w[:i] = w[1:i]$ and $w[i:] = w[i:|w|]$ denotes a *prefix* and a *suffix* of w, respectively. For two strings x and y, $x \prec y$ denotes that x is lexicographically smaller than y, and $x \preceq y$ denotes that either x equals to y or $x \prec y$.

A *multi-track string* (or *multi-track* for short) $\mathbb{W} = (w_1, w_2, ..., w_N)$ is an N-tuple of strings $w_i \in \Sigma^n$, and each w_i is called the *i-th track* of $\mathbb{W}$. The length n of a multi-track is denoted by $|\mathbb{W}|_{len}$. The number N of tracks in a multi-track is called *track count* and denoted by $|\mathbb{W}|_{num}$. For two multi-tracks $\mathbb{X} = (x_1, x_2, .., x_N)$ and $\mathbb{Y} = (y_1, y_2, .., y_N)$, we write $\mathbb{X} = \mathbb{Y}$ if $x_i = y_i$ for $1 \le i \le N$. $\mathbb{W}[i]$ denotes $(w_1[i], w_2[i], ...w_N[i])$, and $\mathbb{W}[i:j]$ denotes $(w_1[i:j], w_2[i:j], \ldots, w_N[i:j])$ for $1 \le i \le j \le |\mathbb{W}|_{len}$. Moreover, the prefix and suffix of $\mathbb{W}$ are denoted by $\mathbb{W}[:i] = \mathbb{W}[1:i]$ and $\mathbb{W}[i:] = \mathbb{W}[i:|\mathbb{W}|_{len}]$, respectively.

Let $\mathbf{r} = (r_1, r_2, \ldots, r_M)$ be a partial permutation of $(1, 2, \ldots, N)$ for $M \le N$. For a multi-track $\mathbb{W} = (w_1, w_2, \ldots, w_N)$, a *permuted multi-track* of $\mathbb{W}$ is denoted by either $\mathbb{W}\langle r_1, r_2, \ldots, r_M\rangle$ or $\mathbb{W}\langle\mathbf{r}\rangle$. Sorted index of a multi track $SI(\mathbb{W}) = (r_1, r_2, \ldots, r_N)$ is defined as a permutation such that $w_{r_i} \preceq w_{r_j}$ for any $1 \le i \le j \le N$. For two multi-tracks $\mathbb{X} = (x_1, x_2, \ldots, x_M)$ and $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \overset{\bowtie}{\sqsubseteq} \mathbb{Y}$, if $\exists\mathbf{r}.\mathbb{X} = \mathbb{Y}\langle\mathbf{r}\rangle$, and $\mathbb{X}$ *full-permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \overset{\bowtie}{=} \mathbb{Y}$, if $M = N$ and $\exists\mathbf{r}.\mathbb{X} = \mathbb{Y}\langle\mathbf{r}\rangle$.

Throughout the paper, we assume that $\mathbb{P}$ is a pattern with $|\mathbb{P}|_{num} = M$ and $|\mathbb{P}|_{len} = m$, and $\mathbb{T}$ is a text with $|\mathbb{T}|_{num} = N$ and $|\mathbb{T}|_{len} = n$. The pattern matching problem on multi-tracks are defined as follows.

Problem 1 (Permuted pattern matching). Given a multi-tracks text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$, output all positions i that satisfy $\mathbb{P} \overset{\bowtie}{\sqsubseteq} \mathbb{T}[i : i+m-1]$.

Moreover, we call it *full-permuted pattern matching* if $M = N$, and *sub-permuted pattern matching* if $M < N$.

Algorithm 1: MTKMP pattern matching algorithm

```
Input: Multi-track 𝕋, Multi-track ℙ
Output: match positions
compute SI(𝕋[i :]) for 1 ≤ i ≤ n and SI(ℙ[i :]) for 1 ≤ i ≤ m;
constructFailureFunction();
i = 1;  j = 1;
while i ≤ n do
    while j > 0 and 𝕋[i]⟨SI(𝕋[i − j + 1 :])⟩ ≠ ℙ[j]⟨SI(ℙ[1 :])⟩ do  j = F[j];
    i = i + 1;  j = j + 1;
    if j > m then
        output (i − j + 1);
        j = F[j];
Function constructFailureFunction()
    i = 1;  j = 1;  F[1] = 0;
    while i ≤ m do
        while j > 0 and ℙ[j]⟨SI(ℙ[1 :])⟩ ≠ ℙ[i]⟨SI(𝕋[i − j + 1 :])⟩ do  j = F[j];
        i = i + 1;  j = j + 1;
        F[i] = j;
```

In this section, we propose an algorithm for full-permuted pattern matching that based on KMP algorithm, named multi-track KMP algorithm (shortly MTKMP). In a similar manner to the original KMP algorithm, MTKMP constructs the failure function from the pattern, then uses it to shift the pattern when mismatch occurs. The MTKMP algorithm is described in Algorithm 1.

The failure function $F[i]$ for $1 \leq i \leq m+1$ in MTKMP is defined as the length of the longest proper suffix of $\mathbb{P}[1 : i-1]$ that permuted-matches with a prefix of $\mathbb{P}$, except $F[1] = 0$ and $F[2] = 1$. In order to construct the failure function, first MTKMP finds the value of sorted index $SI(\mathbb{P}[i : m])$ for all $1 \leq i \leq m$. Then permuted-matching of suffix $\mathbb{P}[j : i]$ and prefix $\mathbb{P}[1 : i - j + 1]$ is performed by using $SI(\mathbb{P}[1 :])$ and $SI(\mathbb{P}[j :])$.

MTKMP performs permuted pattern matching from left to right of the pattern and the text. Sorted indexes of the text and the pattern are used to perform permuted-matching between the pattern and a substring of the text. If characters on text and pattern are mismatched, then we shift the position of the pattern by using the failure function. If the pattern matches a substring of the text, then MTKMP outputs the position of the text and shifts the pattern according to the failure function.

The MTKMP algorithm runs in $O(mM)$ time for constructing the failure function and $O(nN)$ time for matching. Suffix index $SI(\mathbb{P}[i :])$ for $1 \leq i \leq m$ and $SI(\mathbb{T}[i :])$ for $1 \leq i \leq n$ can be computed in $O(mM)$ and $O(nN)$ respectively by using suffix tree [3, 10, 12, 13] or suffix array [4, 8, 9, 11]. Next, both the outer **while** loop and the inner **while** loop are called $O(m)$ times, since the $i - j \leq i \leq m + 1$ and the value of i always increase each time the outer loop is called, and the value of $i - j$ always increases each time the inner loop is called. Since

Algorithm 2: Filter-MTKMP-Full pattern matching algorithm

```
Input: Multi-track 𝕋, Multi-track ℙ
Output: match position
1  𝕋′ = func(𝕋);  ℙ′ = func(ℙ);
2  constructFailureFunction();
3  i = 1,  j = 1;
4  while i ≤ n do
5      while j > 0 and 𝕋′[i] ≠ ℙ′[j] do  j = F[j];
6      i = i + 1;  j = j + 1;
7      if j > m then
8          if 𝕋[i − j + 1 : i − 1] ⋈̸ ℙ then  output (i − j + 1);
9          j = F[j];
10 Function constructFailureFunction()
11     i = 1;  j = 1;  F[1] = 0;
12     while i ≤ m do
13         while j > 0 and ℙ′[i] ≠ ℙ′[j] do  j = F[j];
14         i = i + 1;  j = j + 1;
15         if i ≤ m and ℙ′[i] = ℙ′[j] then  F[j] = F[i];  else  F[j] = i;
```

a comparison of $\mathbb{P}[j]\langle SI(\mathbb{P}[1:])\rangle$ and $\mathbb{P}[i]\langle SI(\mathbb{T}[i-j+1:])\rangle$ consumes $O(M)$ time, *constructFailureFunction* function runs in $O(Mm)$ time. In a similar way, the search algorithm of MTKMP runs in $O(Nn)$ time.

3 KMP Algorithm for Filtering on Multi-Track String

In this section we propose two filtering algorithms that can outperform MTKMP for permuted pattern matching problems. Instead of the suffix index, we use simple functions to transforms the multi-track string, that can be computed faster than the suffix index. Then, by transforming the pattern and the text, the KMP algorithm can be applied to find candidates of permuted-matching positions. Finally, every candidate position is checked whether or not the pattern is permuted-matched in each position.

The first algorithm is an algorithm for full-permuted pattern matching problem called Filter-MTKMP-Full. We use two functions in this algorithm. Generally, we can implement another function that has a *false positive* property. The second algorithm, Filter-MTKMP is an algorithm for both full- and sub-permuted pattern matching problems. Filter-MTKMP transforms multi-track into alphabet bucket and implements KMP algorithm to it.

3.1 Full-Permuted Pattern Matching Algorithm

The Filter-MTKMP-Full algorithm is described in Algorithm 2. First, we transform $\mathbb{P}$ by a function *func*() that has a *false-positive* property, that is if $\mathbb{X} \bowtie \mathbb{Y}$ then $func(\mathbb{X}) = func(\mathbb{Y})$. We propose two functions, *sort*() and *bucket*().

Definition 1. *For* $\mathbb{Z} = (z_1, z_2, \ldots, z_N)$*,* $sort(\mathbb{Z}) = \mathbb{Z}' = (z'_1, z'_2, \ldots, z'_N)$ *such that* $\exists \mathbf{r}.\mathbb{Z}'[i] = \mathbb{Z}[i]\langle \mathbf{r} \rangle$ *and* $z'_j[i] \preceq z'_{j+1}[i]$ *for* $1 \leq i \leq |\mathbb{Z}|_{len}$ *and* $1 \leq j < N$*.*

Definition 2. *For* $\mathbb{Z} = (z_1, z_2, \ldots, z_N)$*,* $bucket(\mathbb{Z}) = (b_1, b_2, \ldots, b_\sigma)$ *such that* $b_j[i]$ *is the number of the lexicographical* j*-th character in* $\mathbb{Z}[i]$*.*

For example, for a multi-track $\mathbb{Z} = (\texttt{abab}, \texttt{bbac}, \texttt{aabb}, \texttt{cabb}, \texttt{abba})$, the $sort(\mathbb{Z})$ and $bucket(\mathbb{Z})$ are defined as follows,

$$\mathbb{Z} = \begin{pmatrix} \texttt{a}\ \texttt{b}\ \texttt{a}\ \texttt{b} \\ \texttt{b}\ \texttt{b}\ \texttt{a}\ \texttt{c} \\ \texttt{a}\ \texttt{a}\ \texttt{b}\ \texttt{b} \\ \texttt{c}\ \texttt{a}\ \texttt{b}\ \texttt{b} \\ \texttt{a}\ \texttt{b}\ \texttt{b}\ \texttt{a} \end{pmatrix}, \quad sort(\mathbb{Z}) = \begin{pmatrix} \texttt{a}\ \texttt{a}\ \texttt{a}\ \texttt{a} \\ \texttt{a}\ \texttt{a}\ \texttt{a}\ \texttt{b} \\ \texttt{a}\ \texttt{b}\ \texttt{b}\ \texttt{b} \\ \texttt{b}\ \texttt{b}\ \texttt{b}\ \texttt{b} \\ \texttt{c}\ \texttt{b}\ \texttt{b}\ \texttt{c} \end{pmatrix}, \quad bucket(\mathbb{Z}) = \begin{pmatrix} 3\ 2\ 2\ 1 \\ 1\ 3\ 3\ 3 \\ 1\ 0\ 0\ 1 \end{pmatrix}.$$

For a multi-track $\mathbb{Z}$, $sort(\mathbb{Z})$ can be computed in $O(n(N + \sigma))$ by using the bucket sorting algorithm, and $bucket(\mathbb{Z})$ can be computed in $O(nN)$ time by counting all characters in $\mathbb{Z}$ even if naively.

Next, Filter-MTKMP-Full constructs the failure function F of the transformed multi-track $\mathbb{P}'$ by similar algorithm as the KMP algorithm.

Pattern matching in Filter-MTKMP-Full is performed on $\mathbb{P}'$ and $\mathbb{T}'$. When unmatched is occurred, the pattern is shifted by using the failure function. If $\mathbb{P}'$ matches to $\mathbb{T}'[i : j]$, then the algorithm checks whether the pattern $\mathbb{P}$ is permuted-matched with $\mathbb{T}[i : j]$ or not, and outputs it if the pattern is permuted-matched.

In the same way as described in MTKMP, the failure function can be constructed in $O(m(M + \sigma))$ time and the filtering algorithm runs in $O(n(N + \sigma))$ time, since $sort()$ and $bucket()$ needs $O(M)$ and $O(\sigma)$ time for each comparison, respectively. Also, the Filter-MTKMP-Full algorithm runs in $O(m(M + \sigma))$ time for constructing the failure function. In addition, Filter-MTKMP-Full need $O(cmM)$ time for checking the candidates, where c is the number of candidates.

3.2 Permuted Pattern Matching Algorithm

Filter-MTKMP algorithm is an extended version of the Filter-MTKMP-Full algorithm that can be applied to the sub-permuted pattern matching problem. This algorithm uses $bucket()$ to transforms the pattern and the text, and implement the KMP algorithm to the transformed pattern and text. Moreover, this algorithm uses $diff(X, Y)$ function instead of $X \neq Y$ to loosen the condition, thus sub-permuted pattern matching can be performed.

Algorithm 3 shows a pseudocode of the Filter-MTKMP algorithm. First, it transforms $\mathbb{P}$ to multi-track bucket $\mathbb{P}'$ and constructs the failure function F of the multi-track bucket by using a function $diff(X, Y) = \sum_{i=1}^{\sigma} max(0, Y[i] - X[i])$ for two integer vectors X and Y. Then, it finds candidates of matched position by using the failure function.

Last, in the similar way as Filter-MTKMP-Full, the Filter-MTKMP algorithm runs in $O(m(M + \sigma))$ time for constructing the failure function, $O(n(N +$

Algorithm 3: Filter-MTKMP pattern matching algorithm

```
Input: Multi-track 𝕋, Multi-track ℙ
Output: match position
1  𝕋′ = bucket(𝕋); ℙ′ = bucket(ℙ);
2  constructFailureFunction();
3  i = 1; j = 1;
4  while i ≤ n do
5      while j > 0 and diff(ℙ′[j], 𝕋′[i]) > 0 do j = F[j];
6      i = i + 1; j = j + 1;
7      if j > m then
8          if 𝕋[i − j + 1 : i − 1] ⋤⋥ ℙ then output (i − j + 1);
9          j = F[j];
10 Function constructFailureFunction()
11     i = 1; j = 1; F[1] = 0;
12     while i ≤ m do
13         while j > 0 and diff(ℙ′[i], ℙ′[j]) > N − M do j = F[j];
14         i = i + 1; j = j + 1;
15         if i ≤ m and ℙ′[i] = ℙ′[j] then F[j] = F[i]; else F[j] = i;
16 Function diff(Vector X, Vector Y)
17     Int sum = 0;
18     for k = 1 to |X| do
19         sum = sum + max(0, Y[k] − X[k]);
20     return sum;
```

$\sigma)$) time for finding candidates, and $O(cmM)$ time for checking the candidates, where c is the number of candidates.

4 Experiments

In order to evaluate the performance of proposed algorithms, we conduct experiments and compare the running time of proposed algorithms with AC automaton based algorithm [5].

In the first experiment, we compare running time of the algorithms for solving full-permuted pattern matching. We change one of the parameters while keeping other parameters constant. We set constant parameter as follows, $n = 100000$, $m = 10$, $N = M = 1000$, and $\sigma = 2$. We use random text and pattern with 50 occurrences of the pattern inserted to the text. Figure 1 (a)-(d) show running time of the algorithms when one of the parameters n, N, m, and σ is changed respectively. We can see that the proposed algorithms are faster than AC automaton based algorithm. However, many false positive candidates are found when the length of the pattern is short, and filtering algorithms need lots of time to verify them.

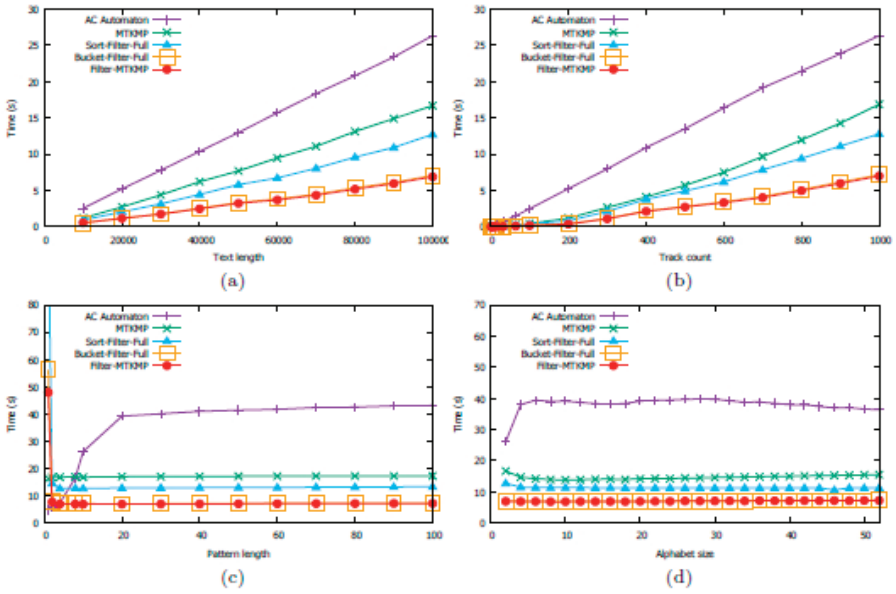

Fig. 1. Running time of the algorithms on full-permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

The second experiment measures the running time of Filter-MTKMP on sub-permuted pattern matching, and compares it with AC automaton algorithm. We set $n = 10000$, $N = 1000$, $m = 10$, $M = 600$, and $\sigma = 26$ as constant parameters. Figure 2 (a) and (b) show that running time of Filter-MTKMP is decreased when the track count of the pattern and the alphabet size are increased, because the number of false-positive candidates are decreased if the track count is increased. We can conclude that Filter-MTKMP performs better if the difference between

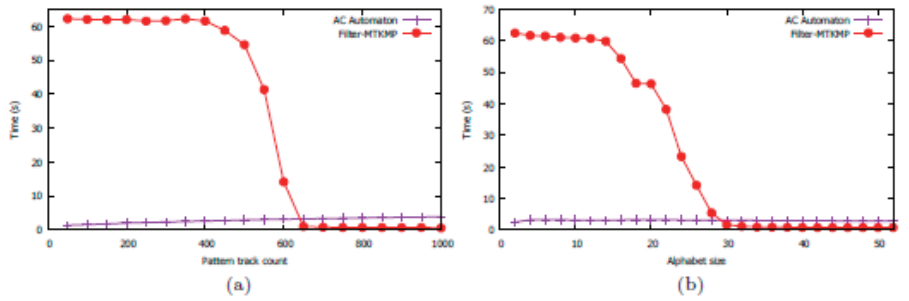

Fig. 2. Running time of the algorithms on sub-permuted pattern matching with respect to (a) pattern track and (b) alphabet size.

track count of the text and track count of the pattern is small, also when the alphabet size is large.

5 Conclusion

We proposed three algorithms based on the KMP algorithm, that are MTKMP, Filter-MTKMP-Full and Filter-MTKMP. These algorithms can solve the permuted pattern matching problem in linear time. We also conducted computational experiments, and showed that proposed algorithms work faster than AC-automaton based algorithm.

Acknowledgments. This work was partly supported by KAKENHI Grant Numbers 15H05706, 25560067 and 25240003. This work was also supported by research grant and scholarship from Tohoku University Division for International Advanced Research and Education.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6) 333–340 (1975)
2. Ehrenfeucht, A., McConnell, R.M., Osheim, N., Woo, S.W.: Position heaps: A simple and dynamic text indexing data structure. Journal of Discrete Algorithms 9(1) 100–121 (2011)
3. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS. 137–143 (1997)
4. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: ICALP. 943–955 (2003)
5. Katsura, T., Narisawa, K., Shinohara, A., Bannai, H., Inenaga, S.: Permuted pattern matching on multi-track strings. In: SOFSEM. 280–291 (2013)
6. Katsura, T., Otomo, Y., Narisawa, K., Shinohara, A.: Position heaps for permuted pattern matching on multi-track strings. In: Proceedings of Student Research Forum Papers and Posters at SOFSEM 2015. 41–531 (2015)
7. Knuth, D.E., Jr., J.H.M., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2) 323–350 (1977)
8. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: CPM. 200–210 (2003)
9. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5) 935–948 (1993)
10. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2) 262–272 (1976)
11. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Data Compression Conference. 193–202 (2009)
12. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3) 249–260 (1995)
13. Weiner, P.: Linear pattern matching algorithms. In: SWAT. 1–11 (1973)