Position Heaps for Permuted Pattern Matching on Multi-Track Strings

Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara

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

Abstract. A multi-set of N strings of length n is called a multi-track string. The permuted pattern matching is the problem that given two multi-track strings $\mathbb{T} = \{t_1, \ldots, t_N\}$ of length n and $\mathbb{P} = \{p_1, \ldots, p_N\}$ of length m, outputs all positions i such that $\{p_1, \ldots, p_N\} = \{t_1[i : i+m-1], \ldots, t_N[i : i+m-1]\}$ We propose two new indexing structures for multi-track stings. One is a time-efficient structure for $\mathbb{T}$ that needs $O(nN)$ space and enables us to solve the problem in $O(m^2N + occ)$ time, where occ is the number of occurrences of the pattern $\mathbb{P}$ in the text $\mathbb{T}$. The other is memory-efficient, it requires only $O(n)$ space, whereas the matching consumes $O(m^2N^2 + occ)$ time. We show that both of them can be constructed in $O(nN)$ time.

Keywords: string matching, multi-track, indexing structure

1 Introduction

The string indexing problem is fundamental and important for information retrieval, and to build an index for a given length n text string that allows us to find all occurrences of a given length m pattern string in the text efficiently. The classical indexing structures, suffix trees [16] and suffix arrays [11], require $O(n)$ space and can be built in $O(n)$ time on a constant-size alphabet [6, 7, 9, 12, 13, 15]. By using suffix trees and suffix arrays, all occurrences of a pattern can be reported in $O(m + occ)$ and $O(m \log n + occ)$ time, respectively, where occ is the total number of occurrences of the pattern in the text.

Ehrenfeucht et al. [5] proposed more space efficient indexing structure called position heaps, which requires $O(n)$ space but the number of nodes in the position heaps is at most $n + 1$ although that of the suffix tree is at most $2n - 1$.

Kucherov [**?**] showed an Ukkonen-like on-line $O(n)$-time algorithm for constructing position heaps. By using position heaps, the occurrences of the pattern can be found in $O(m^2 + occ)$ time. To improve its time bound to $O(m + occ)$, Ehrenfeucht et al. [5] proposed $O(n)$-space auxiliary structure, called the maximal-reach pointers (shortly MRPs).

Recently, Katsura et al. [8] proposed a new framework of the string matching problem, called the *permuted pattern matching* for multi-track strings, that are multi-sets of strings. It can be applied to multiple sequence data such as polyphonic music data, multiple sensor data, and multiple genomes. Formally,

Table 1. Data structures for the permuted pattern matching

data structure	space	# of nodes	construction	search
GST	$O(nN)$	$2nN-1$	$O(nN)$	$O(mN+occ)$
MTST [8]	$O(nN)$	$2n-1$	$O(nN)$	$O(mN+occ)$
MTPH (proposal)	$O(nN)$	$nN+1$	$O(nN)$	$O(m^2N+occ)$
CMTPH (proposal)	$O(n)$	$n+1$	$O(nN)$	$O(m^2N^2+occ)$

two multi-sets of strings $\mathbb{T} = \{t_1, \ldots, t_N\}$ and $\mathbb{P} = \{p_1, \ldots, p_N\}$ are given, where $|t_k| = n$ and $|p_k| = m$ for $1 \leq k \leq N$, and $m \leq n$. $\mathbb{P}$ is said to permuted-match $\mathbb{T}$ at position i if there exists a permutation $(j_1, \ldots, j_N)$ of a subsequence of $(1, \ldots, N)$ such that $p_1 = t_{j_1}[i : i+m-1], \ldots, p_N = t_{j_N}[i : i+m-1]$, where $t_j[b : e]$ is the substring of t_j from b to e. Then, the permuted pattern matching problem is to find all positions i that $\mathbb{P}$ permuted-matches $\mathbb{T}$.

To solve this problem efficiently, Katsura et al. proposed an indexing structure for multi-track strings, called multi-track suffix trees (shortly MTST). MTST can be built in $O(nN)$ time and space, and it provides an $O(mN + occ)$-time matching algorithm. MTST has most $2n - 1$ nodes.

Note that another well-known indexing structure *generalized suffix tree* (GST) is also applicable to the problem. By a natural extension, the matching can be done in $O(mN + occ)$ time. The space complexity is also $O(nN)$, but the number of nodes is at most $2nN - 1$.

In this paper, we propose two new memory efficient indexing structures for multi-track strings, *multi-track position heap* (MTPH) and *contracted multi-track position heap* (CMTPH).

CMTPH is a compact version of MTPH, where some nodes are rearranged and omitted. The number of nodes in MTPH and CMTPH is at most $nN + 1$ and $n + 1$, respectively, although the input size of the multi-track text is nN.

The permuted pattern matching using MTPH or CMTPH requires $O(m^2N + occ)$ or $O(m^2N^2 + occ)$ time, respectively. Moreover, for MTPH and CMTPH, we define the MRPs to accelarate the matching.

We show $O(nN)$-time construction algorithms for MTPH and CMTPH with their MRPs. The contributions of this paper is summarized in Table 1.

2 Preliminaries

Let Σ be a finite set of characters, called an *alphabet*. We assume that Σ is fixed throughout the paper. An element of Σ^* is called a *string*. For two strings x and y, let $x \cdot y$, or xy briefly, be the *concatenation* of x and y. For a string $w = xyz$, strings x, y, z are called *prefix*, *substring*, *suffix* of w, respectively. $|w|$ is the length of w. The empty string is denoted by ε, that is $|\varepsilon| = 0$. $w[i]$ is the i-th character of w, and $w[i : j]$ is the substring of w that begins at position i and ends at j for $1 \leq i \leq j \leq |w|$. Moreover, let $w[: i] = w[1 : i]$ and $w[i :] = w[i : |w|]$. We denote by $x \prec y$ if x is lexicographically smaller than y, and denote by $x \preceq y$ if either $x \prec y$ or $x = y$. For a set S, we denote by $|S|$ the cardinality of S.

An N-tuple [1] of strings over Σ of length n is called a *multi-track string* over Σ or simply *multi-track*.

For a multi-track $\mathbb{T} = (t_1, t_2, \dots, t_N)$ over Σ, the i-th element t_i of $\mathbb{T}$ is called the *i-th track*, the length of multi-track $\mathbb{T}$ is denoted by $|\mathbb{T}|_{len} = |t_1| = |t_2| = \cdots = |t_N| = n$, and the number of tracks in multi-track $\mathbb{T}$ or the *track count* of $\mathbb{T}$, is denoted by $|\mathbb{T}|_{num} = N$. For two multi-tracks $\mathbb{X} = (x_1, x_2, \dots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \dots, y_N)$, we say that $\mathbb{X}$ *equals* $\mathbb{Y}$, denoted by $\mathbb{X} = \mathbb{Y}$, if $x_i = y_i$ for all $1 \le i \le N$.

For a multi-track $\mathbb{T} = \mathbb{XYZ}$, multi-track $\mathbb{X}$, $\mathbb{Y}$, and $\mathbb{Z}$ are called *prefix*, *substring*, and *suffix* of $\mathbb{T}$, respectively. $\mathbb{T}[i]$ denotes $(t_1[i], t_2[i], \dots, t_N[i])$ for $1 \le i \le |\mathbb{T}|_{len}$, i.e., $\mathbb{T} = \mathbb{T}[1]\mathbb{T}[2]\dots\mathbb{T}[|\mathbb{T}|_{len}]$. The substring of $\mathbb{T}$ that begins at position i and ends at position j is denoted by $\mathbb{T}[i:j] = (t_1[i:j], t_2[i:j], \dots, t_N[i:j])$ for $1 \le i \le j \le |\mathbb{T}|_{len}$. Moreover, let $\mathbb{T}[:i] = \mathbb{T}[1:i]$ and $\mathbb{T}[i:] = \mathbb{T}[i:|\mathbb{T}|_{len}]$, respectively.

Let $\mathbb{X} = (x_1, x_2, \dots, x_N)$ be a multi-track of track count N, and $\mathbf{r} = (r_1, r_2, \dots, r_N)$ be a permutation of $(1, \dots, N)$. A *permuted multi-track of* $\mathbb{X}$ *specified by* $\mathbf{r}$ is a multi-track $(x_{r_1}, x_{r_2}, \dots, x_{r_N})$, denoted by either $\mathbb{X}\langle r_1, r_2, \dots, r_N\rangle$ or $\mathbb{X}\langle\mathbf{r}\rangle$. For two multi-tracks $\mathbb{X}$ and $\mathbb{Y}$, we say that $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \overset{\bowtie}{=} \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}'$ for some permuted multi-track $\mathbb{Y}'$ of $\mathbb{Y}$. The problem we consider is defined as following:

Problem 1 (Permuted pattern matching). Given two multi-tracks $\mathbb{T} = (t_1, t_2, \dots, t_N)$ of length n and $\mathbb{P} = (p_1, p_2, \dots, p_N)$ of length m, output all positions i that satisfy $\mathbb{P} \overset{\bowtie}{=} \mathbb{T}[i : i+m-1]$.

For a multi-track $\mathbb{X} = (x_1, x_2, \dots, x_N)$, let $SI(\mathbb{X}) = (r_1, r_2, \dots, r_N)$ be a permutation such that $x_{r_i} \preceq x_{r_j}$ for any $1 \le i \le j \le |\mathbb{X}|_{num}$, and let $\Psi(\mathbb{X}) = \mathbb{X}\langle SI(\mathbb{X})\rangle$. All $SI(\mathbb{T}[i:])$ for $1 \le i \le n$ and $\Psi(\mathbb{T})$ can be computed in $O(nN)$ time using the suffix tree [16] or the suffix array [11]. It is known that the suffix tree and the suffix array can be constructed in linear time with respect to the length of the input string [6, 7, 9, 12, 13, 15]. Therefore, Problem 1 can be solved in $O(nmN)$ time and $O(nN)$ space by storing all $SI(\mathbb{T}[i:])$ naively. The aim of this paper is to solve Problem 1 more efficiently than the above naive result.

A *trie* on Σ is a rooted tree that has the following two properties: (1) each edge is labeled by a character $c \in \Sigma$, and (2) for each node u and a character $c \in \Sigma$, u has at most one edge that is labeled by c from u to a child of u. Let $T = (V, E)$ be a trie, where V and E are sets of nodes and edges, respectively. The root node of T is denoted by *root*. Each edge $e \in E$ is denoted by (u, c, v), where $c \in \Sigma$ is the label of e, and v is a child node of a node u. Note that the time required to find the child of a node on the child edge labeled by $c \in \Sigma$ is $O(\log|\Sigma|)$. Because $|\Sigma|$ is a fixed constant in this paper, so that the above time cost is also constant. For a node $v \in V$, the sequence of nodes and edges from *root* to v, that is *root*$, e_1, v_1, e_2, v_2, \dots, e_k, v$, is called the *path* from *root* to

[1] A multi-track string was regarded as a multi-set of strings in [8]. In this paper, however, we define it as a tuple of strings for notational convenience.

v, denoted by $path(root, v)$. The number of edges on $path(root, v)$ is called the *depth* of v, denoted by $depth(v)$. Let c_i be the label of e_i for $i = 1, \ldots, k$. Then, we say that the string $w = c_1c_2 \ldots c_k$ is *represented in* T, and denote the node v by $\overline{w}$ and the string w by $label(v)$.

Thus, $root = \overline{\varepsilon}$ and $label(root) = \varepsilon$. For any node v in T, the set of ancestors of v is denoted by $Anc(v)$ and the descendants of v by $Des(v)$.

A *sequence hash tree* [4] is a trie for hashing a set of strings.

Definition 1 (Sequence hash trees [4]). *Let $W = \{w_1, w_2, \ldots, w_k\}$ be an ordered set of strings, where $w_i \in \Sigma^*$. For $1 \leq i \leq k$, $SHT_i(W) = (V_i, E_i)$ is a trie recursively defined by $(V_0, E_0) = (\{root\}, \emptyset)$, and $SHT_i(W) = (V_{i-1} \cup \{\overline{q_i}\}, E_{i-1} \cup \{(\overline{q_i[: |q_i| - 1]}, c, \overline{q_i})\})$, where*

q_i is the shortest prefix of w_i satisfying $\overline{q_i} \notin V_{i-1}$, and $c = q_i[|q_i|]$. $SHT_k(W)$ is called a sequence hash tree *of W and denoted by $SHT(W)$.*

For any i, $SHT_i(W)$ is obtained by adding at most one node and one edge. Thus, $SHT(W) = SHT_k(W)$ consumes $O(k)$ space. $SHT_i(W)$ is obtained by adding a node corresponding to w_i into $SHT_{i-1}(W)$. When the node corresponding to w_i is added into $SHT_{i-1}(W)$, we say that w_i is *inserted to* $SHT_{i-1}(W)$.

Lemma 1. *Let $W = \{w_1, w_2, \ldots, w_K\}$ and $W' = \{w'_1, w'_2, \ldots, w'_k\}$ $(k \leq K)$ be ordered sets of strings such that W' is a subset of W. Then $SHT(W')$ is a subtree of $SHT(W)$ rooted by the root of $SHT(W)$.*

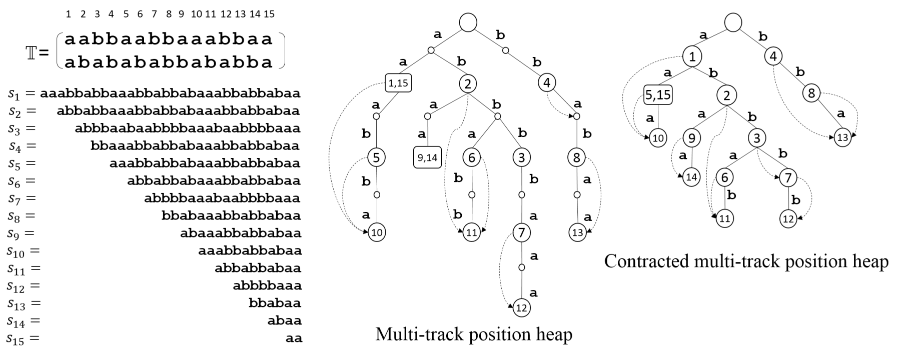
Proof. Let v be the node that is added to $SHT(W')$ when a string $w \in W'$ is inserted to $SHT(W')$, and let $d = depth(v)$. Then there exist $1 \leq i_1 < i_2 < \ldots < i_{d-1} \leq k$ such that the strings w'_{i_1}, w'_{i_2}, ..., $w'_{i_{d-1}}$ precede w in W', and $w'_{i_j}[: j] = w[: j]$ holds for each $1 \leq j \leq d-1$. These strings also precede w in W, because W' is a subset of W. Thus, w'_{i_1}, $w'_{i_2}, \ldots, w'_{i_{d-1}}$, and w are inserted to $SHT(W)$ in this order. When w'_{i_j} is inserted to $SHT(W)$, the node $\overline{w[: j]}$ is added to $SHT(W)$ if $\overline{w[: j]}$ does not exist in $SHT(W)$ for $1 \leq j \leq d-1$. Therefore, when w is inserted to $SHT(W)$, $w[: d-1]$ has already been represented in $SHT(W)$ and $\overline{w[: d]}$ is added to $SHT(W)$. As a result, any string represented in $SHT(W')$ is also represented in $SHT(W)$, so that the statement holds. □

We use the following results for the rooted tree T of n nodes and σ degree.

Lemma 2 (Lowest common ancestor query [14, 2]). *For any given two nodes u and v, the lowest common ancestor $LCA(u, v)$ of u and v in T can be answered in $O(1)$ time, after an $O(n)$ time and space preprocessing of T.*

Lemma 3 (Nearest marked ancestor query [17, 1]). *For any given node v, both marking v, denoted by $Mark(v)$, and finding the nearest ancestor $NMA(v)$ of v that is marked, can be done in $O(1)$ time, after an $O(n \log \sigma)$ time and $O(n)$ space preprocessing of T.*

Lemma 4 (Level ancestor query [3]). *For any given node v and an integer $d > 0$, the ancestor $LevA(v, d)$ of v at depth d can be answered in $O(1)$ time, after an $O(n)$ time and space preprocessing of T.*


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

$\mathbb{T}= \begin{pmatrix} \texttt{aabbaabbaaabbaa} \\ \texttt{ababababbababba} \end{pmatrix}$

$s_1 =$ aaabbabbaaabbabbabaaabbabbabaa
$s_2 =$ abbabbaaabbabbabaaabbabbabaa
$s_3 =$ abbbaabaabbbbaaabaabbbbaaa
$s_4 =$ bbaaabbabbabaaabbabbabaa
$s_5 =$ aaabbabbabaaabbabbabaa
$s_6 =$ abbabbabaaabbabbabaa
$s_7 =$ abbbbaaabaabbbbaaa
$s_8 =$ bbabaaabbabbabaa
$s_9 =$ abaaabbabbabaa
$s_{10} =$ aaabbabbabaa
$s_{11} =$ abbabbabaa
$s_{12} =$ abbbbaaa
$s_{13} =$ bbabaa
$s_{14} =$ abaa
$s_{15} =$ aa
Multi-track position heap
Contracted multi-track position heap
1,15
5,15
9,14

Fig. 1. The column concatenated strings $s_i = CS_{\Psi(\mathbb{T}[i:])}$ in the left, $MTPH(\mathbb{T})$ in the middle, and $CMTPH(\mathbb{T})$ in the right for a multi-track $\mathbb{T} =$ ($\texttt{aabbaabbaaabbaa}, \texttt{ababababbababba}$). Maximal-reach pointers $mrp(v)$ in MTPH and CMTPH are drawn as broken lines, where they are omitted if $mrp(v) = v$ for clarity. In indexing nodes, its associated positions (either one or two) are written.

3 Multi-Track Position Heaps

In this section, we propose a new data structure, named *multi-track position heap* (shortly MTPH), based on the sequence hash tree. We define a *column concatenated string* $CS_{\mathbb{T}}$ of a multi-track $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ by

$$CS_{\mathbb{T}} = t_1[1]t_2[1] \ldots t_N[1]\, t_1[2]t_2[2] \ldots t_N[2]\, \ldots t_1[n]t_2[n] \ldots t_N[n].$$

For instance, for $\mathbb{T} = (\texttt{abac}, \texttt{deba})$, we have $CS_{\mathbb{T}} = \texttt{adbeabca}$.

Definition 2 (MTPH). *Let $\mathbb{T}$ be a multi-track string of length n and track count N over Σ. Let $s_{i,j} = CS_{\Psi(\mathbb{T}[i:j])}$ for $1 \leq i \leq j \leq n$, and $s_{i,n}$ is denoted by s_i briefly. Let $S = \{s_1, s_2, \ldots, s_n\}$ be an ordered set of the strings. For $1 \leq i \leq n$, $MTPH_i(\mathbb{T}) = (V_i, E_i)$ is a trie recursively defined by $(V_0, E_0) = (\{root\}, \emptyset)$, and $V_i = V_{i-1} \cup \bigcup_{j=0}^{\Delta_i - 1} \{\overline{s_i[: |q_i| + j]}\}$, $E_i = E_{i-1} \cup \bigcup_{j=0}^{\Delta_i-1} \{(\overline{s_i[: |q_i| + j - 1]}, s_i[|q_i| + j], \overline{s_i[: |q_i| + j]})\}$, where q_i is the shortest prefix of s_i such that $\overline{q_i} \notin V_{i-1}$ and $q_i \neq \varepsilon$, and $\Delta_i = N - ((|q_i| - 1) \bmod N)$. If no such q_i exists, that is $\overline{s_i} \in V_{i-1}$, then $(V_i, E_i) = (V_{i-1}, E_{i-1})$. $MTPH_n(\mathbb{T})$ is called a* multi-track position heap *of $\mathbb{T}$, and denoted by $MTPH(\mathbb{T})$.*

Figure 1 shows an example of $MTPH(\mathbb{T})$ and column concatenated strings.

Both the numbers of nodes and edges of $MTPH_i(\mathbb{T})$ increase at most N from $MTPH_{i-1}(\mathbb{T})$ for each $1 \leq i \leq n$. Thus, $MTPH(\mathbb{T})$ consumes $O(nN)$ space. If there exists q_i, then we associate the position i to the node $\overline{s_i[: |q_i| + \Delta_i - 1]}$, and call it an *indexing node*. Otherwise, that is $\overline{s_i} \in V_{i-1}$, we associate i to the node $\overline{s_i}$. Therefore, each indexing node stores either one or two positions. In case that an indexing node v stores two positions i and j with $i < j$, we call that i is the *primary position* and j is the *secondary position* in v.

We will show that $MTPH(\mathbb{T})$ can be constructed in $O(nN)$ time by updating $MTPH(\mathbb{T}[: i-1])$ to $MTPH(\mathbb{T}[: i])$ iteratively for $i = 1, 2, \ldots, n$, similar to the online construction algorithm for position heaps [10]. We remark that it is not trivial because s_i is not necessarily a suffix of s_{i-1} (see Figure 1, left). Let us focus on the differences between $\mathit{MTPH}(\mathbb{T}[:i-1])$ and $\mathit{MTPH}(\mathbb{T}[:i])$. For $1 \leq j \leq i$, if j is a primary position in a node v in $\mathit{MTPH}(\mathbb{T}[:i-1])$, j must be the primary position stored in the same node v in $\mathit{MTPH}(\mathbb{T}[:i])$. If j is a secondary position in $\mathit{MTPH}(\mathbb{T}[:i-1])$, there are two cases in $\mathit{MTPH}(\mathbb{T}[:i])$: (1) j becomes a primary position in a newly created node v', or (2) j remains the secondary position, but in another node v'. In any case, the node v' is in $\mathit{Des}(v)$. Thus, we consider how to update the nodes storing two positions.

Let j $(1 \leq j < i)$ be any secondary position in a node v in $\mathit{MTPH}(\mathbb{T}[:i-1])$. If the string $s_{j,i}$ is not represented in $\mathit{MTPH}(\mathbb{T}[:i-1])$ yet, then we create a new path $\mathit{path}(\mathit{root}, \overline{s_{j,i}})$ and reset the position j from v to a newly created node $\overline{s_{j,i}}$. Otherwise, the position j must be a secondary in another existing node v' in $\mathit{MTPH}(\mathbb{T}[:i])$. Thus, we should reset j from v to $v' = \overline{s_{j,i}}$. These update process can be done by traversing the nodes storing the secondary positions.

We will show that for any position b $(1 \leq b < i)$, if b is a secondary position then $b+1$ is also a secondary position, by a series of lemmas as follows.

Lemma 5. *For two multi-tracks* $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ *and* $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, *if* $\Psi(\mathbb{X}) = \Psi(\mathbb{Y})$ *then* $\Psi(\mathbb{X}[2:]) = \Psi(\mathbb{Y}[2:])$.

Proof. Trivial. □

Lemma 6. *For any multi-track* $\mathbb{W}$ *of length* m, *if* $CS_{\Psi(\mathbb{W})}$ *is represented in* $\mathit{MTPH}_i(\mathbb{T})$, *then* $CS_{\Psi(\mathbb{W}[2:])}$ *is also represented in* $\mathit{MTPH}_i(\mathbb{T})$ *for any* $1 \leq i \leq n$.

Proof. Because $CS_{\Psi(\mathbb{W})}$ is represented in $\mathit{MTPH}_i(\mathbb{T})$, there are $1 \leq j_1 \leq j_2 \leq \ldots \leq j_m \leq i$ such that $\Psi(\mathbb{T}[j_k : j_k + k - 1]) = \Psi(\mathbb{W}[:k])$ for $1 \leq k \leq m$, and they have been inserted to MTPH in the order of $k = 1, 2, \ldots, m$. At the same time, $\Psi(\mathbb{T}[j_k + 1 : j_k + k - 1])$ for $1 \leq k \leq m$ have also been inserted in this order, because MTPH is constructed by inserting suffixes in descending order with respect to the length. Lemma 5 leads $\Psi(\mathbb{T}[j_k + 1 : j_k + k]) = \Psi(\mathbb{W}[2:k])$. Thus, $CS_{\Psi(\mathbb{T}[j_m+1:j_m+m])} = CS_{\Psi(\mathbb{W})[2:]}$ is represented in $\mathit{MTPH}_i(T)$. □

Lemma 7. *If* b *is a secondary position of a node* v *in* $\mathit{MTPH}_i(\mathbb{T})$ *for* $1 \leq b < i$, *then* $b+1$ *is also a secondary position of another node in it.*

Proof. Let b' be the primary position of v. Then, $b' < b$ and $s_b = s_{b'}[:|s_b|]$ hold. From Lemma 6, s_{b+1} is represented in $\mathit{MTPH}_i(\mathbb{T})$. In addition, $s_{b'+1}[:|s_{b+1}|]$ is also represented. From Lemma 5, $s_{b+1} = s_{b'+1}[:|s_{b+1}|]$ holds. Since $b' < b$, $b'+1 < b+1$. Thus, $b+1$ is a secondary position of $\overline{s_{b'+1}}$. □

Let b be the smallest position that is a secondary position in $\mathit{MTPH}(\mathbb{T}[:i-1])$ with $1 \leq b \leq i-1$. By Lemma 7, all the secondary positions are written as $b, b+1, \ldots, i-1$. In addition, these positions are partitioned into two intervals. Let b' be the smallest position such that $s_{b',i}$ is represented in $\mathit{MTPH}(\mathbb{T}[:i-1])$. Then, $s_{b'+1,i}$ is also represented in it by Lemma 6. Similarly, all $s_{b'+2,i}, \ldots, s_{i-1,i}$ are represented in it, too. Therefore, all the positions $b, b+1, \ldots, b'-1$ in the first interval are primary positions in $MTPH(\mathbb{T}[:i])$, while all $b', b'+1, \ldots, i-1$ in the second interval are secondary positions in $MTPH(\mathbb{T}[:i])$. Summarizing the above discussion, to obtain $MTPH(\mathbb{T}[:i])$, $MTPH(\mathbb{T}[:i-1])$ should be updated as follows: (1) for $b \leq j < b'$, build $path(\overline{s_{j,i-1}}, \overline{s_{j,i}})$ and reset the position j from $\overline{s_{j,i-1}}$ to the new node $\overline{s_{j,i}}$ as its primary position, and (2) for $b' \leq j \leq i$, reset the position j from $\overline{s_{j,i-1}}$ to the existing node $\overline{s_{j,i}}$ as its secondary position. We refer the position b as the *active position*, and the indexing node $\overline{s_{b,i-1}}$ as the *active node*, similarly to [**?**]. The nodes storing positions $b, b+1, \ldots, i-1$ can be traversed efficiently by using the suffix pointers defined below.

Definition 3 (Multi-track suffix pointers). *For any indexing node $\overline{s_{i,j}}$ in $MTPH(\mathbb{T})$, the* multi-track suffix pointer *of $\overline{s_{i,j}}$ is a pointer from $\overline{s_{i,j}}$ to the node $\overline{s_{i+1,j}}$, and denoted as $mtsp(\overline{s_{i,j}}) = \overline{s_{i+1,j}}$.*

For every indexing node $\overline{s_{i,j}}$ in $MTPH(\mathbb{T}[:i])$, the existence of $mtsp(\overline{s_{i,j}})$ is guaranteed by Lemma 6. In our algorithm, we will use a chain of N nodes $\perp_1, \perp_2, \ldots, \perp_N$, such that each $\perp_k$ $(1 \leq k \leq N)$ is connected to $\perp_{k+1}$ by an edge labeled by all $c \in \Sigma$, regarding that $\perp_{N+1} = root$, and $mtsp(root) = \perp_1$. They play a role of *sentinel nodes*, similarly to [15] and [**?**].

We now describe the construction algorithm of $MTPH(\mathbb{T})$. First of all, we compute $SI(\mathbb{T}[i:n])$ for all $1 \leq i \leq n$ in $O(nN)$ time. It determines every $s_i = CS_{\Psi(\mathbb{T}[i:])}$. In each iteration, we do not need to keep all the secondary nodes to update $MTPH(\mathbb{T}[:i-1])$, because these nodes can be visited through the suffix pointers recursively from the active node. Thus, we only maintain the active position b and the active node $\overline{s_{b,i-1}}$. If there is no secondary node in $MTPH(\mathbb{T}[:i-1])$, the active node is *root* and the active position is i.

In i-th iteration, the algorithm checks whether there is $path(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ or not. If it does not exist, the algorithm performs the modifications of Case (1) described above. After the modification, the new indexing node $\overline{s_{b,i}}$ is created as a descendant of $\overline{s_{b,i-1}}$. Then, the active position and the active node are updated to $b+1$ and $mtsp(\overline{s_{b,i-1}}) = \overline{s_{b+1,i-1}}$ respectively, and the algorithm performs the above process iteratively until the path is found. The multi-track suffix pointer $mtsp(\overline{s_{b,i}})$ is built as $mtsp(\overline{s_{b,i}}) = \overline{s_{b+1,i}}$ after the next modification. When $path(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ is found, the algorithm updates the active node to $\overline{s_{b,i}}$ and makes the suffix pointer from the last created indexing node to the new active node if such a node exists. Hence, for any indexing node, suffix pointer of it is defined indeed. To update $MTPH(\mathbb{T}[:i-1])$ into $MTPH(\mathbb{T}[:i])$, it is enough to perform only the modifications of Case (1), because the modifications of Case (2) does not add any node nor edge to MTPH. All the secondary positions will be determined after constructing $MTPH(\mathbb{T})$ by traversing nodes through the suffix pointers recursively from the active node.

Algorithm 1 shows a pseudo-code of the construction algorithm, and the function to find $path(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ at line 26. Let us analyze the running time of Algorithm 1. Each iteration of the **while**-loop from line 9 takes $O(nN)$ time over the whole run of the algorithm, because at most N nodes and edges are visited or created in each iteration and $1 \leq b \leq n$. Each process in the rest of

Algorithm 1: MTPH construction algorithm

```
   Input: $\mathbb{T}$
   Output: $MTPH(\mathbb{T})$
1  compute $SI(\mathbb{T}[i:n])$ for all $1 \le i \le n$;
2  create $root$ and $\bot_N$;    create edge $(\bot_N, c, root)$ for each character $c$;
3  for $j = N-1$ downto 1 do
4      create node $\bot_j$;    create edge $(\bot_j, c, \bot_{j+1})$ for each character $c$;
5  $mtsp(root) = \bot_1$;    $activeNode = root$;    $b = 1$;
6  for $i = 1$ to $n$ do
7      $lastNode$ = undefined;
8      $targetNode = find(activeNode, s_b[depth(activeNode)+1:])$;
9      while $targetNode$ = undefined do
10         for $j = 1$ to $N$ do
11             $u = activeNode$;
12             if there is not an edge labeled by $w[j]$ from $u$ then
13                 create a node $v$ and an edge $(u, w[j], v)$ where
                   $label(v) = label(u) \cdot w[i]$;
14             else
15                 Let $v$ be a child of $u$ connected by the edge $(u, w[j], v)$;
16             $u = v$;
17         if $lastNode \neq$ undefined then  $mtsp(lastNode) = u$;
18         $lastNode = u$;    Let $lastNode$ store $b$;
19         $activeNode = mtsp(activeNode)$;    $b = b + 1$;
20         $targetNode = find(activeNode, s_b[depth(activeNode)+1:])$;
21     $activeNode = targetNode$;
22     if $lastNode \neq$ undefined then  $mtsp(lastNode) = activeNode$;
23 if $b \neq n+1$ then
24     for $b \le j \le n$ do
25         Let $activeNode$ store $j$ ;    $activeNode = mtsp(activeNode)$;

26 Function $find(node, w)$
27     for $j = 1$ to $N$ do
28         if exist edge $(node, w[j], v)$ then $node = v$;
29         else return undefined;
30     return $node$;
```

the **for**-loop from line 6 takes at most $O(N)$ time, and the loop iterates exactly n times. Thus, the running time of Algorithm 1 is $O(nN)$ time.

Theorem 1. *Algorithm 1 constructs $MTPH(\mathbb{T})$ in $O(nN)$ time and space.*

We now consider to solve Problem 1 for a pattern $\mathbb{P}$ by using $MTPH(\mathbb{T})$. Let w be the longest prefix of $CS_{\Psi(\mathbb{P})}$ represented in $MTPH(\mathbb{T})$. We can compute w in $O(mN)$ time by traversing $path(root, \overline{w})$. If $w = CS_{\Psi(\mathbb{P})}$, all the positions stored in the nodes of the subtree rooted by $\overline{w}$ are the occurrences of the pattern $\mathbb{P}$ in $\mathbb{T}$. We will deal with enumerating all these positions later. Before it, remark

Algorithm 2: Adding maximal reach pointers for MTPH

```
Input: $\mathbb{T}$and $MTPH(\mathbb{T})$ with multi-track suffix pointers
1 $currNode = root$;   $pointerNode = root$;   $\ell = 1$;
2 for $i = 1$ to $n$ do
3     while $currNode$ has an outgoing edge labeled by $\overline{s_i[\ell]}$ and $\ell < |s_i|$ do
4         $currNode = \overline{s_i[:\ell]}$;
5         if $currNode$ is an indexing node then $pointerNode = currNode$;
6         $\ell = \ell + 1$;
7     $mrp(\alpha_i) = currNode$;   $currNode = mtsp(pointerNode)$;
8     $pointerNode = currNode$;   $\ell = depth(currNode)$;
```

that we should also consider another type of occurrences; let I be the set of positions stored in the nodes on $path(root, \overline{w})$. These are also candidates for the occurrences. Because $path(root, \overline{w})$ contains at most m indexing nodes, $|I| = O(m)$. For each $i \in I$, we check whether $\mathbb{P} \between \mathbb{T}[i : i+m-1]$ or not by simply comparing $CS_{\Psi(\mathbb{P})}$ with $s_{i,i+m-1} = CS_{\Psi(\mathbb{T}[i:i+m-1])}$, and report it if it does. Both the computation of $s_{i,i+m-1}$ and the comparison are done in $O(mN)$ time.

We now explain how to enumerate all the positions in the nodes of the subtree rooted by $\overline{w}$, in case that $w = CS_{\Psi(\mathbb{P})}$. Let α_i be an indexing node that stores the position i in $MTPH(\mathbb{T})$. To obtain these positions efficiently in $O(occ)$ time, we construct another tree T consists only of indexing nodes α_i's in $MTPH(\mathbb{T})$, where a node α_i is a child of another node α_j in T if and only if $\alpha_i \in Des(\alpha_j)$ and no other indexing node exists between α_i and α_j in $MTPH(\mathbb{T})$. Obviously, T can be built by depth-first-traversal of $MTPH(\mathbb{T})$ in $O(nN)$ time, and by traversing the subtree of T rooted by the node $\overline{w}$, we can enumerate all matched positions in $O(occ)$ time. Thus, we can determine all the positions i such that $\mathbb{P} \between \mathbb{T}[i : i+m-1]$ in $O(m^2N + occ)$ time.

We now show that the matching by using MTPH can be accelerated by adding *maximal-reach pointers* (shortly MRPs). MRPs are the auxiliary structures for standard position heaps proposed by Ehrenfeucht et al. [5]. We will extend it to MTPHs as follows.

Definition 4 (MRPs for MTPHs). *For an indexing node α_i storing i in $MTPH(\mathbb{T})$, the* maximal-reach pointer *of α_i is a pointer from α_i to $\overline{s_i[:\ell_i]}$, and denoted by $mrp(\alpha_i) = \overline{s_i[:\ell_i]}$, where $s_i[:\ell_i]$ is the longest prefix of $s_i = CS_{\Psi(\mathbb{T}[i:])}$ represented in $MTPH(\mathbb{T})$.*

Algorithm 2 is an algorithm for adding MRPs to MTPH, that is based on Kucherov's algorithm for standard position heaps [**?**]. First of all, the algorithm preprocesses $MTPH(\mathbb{T})$ so that for any node v, it can obtain the depth of v in $O(1)$ time, by assigning unique numbers to the nodes by the depth first traversal. In i-th iteration between line 2 and line 8, it adds a pointer $mrp(\alpha_i) = \overline{s_i[:\ell_i]}$, where $\overline{s_i[:\ell_i]}$ is determined as follows: beginning by $currNode = root$, it goes down to a child $\overline{s_i[:\ell]}$ of $depth(currNode) \leq \ell \leq |s_i|$ until either $currNode$ does not have a child $\overline{s_i[:\ell]}$ or $\ell = |s_i|$ holds (line 3 to line 6). Then, $mrp(\alpha_i)$ is obtained as $\overline{s_i[:\ell_i]}$ (line 7). The next $i+1$-th iteration begins at $mtsp(pointerNode)$, where *pointerNode* is the deepest indexing node visited in the i-th iteration and computed in line 5. Note that, the process of line 5 can be computed in constant time because whether *currNode* is an indexing node or not is determined by $depth(currNode) \bmod N = 0$ or not.

Let us consider the time complexity of Algorithm 2. Each process of line 1, line 7 and line 8 can be done in $O(1)$ time, so that these processes take $O(n)$ time in total. Let us consider the number of executions of **while**-loop at line 3. In each loop, $s_i[\ell]$ corresponding to a letter in the text $\mathbb{T}$ is read. We assume $s_i[\ell_i]$ belongs to k-th column of $\mathbb{T}$ for $1 \le k \le n$. k does not decrease between i-th and $(i+1)$-th iterations because $(i+1)$-th iteration begins at $mtsp(\overline{s_{i,k}}) = \overline{s_{i+1,k}}$. In addition, since $|s_i[\ell_i]| \ge N$, $i \le k$ holds. Thus, all letters in $\mathbb{T}$ are read at least one time in all iterations. On the other hand, the letters corresponding to the labels on $path(pointerNode, currNode)$ in i-th iteration can be read redundantly in $(i+1)$-th iteration. However, the number of such labels does not exceed N in each iteration. Therefore, the total number of executions of **while**-loop does not exceed $2nN$. As a result, the running time of Algorithm 2 is $O(nN)$ time.

In the naive matching algorithm with MTPH described above, the cost of the comparison of $CS_{\Psi(\mathbb{P})}$ with $s_{i,i+m-1}$ for $i \in I$ can be reduced from $O(mN)$ to $O(1)$ by using the MRPs and the lowest common ancestor queries mentioned in Lemma 4, if $CS_{\Psi(\mathbb{P})}$ itself is represented in $MTPH(\mathbb{T})$. Whether $CS_{\Psi(\mathbb{P})} = s_{i,i+m-1}$ or not is determined by $mrp(\alpha_i) \in Des(\overline{CS_{\Psi(\mathbb{P})}})$. If $LCA(mrp(\alpha_i), \overline{CS_{\Psi(\mathbb{P})}}) = \overline{CS_{\Psi(\mathbb{P})}}$, then $mrp(\alpha_i) \in Des(\overline{CS_{\Psi(\mathbb{P})}})$ holds. By Lemma 4, the query $LCA(mrp(\alpha_i), \overline{CS_{\Psi(\mathbb{P})}})$ can be answered in $O(1)$ time after an $O(nN)$ time and space preprocessing of $MTPH(\mathbb{T})$. Thus, the comparison of $CS_{\Psi(\mathbb{P})}$ with $s_{i,i+m-1}$ can be done in $O(1)$ time. Hence, the total time is $O(mN + occ)$ in this case, different from $O(m^2N + occ)$ time of the naive algorithm. Remark that, unfortunately, this result does not improve the upper-bound of the time complexity of the matching for the worst case. If $CS_{\Psi(\mathbb{P})}$ is not represented in $MTPH(\mathbb{T})$, we must compute $CS_{\Psi(\mathbb{P})} = s_{i,i+m-1}$, that takes $O(mN)$ time. In this case, all comparisons are done in $O(m^2N)$ time because $|I| < m$. By considering the above two cases, the time bound of the matching is $O(mN + occ + m^2N) = O(m^2N + occ)$.

Theorem 2. *Problem 1 can be solved in $O(m^2N+occ)$ time by using $MTPH(\mathbb{T})$ with MRPs.*

4 Contracted Multi-Track Position Heaps

We propose a more space-efficient version of MTPH, by omitting non-indexing nodes of MTPH (see Figure 1, right).

Definition 5 (CMPTH). *Let $\mathbb{T}$ be a multi-track string of length n and track count N over Σ. Let $S = \{s_1, s_2, \ldots, s_n\}$ be an ordered set of strings, where $s_i = CS_{\Psi(\mathbb{T}[i:])}$ for $1 \le i \le n$. A* contracted multi-track position heap *of $\mathbb{T}$, denoted by $CMTPH(\mathbb{T})$, is a sequence hash tree of S, i.e., $CMTPH(\mathbb{T}) = SHT(S)$.*

Since $\mathit{CMTPH}(\mathbb{T})$ is a sequence hash tree for an ordered set of cardinality n, it has $n+1$ nodes and n edges, so that $\mathit{CMTPH}(\mathbb{T})$ consumes only $O(n)$ space. Given $\mathbb{T}$, we can construct easily $\mathit{CMTPH}(\mathbb{T})$ in $O(nN+n^2)$ time as follows: compute $\mathit{SI}(\mathbb{T}[i:])$ for $1 \leq i \leq n$ in $O(nN)$ time, then insert all $s_i = CS_{\Psi(\mathbb{T}[i:])}$ to the tree in order; each insertion can be done in $O(n)$, so that $O(n^2)$ in total.

We now show a more efficient construction algorithm for $\mathit{CMTPH}(\mathbb{T})$, that runs in $O(nN)$ time. It re-assign the positions in the nodes in $\mathit{MTPH}(\mathbb{T})$, and eliminates all non-indexing nodes as follows. First let us noticed that in Figure 1, $\mathit{CMTPH}(\mathbb{T})$ is a subtree of $\mathit{MTPH}(\mathbb{T})$ with the same root node, if we ignore the positions stored in the nodes. It is always the cases, as follows.

Lemma 8. *For $1 \leq i \leq n$, $\mathit{CMTPH}_i(\mathbb{T})$ is a subtree of $\mathit{MTPH}_i(\mathbb{T})$ rooted by the root of $\mathit{MTPH}_i(\mathbb{T})$, if we ignore the positions stored in the nodes.*

Proof. $\mathit{CMTPH}_i(\mathbb{T})$ is a sequence hash tree of $S = \{s_1, s_2, \ldots, s_i\}$. On the other hand, $\mathit{MTPH}_i(\mathbb{T})$ is equivalent to a sequence hash tree of $S' = \bigcup_{k=1}^{i} \bigcup_{j=1}^{\Delta_k} \{s_k\} = \{\underbrace{s_1, \ldots, s_1}_{\Delta_1}, \underbrace{s_2, \ldots, s_2}_{\Delta_2}, \ldots, \underbrace{s_i, \ldots, s_i}_{\Delta_i}\}$. Because S is a subset of S', the statement holds by Lemma 1. □

Lemma 8 implies that all nodes and edges in $\mathit{CMTPH}(\mathbb{T})$ are included in $\mathit{MTPH}(\mathbb{T})$. Therefore, $\mathit{CMTPH}(\mathbb{T})$ can be obtained by the following process. For each $i = 1, 2, \ldots, n$, we re-assign the position i stored in an indexing node α_i to its ancestor node $\beta_i \in \mathit{Anc}(\alpha_i)$, that does not store the primary position (i.e., β_i may store the secondary position) and the farthest from α_i (i.e., nearest from the *root*). If there is no such a node, then α_i keeps storing i. After that, we eliminate all nodes that stores no position. Then, the remaining tree is $\mathit{CMTPH}(\mathbb{T})$.

To find β_i efficiently, we use the two types of queries on a rooted tree, that are the nearest marked ancestor query and the level ancestor query referred in Lemma 3 and Lemma 4, respectively. Each query can be answered in constant time after a linear-time preprocess of the tree.

We now show how to find β_i from α_i. We mark a node to indicate that the node stores some positions in $\mathit{CMTPH}(\mathbb{T})$. At the beginning, only the *root* of $\mathit{MTPH}(\mathbb{T})$ is marked. Because β_i is the farthest unmarked ancestor of α_i, it is the depth $d+1$ ancestor of α_i, where d is the depth of the nearest marked ancestor u of α_i. The ancestor u is obtained by $\mathit{NMA}(\alpha_i)$, and then β_i by $\mathit{LevA}(u, \mathit{depth}(u) + \mathit{1})$, both in $O(1)$ time. If $u \neq \alpha_i$, re-assign the position i from α_i to β_i, and mark β_i. Otherwise, i.e., $u = \alpha_i$, do nothing. Repeating it for $i = 1, 2, \ldots, n$, we get $\mathit{CMTPH}(\mathbb{T})$ in $O(n)$ time from $\mathit{MTPH}(\mathbb{T})$. Because $\mathit{MTPH}(\mathbb{T})$ can be constructed in $O(nN)$ time, we obtain the following result.

Theorem 3. *Given a multi-track $\mathbb{T}$ of length n and track count N, $\mathit{CMTPH}(\mathbb{T})$ can be constructed in $O(nN)$ time and space.*

CMTPHs is useful for permuted pattern matching instead of MTPHs. Because any node in CMTPH stores at least one position, the candidate positions for matching is at most $|I| = O(mN)$. Thus, the time complexity is

$O(m^2N^2 + occ)$. It can be improved by using the maximal-reach pointers for CMTPH, denoted by $cmrp(\beta_i)$, in the same way as MTPH. Because $cmrp(\beta_i) = NMA(mrp(\alpha_i))$ holds for any i after marking all β_i's, we get them in $O(n)$ time. Thus we have the following results.

Theorem 4. *Given a multi-track $\mathbb{T}$ of length n and track count N, CMTPH($\mathbb{T}$) with the maximal-reach pointers for CMTPH($\mathbb{T}$) can be constructed in $O(nN)$ time and space.*

For the permuted pattern matching, the maximal-reach pointers of CMTPH work similarly to that of MTPH. Thus, it is not difficult to see that the following theorem holds.

Theorem 5. *Problem 1 can be solved in $O(m^2N^2 + occ)$ time by using CMTPH($\mathbb{T}$) with the maximal-reach pointers.*

5 Conclusion and Future Work

We proposed two new indexing structures, MTPH and CMTPH, for multi-track strings, that are memory-efficient compared with the multi-track suffix tree in [8]; MTPH and CMTPH need $O(nN)$ and $O(n)$ space, respectively. We showed an $O(nN)$-time construction algorithms of MTPH and CMTPH, and proposed MRPs for both of them. By using these data structures, the permuted pattern matching problem can be solved efficiently: $O(m^2N + occ)$ time by MTPH, and $O(m^2N^2 + occ)$ time by CMTPH. Our future work is to construct CMPTH directly in $O(nN)$ time without constructing MTPH. We are also preparing experiments to evaluate these structures.

References

1. Amir, A., Farach, M., Idury, R.M., Poutre, J.A.L., Schäffer, A.A.: Improved dynamic dictionary matching. Information and Computation 119(2), 258–282 (1995)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: LATIN 2000: Theoretical Informatics, 88–94 (2000)
3. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplied. Theoretical Computer Science 321(1), 5–12 (2004)
4. Coffman, Jr., E.G., Eve, J.: File structures using hashing functions. Communications of the ACM 13(7), 427–432 (1970)
5. 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)
6. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS. 137–143 (1997)
7. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: ICALP. 943–955 (2003)
8. Katsura, T., Narisawa, K., Shinohara, A., Bannai, H., Inenaga, S.: Permuted pattern matching on multi-track strings. In: SOFSEM, 280–291 (2013)

9. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: CPM, 200–210 (2003)
10. Kucherov, G.: On-line construction of position heaps. Journal of Discrete Algorithms 20, 3–11 (2013)
11. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
12. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2), 262–272 (1976)
13. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: DCC, 193–202 (2009)
14. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
15. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
16. Weiner, P.: Linear pattern matching algorithms. In: SWAT, 1–11 (1973)
17. Westbrook, J.: Fast incremental planarity testing. In: ICALP, 342–353 (1992)