Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings

Diptarama, Ryo Yoshinaka, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University
6-6-05 Aramaki Aza Aoba, Aoba-ku, Sendai, Japan
{diptarama@shino., ry@, ayumi@}ecei.tohoku.ac.jp

Abstract. A multi-track string is a tuple of strings of the same length. The full permuted pattern matching problem is, given two multi-track strings $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ and $\mathbb{P} = (p_1, p_2, \ldots, p_N)$ such that $|p_1| = \cdots = |p_N| \leq |t_1| = \cdots = |t_N|$, to find all positions i such that $\mathbb{P} = (t_{r_1}[i : i+m-1], \ldots, t_{r_N}[i : i+m-1])$ for some permutation $(r_1, \ldots, r_N)$ of $(1, \ldots, N)$, where $m = |p_1|$ and $t[i : j]$ denotes the substring of t from position i to j. We propose new algorithms that perform full permuted pattern matching practically fast. The first and second algorithms are based on the Boyer-Moore algorithm and the Horspool algorithm, respectively. The third algorithm is based on the Aho-Corasick algorithm where we use a multi-track character instead of a single character in the so-called *goto* function. The fourth algorithm is an improvement of the multi-track Knuth-Morris-Pratt algorithm that uses an automaton instead of the failure function of the original algorithm. Our experiment results demonstrate that those algorithms perform permuted pattern matching faster than existing algorithms.

Keywords: permuted pattern matching, multi-track string, Boyer-Moore algorithm, Horspool algorithm, AC-automaton

1 Introduction

The pattern matching problem on strings is to find all occurrences of a pattern string in a text string. Pattern matching algorithms such as the Knuth-Morris-Pratt (KMP) algorithm [8], Boyer-Moore algorithm [2], and Horspool algorithm [5], perform pattern matching fast by preprocessing the pattern. On the other hand, pattern matching can be also performed by preprocessing the text into some data structure such as a suffix tree [12], a suffix array [10], and a position heap [4].

The *permuted pattern matching problem*, proposed by Katsura *et al.* [6,7], is a generalization of the pattern matching problem, where we compare tuples of strings. Tuples of strings can model various types of real data such as multiple-sensor data, polyphonic music data, and multiple genomes. We call a tuple of strings of the same length a *multi-track string*. The permuted pattern matching problem is, given two multi-track strings $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ and $\mathbb{P} = (p_1, p_2, \ldots, p_M)$ where $M \leq N$ and $|t_1| = \cdots = |t_N| = n \geq |p_1| = \cdots = |p_M| = m$, to find all positions i such that $\mathbb{P}$ is a permutation of a sub-tuple of $(t_1[i : i + m - 1], \ldots, t_N[i : i + m - 1])$, where $w[i : j]$ denotes the substring of w from position i to j. As an example, data from multiple sensors such as a three dimensional accelerometer can be considered as a multi-track string. This problem can be solved by constructing some data structure from the text such as a multi-track suffix tree [6] and a multi-track position heap [7], or by preprocessing the pattern like the Aho-Corasick (AC) automaton based algorithm [6] and KMP based algorithm [3] do.

In this paper, we focus on the *full* permuted matching problem, which is a special case of the permuted matching problem where we have $M = N$. We propose several

Diptarama, Ryo Yoshinaka, Ayumi Shinohara: Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings, pp. 7–21.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

Algorithm	Preprocessing time	Matching time	Online
AC-automaton based [6]	$O(mM \log \sigma)$	$O(nN \log \sigma)$	yes
Multi-track KMP* [3]	$O(mM)$	$O(nN)$	no
Filter-MTKMP [3]	$O(m(M + \sigma))$	$O(n(mN + \sigma))$	yes
MT-BM*	$O(m(M \log \sigma + \sigma))$	$O(nN(m + \log \sigma) + n(N + \sigma))$	yes
MT-H*	$O(m(M \log \sigma + \sigma))$	$O(nN(m + \log \sigma) + n(N + \sigma))$	yes
MT AC-automaton*	$O(dM \log \sigma)$	$O(nN \log \sigma)$	no
MT permuted matching automaton*	$O(mM \log \sigma)$	$O(nN \log \sigma)$	yes

Table 1. Comparison of the algorithms for permuted pattern matching. Multi-track AC-automaton can find occurrences of *multiple* patterns. Algorithms with an asterisk are for full permuted pattern matching ($M = N$).

new algorithms that perform full permuted pattern matching practically fast. The first algorithm, *MT-BM*, is based on the Boyer-Moore algorithm [2] and the second one, *MT-H*, is based on the Horspool algorithm [5], on which we made a significant improvement by using a data structure called *track trie*. The third algorithm, *multi-track AC-automaton*, is an algorithm for dictionary matching on multi-track strings based on the AC-algorithm [1], where we use a multi-track character instead of a single character in the so-called *goto* function. The fourth algorithm, *multi-track permuted matching automaton*, is an improvement of multi-track KMP algorithm [3] that uses an automaton instead of the failure function in the KMP algorithm. Moreover, we conduct experiments and show that our algorithms perform permuted pattern matching faster than existing algorithms. The worst case running time of proposed algorithms and existing algorithms are summarized in Table 1, where d is the total length of the patterns and σ is the size of the alphabet.

2 Preliminaries

Let $w \in \Sigma^n$ be a string of length n over an alphabet Σ and $\sigma = |\Sigma|$ be the alphabet size. The length n of w is denoted by $|w|$. The *empty string*, denoted by ε, is a string of length 0. By $w[i]$ we denote the i-th character of w for $i \in \{1, \ldots, n\}$. The substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. We abbreviate $w[1 : i]$ to $w[: i]$ and $w[i : n]$ to $w[i :]$, which are called a *prefix* and a *suffix* of w, respectively. The reverse string of w is denoted by w^R: that is, $w^R = w[n]w[n-1] \ldots w[2]w[1]$. For two strings x and y, we denote by $x \prec y$ that x is lexicographically smaller than y, and by $x \preceq y$ that either $x = y$ 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 $\mathbb{W}$. A *multi-track character* $\mathbb{C} = (c_1, c_2, ..., c_N)$ is an N-tuple of characters $c_i \in \Sigma$. The length n of strings in $\mathbb{W}$ is called the *length* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{len}$. The number N of tracks in $\mathbb{W}$ is called the *track count* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{num}$. The multi-track character $(w_1[i], w_2[i], ...w_N[i])$ is denoted by $\mathbb{W}[i]$ and the multi-track $\mathbb{W}[i : j]$ is $(w_1[i : j], w_2[i : j], \ldots, w_N[i : j])$ for $1 \leq i \leq j \leq |\mathbb{W}|_{len}$. Similarly to the notation for strings, $\mathbb{W}[: i]$ and $\mathbb{W}[i :]$ mean $\mathbb{W}[1 : i]$ and $\mathbb{W}[i : |\mathbb{W}|_{len}]$ and called a prefix and a suffix of $\mathbb{W}$, respectively. Moreover, $\mathbb{W}[i][j]$ denotes $w_j[i]$.

Let $\mathbf{r} = (r_1, r_2, \ldots, r_N)$ be a permutation of $(1, 2, \ldots, N)$. For a multi-track $\mathbb{W} = (w_1, w_2, \ldots, w_N)$, $\mathbb{W}\langle \mathbf{r} \rangle = \mathbb{W}\langle r_1, r_2, \ldots, r_N \rangle = (w_{r_1}, \ldots, w_{r_N})$ is called a *permuted multi-track* of $\mathbb{W}$. The *sorted index* $SI(\mathbb{W})$ of a multi-track $\mathbb{W}$ is a permutation

$(r_1, \ldots, r_N)$ such that $w_{r_i} \preceq w_{r_{i+1}}$ for any $1 \leq i < N$, where we assume $r_i < r_{i+1}$ in the case $w_{r_i} = w_{r_{i+1}}$. The sorted multi-track $sort(\mathbb{W})$ is defined as $\mathbb{W}\langle SI(\mathbb{W})\rangle$. The *reverse* of a multi-track $\mathbb{W} = (w_1, \ldots, w_N)$ is $\mathbb{W}^R = (w_1^R, \ldots, w_N^R)$. The sorted index of the reverse multi-track, denoted by $RI(\mathbb{W})$, is a permutation $(r_1, \ldots, r_N)$ such that $w_{r_i}^R \preceq w_{r_{i+1}}^R$ for any $1 \leq i < N$. Note that $SI(\mathbb{W}[i:])$ and $RI(\mathbb{W}[:i])$ for $1 \leq i \leq n$ can be computed in $O(nN \log \sigma)$ time *offline* by using a suffix tree [6,11] or a suffix array [9], and $RI(\mathbb{W}[:i])$ for $1 \leq i \leq n$ can be computed in $O(n(N + \sigma))$ time *online* by using radix sort.

For two multi-tracks $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \bowtie \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}\langle \mathbf{r} \rangle$ for some permutation $\mathbf{r}$.

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 $|\mathbb{T}|_{num} = N = M$ and $|\mathbb{T}|_{len} = n \geq m$. The pattern matching problem on multi-tracks is defined as follows.

Definition 1 (Full permuted pattern matching). *Given a multi-track text* $\mathbb{T}$ *and a multi-track pattern* $\mathbb{P}$*, compute all positions* i *that satisfy* $\mathbb{P} \bowtie \mathbb{T}[i : i + m - 1]$.

For example, given a text $\mathbb{T} = \begin{pmatrix} \texttt{aabaaaaa}, \\ \texttt{abaabbaa}, \\ \texttt{baaababa} \end{pmatrix}$ and a pattern $\mathbb{P} = \begin{pmatrix} \texttt{aba}, \\ \texttt{baa}, \\ \texttt{aaa} \end{pmatrix}$, we can see that the pattern matches at $\mathbb{T}[2:4] = \mathbb{P}$. Moreover, the pattern permuted matches with $T[6:8]$, since $\mathbb{P}\langle 3, 2, 1\rangle = \begin{pmatrix} \texttt{aaa}, \\ \texttt{baa}, \\ \texttt{aba} \end{pmatrix} = T[6:8]$. Therefore, we should output $\{2, 6\}$ in this case.

We remark that Katsura *et al.* defined a more general problem, where we have $|\mathbb{T}|_{num} = N \geq |\mathbb{P}|_{num} = M$ and our task is to find a subsequence $(r_1, \ldots, r_M)$ of $(1, \ldots, N)$ and a position i for which $\mathbb{P} \bowtie \mathbb{T}\langle t_{r_1}, \ldots, t_{r_M}\rangle[i : i + m - 1]$ holds.

3 Boyer-Moore and Horspool algorithms for multi-track strings

In this section, we propose two permuted pattern matching algorithms that are based on the Boyer-Moore algorithm and the Horspool algorithm, which we call MT-BM and MT-H, respectively.

3.1 Multi-track Boyer-Moore algorithm

The original Boyer-Moore algorithm uses two failure functions GS (good suffixes) and BC (bad characters) to determine how much the position of a substring to compare should be shifted when a mismatch is found between the input patten and the substring of the text. Those functions are defined as follows on multi-tracks.

Definition 2 (Suffixes). *For a multi-track* $\mathbb{P}$ *of length* $|\mathbb{P}|_{len} = m$*,* $suf[i]$ *is the maximum value of* l *such that* $\mathbb{P}[i - l + 1 : i] \bowtie \mathbb{P}[m - l + 1 : m]$ *for* $1 \leq i \leq m$.

Definition 3 (Good suffixes). *For multi-track* $\mathbb{P}$ *of length* $|\mathbb{P}|_{len} = m$*,* $GS[m] = 1$ *and* $GS[i] = \min A$ *for* $0 \leq i < m$*, where*

$$A = \left\{ 0 < s < i \;\middle|\; \mathbb{P}[i - s + 1 : m - s] \bowtie \mathbb{P}[i + 1 : m], \mathbb{P}[i - s : m - s] \not\bowtie \mathbb{P}[i : m] \right\} \cup \left\{ i \leq s < m \;\middle|\; \mathbb{P}[1 : m - s] \bowtie \mathbb{P}[s + 1 : m] \right\} \cup \{m\}.$$

Algorithm 1: MT-BM and MT-H preprocessing functions

Function *ComputeSuf*($\mathbb{P}$)
  compute $RI(\mathbb{P}[:i])$ for $1 \leq i \leq m$;
  $suf[m] \leftarrow m$, $j \leftarrow m$, $k \leftarrow m$;
  for $i \leftarrow m-1$ **to** 1 **do**
    if $i > k$ **and** $suf[m-(j-i)] < i-k$ **then** $suf[i] \leftarrow suf[m-(j-i)]$;
    else
      if $i < k$ **then** $k \leftarrow i$;
      $j \leftarrow i$;
      while $k > 0$ **and** $\mathbb{P}[k]\langle RI(\mathbb{P}[:j])\rangle = \mathbb{P}[k+m-j]\langle RI(\mathbb{P}[:m])\rangle$ **do** $k \leftarrow k-1$;
      $suf[i] \leftarrow j-k$;
  return suf;
Function *ComputeGS*($\mathbb{P}$)
  $suf \leftarrow$ ComputeSuf($\mathbb{P}$);
  for $i \leftarrow 1$ **to** m **do** $GS[i] \leftarrow m$;
  $j \leftarrow 1$;
  for $i \leftarrow m$ **to** 1 **do**
    if $suf[i] = i$ **then**
      while $j \leq m-i$ **do**
        if $GS[j] = m$ **then** $GS[j] \leftarrow m-i$;
        $j \leftarrow j+1$;
  for $i \leftarrow 1$ **to** $m-1$ **do** $GS[m-suf[i]] \leftarrow m-i$;
  return GS;
Function *ComputeBC*($\mathbb{P}$)
  compute $RI(\mathbb{P}[:i])$ for $1 \leq i \leq m$;
  for $i \leftarrow 1$ **to** $m-1$ **do**
    if $BC(\mathbb{P}[i]\langle RI(\mathbb{P}[:i])\rangle) = m$ **then** $BC.add(\mathbb{P}[i]\langle RI(\mathbb{P}[:i])\rangle, m-i)$;
    else $BC(\mathbb{P}[i]\langle RI(\mathbb{P}[:i])\rangle) \leftarrow m-i$;
  return BC;

Algorithm 2: MT-BM

Input: Multi-track $\mathbb{T}$, Multi-track $\mathbb{P}$
Output: match positions
compute $RI(\mathbb{T}[:i])$ for $1 \leq i \leq n$;
compute $RI(\mathbb{P}[:i])$ for $1 \leq i \leq m$;
$BC \leftarrow$ ComputeBC($\mathbb{P}$);
$GS \leftarrow$ ComputeGS($\mathbb{P}$);
$j \leftarrow 0$;
while $j \leq n-m+1$ **do**
  $i \leftarrow m$;
  while $i > 0$ **and** $\mathbb{T}[i+j]\langle RI(\mathbb{T}[:j+m])\rangle = \mathbb{P}[i]\langle RI(\mathbb{P}[:m])\rangle$ **do** $i \leftarrow i-1$;
  if $i \leq 0$ **then**
    output $j+1$;
    $j \leftarrow j+GS[0]$;
  else $j \leftarrow j+\max(GS[i], BC(\mathbb{T}[i+j]\langle RI(\mathbb{T}[:i+j])\rangle)-(m-i))$;

Definition 4 (Bad character). *For multi-track* $\mathbb{P}$ *of length* $|\mathbb{P}|_{len} = m$ *and a multi-track character* $\mathbb{C}$, $BC(\mathbb{C})$ *is the first occurrence position of* $sort(\mathbb{C})$ *in* $\mathbb{P}^R[2:]$. *The function* $BC(\mathbb{C})$ *returns* m *if there is no occurrence of* $sort(\mathbb{C})$ *in* $\mathbb{P}^R[2:]$.

Algorithm 3: MT-H

Input: Multi-track $\mathbb{T}$, Multi-track $\mathbb{P}$
Output: match positions

1 compute $RI(\mathbb{T}[:i])$ for $1 \leq i \leq n$;
2 compute $RI(\mathbb{P}[:i])$ for $1 \leq i \leq m$;
3 $BC \leftarrow \mathsf{ComputeBC}(\mathbb{P})$;
4 $j \leftarrow 0$;
5 **while** $j \leq n - m + 1$ **do**
6 $\quad i \leftarrow m$;
7 $\quad$ **while** $i > 0$ **and** $\mathbb{T}[i+j]\langle RI(\mathbb{T}[:j+m])\rangle = \mathbb{P}[i]\langle RI(\mathbb{P}[:m])\rangle$ **do** $i \leftarrow i - 1$;
8 $\quad$ **if** $i \leq 0$ **then output** $j + 1$;
9 $\quad$ **else** $j \leftarrow j + BC(\mathbb{T}[j+m]\langle RI(\mathbb{T}[:j+m])\rangle)$) ;

In the implementation, *suf* and GS can be represented as arrays, while BC can be realized in a trie of the multi-track characters. We perform permuted-match instead of exact match when computing GS. Algorithm 1 shows how to construct GS and BC. The array GS is computed by ComputeGS, which uses array *suf* computed by ComputeSuf. Note that we compute RI at the beginning (Lines 2 and 24) of the algorithm and will not recompute them when we use the values later.

Lemma 5. *The function ComputeSuf computes the array suf in $O(m(M+\sigma))$ time.*

Proof. First, $RI(\mathbb{P}[:i])$ can be computed in $O(m(M+\sigma))$ time by using radix sort. The **for** loop is executed $m-1$ times and the **while** loop at line 9 is executed at most m times through the whole run, because k is always reduced in each loop. Comparison of two multi-track characters of the pattern that executed in each loop can be computed in $O(M)$ time. □

Lemma 6. *The function ComputeGS computes GS in $O(m)$ time.*

Proof. All the **for** loops are executed at most m times. The **while** loop is executed at most m times through the whole execution of the algorithm, since j is always increased and does not exceed m. □

Lemma 7. *The function ComputeBC computes BC in $O(m(M \log \sigma + \sigma))$ time.*

Proof. $RI(\mathbb{P}[:i])$ can be computed in $O(m(M+\sigma))$ time by using radix sort. Each edge in the trie of BC can be accessed in $O(\log \sigma)$ time by using binary search. Since the depth of the trie is at most M, each $BC(\mathbb{P}[i])$ for $1 \leq i \leq m$ can be added and accessed in $O(M \log \sigma)$ time. □

By using both GS and BC, MT-BM outputs the positions of the text that are permuted-matched with the pattern. The matching algorithm of MT-BM is shown in Algorithm 2.

Theorem 8. *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-BM outputs the positions of the text that permuted-match with the pattern online in $O(nN(m + \log \sigma) + n(N+\sigma))$ time in the worst case with $O(m(M \log \sigma + \sigma))$ time preprocessing.*

Proof. From Lemmas 5, 6, and 7, Algorithm 2 needs $O(m(M \log \sigma + \sigma))$ time for preprocessing. Next, $RI(\mathbb{T}[:i])$ can be computed in $O(n(N+\sigma))$ time by using radix sort. In the outer **while** loop starting at line 6, the value of j is increased by at least 1,

Algorithm 4: Track-trie construction algorithm (constructTrackTrie($\mathbb{P}$))

```
Input: Multi-track ℙ
Output: trackTrie
1  newNode ← rootNode;
2  weight(rootNode) ← M;
3  for i ← 1 to M do
4      activeNode ← rootNode;
5      for j ← m to 1 do
6          if goto(activeNode, ℙ[j][i]) = Null then
7              newNode ← newNode + 1;
8              weight(newNode) ← 1;
9              goto(activeNode, ℙ[j][i]) ← newNode;
10             activeNode ← newNode;
11         else
12             activeNode ← goto(activeNode, ℙ[j][i]);
13             weight(activeNode) ← weight(activeNode) + 1;
```

Algorithm 5: Track-trie matching algorithm (matchTrackTrie($\mathbb{T}, j$))

```
Input: Multi-track 𝕋, index j, trackTrie
Output: mismatch position
1  activeNode[k] ← rootNode for 1 ≤ k ≤ M;
2  temp(node) ← 0 for all node in trackTrie;
3  for i ← m to 1 do
4      for k ← 1 to M do
5          if goto(activeNodes, 𝕋[i + j][k]) = Null then return i ;
6          else
7              activeNodes[k] ← goto(activeNodes[k], 𝕋[i + j][k]);
8              temp(activeNodes[k]) ← temp(activeNodes[k]) + 1;
9              if temp(activeNodes[k]) > weight(activeNodes[k]) then return i ;
10 return 0;
```

so the loop is executed at most $n - m + 2$ times. In each execution of the outer loop, the inner **while** loop is executed at most m times, where multi-track characters of the pattern and the text can be compared in $O(N)$ time. BC can be accessed in $O(N \log \sigma)$ time and GS can be executed in $O(1)$ time. □

3.2 Multi-track Horspool algorithm

MT-H in Algorithm 3 uses BC to shift the pattern that can be computed in the same way as BC of MT-BM shown in Algorithm 1.

Theorem 9. *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-H outputs the positions of the text that are permuted-matched with the pattern in $O(nN(m+\log \sigma)+n(N+\sigma))$ time in the worst case with $O(m(M \log \sigma + \sigma))$ time preprocessing.*

Proof. Similar to the proof of Theorem 8, beside MT-H uses BC only. □

3.3 Boyer-Moore and Horspool matching algorithms with track-trie

The two algorithms presented in the previous subsections decide if two multi-tracks permuted-match by sorting them. In this subsection, we present another idea for this

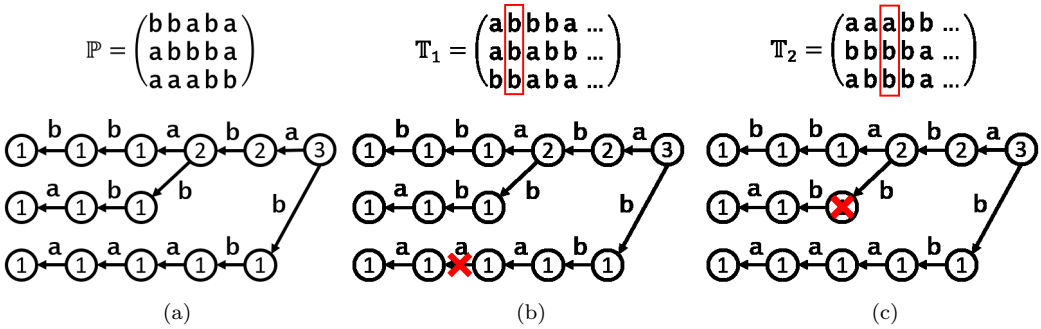

Figure 1. (a) Track trie of $\mathbb{P} = (\texttt{bbaba}, \texttt{abbba}, \texttt{aaabb})$, (b) example of mismatch when the track trie cannot find the transition, (c) example of mismatch when the number of tracks is more than the weight of the node.

task using a data structure called a *track trie*. The track trie of a multi-track $\mathbb{P}$ stores all the reversed strings of the tracks of $\mathbb{P}$, that is, $\{p_1^R, p_2^R, \ldots, p_M^R\}$. Fig. 1(a) shows the track trie of a multi-track pattern $\mathbb{P} = (\texttt{aaabb}, \texttt{abbba}, \texttt{bbaba})$.

Algorithm 4 is the construction algorithm for the track-trie of $\mathbb{P}$. For a node s of the track trie and a character $c \in \Sigma$, the goto function $goto(s, c)$ returns the child of s that has an edge labeled c. We naturally extend it to the domain Σ^* by $goto(s, \varepsilon) = s$ and $goto(s, aw) = goto(goto(s, a), w)$ for any $a \in \Sigma$ and $w \in \Sigma^*$. We also associate a weight with each node to find mismatch on a text, as we will explain later.

Theorem 10. *Algorithm 4 constructs the track-trie of $\mathbb{P}$ in $O(mM \log \sigma)$ time.*

Proof. The function *goto* can be calculated in $O(\log \sigma)$ time by binary search. On each execution of the inner **for** loop (line 5), Algorithm 4 executes *goto* to check child nodes of *activeNode*. If there is no node with an edge labeled $\mathbb{P}[j][i]$, then a new node is constructed, which can be done in $O(1)$ time. On the other hand, if there is a node with an edge labeled $\mathbb{P}[j][i]$, Algorithm 4 accesses the child node and then increases its weight by one. The total number of iterations of the inner loop is mM. □

For a given multi-track text $\mathbb{T}$ and a position i, Algorithm 5 finds a mismatch position in two cases; (1) when a track cannot find its *goto* destination, and (2) when the number of tracks that have the same string w is more than the weight of the node that represents the string $goto(root, w)$. Those mismatch conditions are illustrated in Fig. 1 (b) and (c), respectively. Fig. 1 (b) shows that the track trie cannot find a transition for the second character `b` of the third track. On the other hand, Fig. 1 (c) shows that $\mathbb{T}_2[3:]$ has two '`bba`' on its track, however the $\mathbb{P}[3:]$ has only one '`bba`' on its track, i.e. the node that represents '`bba`' has one on its weight.

Theorem 11. *Given a multi-track text $\mathbb{T}$ and a position j, Algorithm 5 finds a mismatch position in the pattern in $O(mM \log \sigma)$ time.*

Proof. For each position $i+j$ on the text, Algorithm 5 executes *goto* to check whether $activeNodes[k]$ has a child node with an edge labeled $\mathbb{T}[i+j][k]$ for $1 \leq k \leq M$. If there is no child node with an edge labeled $\mathbb{T}[i+j][k]$, then Algorithm 5 considers it as mismatch and returns the mismatch position. On the other hand, if there is such a child node, Algorithm 5 changes $activeNodes[k]$ to the child node, and then check whether the number of tracks of $\mathbb{T}[i+j:]$ that contain $\mathbb{T}[k][i+j:i+m]$ as a prefix is more than the weight of the child node. If the number of tracks exceeds the weight, then Algorithm 5 treats it as mismatch and returns the mismatch position. The total number of iterations of the inner loop is at most mM. □

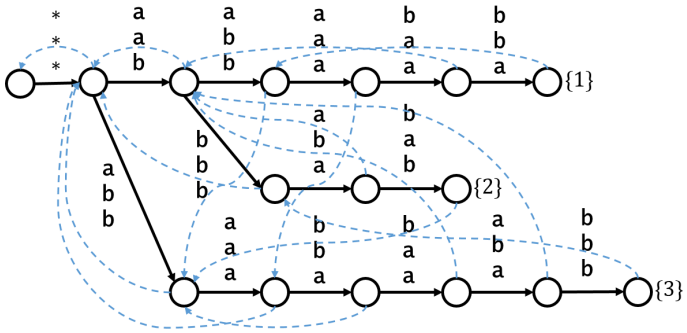

Figure 2. Multi-track AC-automaton of $D = \{\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3\}$, where $\mathbb{P}_1 = (\texttt{aaabb}, \texttt{abaab}, \texttt{bbaaa})$, $\mathbb{P}_2 = (\texttt{abab}, \texttt{abba}, \texttt{bbab})$, and $\mathbb{P}_3 = (\texttt{aabbab}, \texttt{bababb}, \texttt{baaaab})$. The asterisk '*' is a special character that matches with any characters in Σ.

Algorithm 6: Multi-track AC-automaton goto function and initial output function construction algorithm

Input: Set of multi-track patterns $D = \{\mathbb{P}_1, \mathbb{P}_2, ..., \mathbb{P}_r\}$
Output: Goto function and initial output function

```
1  compute SI(P_i[j :]) for 1 ≤ i ≤ r and 1 ≤ j ≤ m_i;
2  create states rootState and ⊥;
3  goto(⊥, W) ← rootState for all multi-track character W ∈ Σ^M;
4  newState ← rootState;
5  for i ← 1 to r do
6      activeState ← rootState;
7      for 1 ≤ j ≤ m_i do
8          if goto(activeState, P_i[j]⟨SI(P_i[1 :])⟩) ≠ fail then
9              activeState ← goto(activeState, P_i[j]⟨SI(P_i[1 :])⟩);
10         else
11             newState ← newState + 1;
12             goto(activeState, P_i[j]⟨SI(P_i[1 :])⟩) ← newState;
13             label(newState) ← i;
14             activeState ← newState;
15         if k = m_i then
16             output(activeState) ← output(activeState) ∪ {P_i};
```

Although the worst case time complexity remains the same, by using track-trie, both MT-BM and MT-H can match the pattern to the text practically faster, because we do not need to compute the reverse sorted index of the text. First, we construct the track-trie of the pattern by using constructTrackTrie($\mathbb{P}$). Then, we replace line 8 (resp. line 7) of Algorithm 2 (resp. Algorithm 3) by matchTrackTrie($\mathbb{T}, j$) to find a mismatch position.

4 Multi-track AC-automaton

In this section, we will explain a data structure called a multi-track AC-automaton that can perform dictionary matching on multi-tracks. Given a set $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ of multi-track patterns called a *dictionary* and a multi-track text $\mathbb{T}$, by preprocessing the patterns, the multi-track AC-automaton can find all occurrence positions of each pattern in the text. Let $d = \Sigma_{i=1}^{r} m_i$ be the total length of the patterns in D, where

Algorithm 7: Multi-track AC-automaton failure function and output function construction algorithm.

Input: Goto function and initial output function
Output: Failure and output functions

compute $SI(\mathbb{P}_i[j:])$ for $1 \leq i \leq r$ and $1 \leq j \leq m_i$;
$failure(rootState) \leftarrow \perp$;
push $rootState$ to $queue$;
while $q \neq empty$ **do**
    pop $activeState$ from $queue$;
    for a *such that* $goto(activeState, a) = s \neq$ **fail do**
        push s to $queue$;
        $state \leftarrow failure(activeState)$;
        $j \leftarrow label(s)$;
        while $goto(state, \mathbb{P}_j[depth(s)]\langle SI(\mathbb{P}_i[depth(s) - depth(state):])\rangle) =$ **fail do**
            $state \leftarrow failure(state)$;
        $failure(s) \leftarrow goto(state, \mathbb{P}_j[depth(s)]\langle SI(\mathbb{P}_i[depth(s) - depth(state):])\rangle)$;
        $output(s) \leftarrow output(s) \cup output(failure(s))$;

Algorithm 8: multi-track AC-automaton matching algorithm

Input: Goto, failure and output functions
Output: Set of (pattern,position) tuple $\{(\mathbb{P}_{k_1}, pos_1), (\mathbb{P}_{k_2}, pos_2), ...\}$

compute $SI(\mathbb{T}[i:])$ for $1 \leq i \leq n$;
$activeState \leftarrow rootState$;
for $i = 1$ **to** n **do**
    while $goto(activeState, \mathbb{T}[i]\langle SI(\mathbb{T}[i - depth(activeState) + 1:])\rangle) =$ **fail do**
        $activeState \leftarrow failure(activeState)$;
    $activeState \leftarrow goto(activeState, \mathbb{T}[i]\langle SI(\mathbb{T}[i - depth(activeState) + 1:])\rangle)$;
    for $k \in output[activeState]$ **do output** $(k, i - m_k + 1)$;

$m_i = |\mathbb{P}_i|_{len}$. The multi-track AC-automaton of D, denoted by $MTAC(D)$, consists of three functions; *goto*, *failure*, and *output* functions.

Unlike the original AC-automaton, the multi-track AC-automaton uses a multi-track character, instead of a single character to define *goto*. The states and *goto* in $MTAC(D)$ construct a trie of $sort(\mathbb{P}_i)$ for all $\mathbb{P}_i \in D$. Each state in $MTAC(D)$ represents a prefix of $sort(\mathbb{P}_i)$, thus each state can be denoted by $S(\mathbb{W})$, where $\mathbb{W}$ is the string obtained by concatenating the labels of the edges from the root to the state. Therefore, we can define $goto(S(\mathbb{P}_i[:j]), \mathbb{P}_i[j+1]) = S(\mathbb{P}_i[:j+1])$ for $1 \leq i \leq r$ and $1 \leq j < m_i$. For convenience, we denote $goto(goto(s, \mathbb{P}_i[i]), \mathbb{P}_i[i+1])$ as $goto(s, \mathbb{P}_i[i:i+1])$, and $goto(goto(s, \mathbb{P}_i[j:k-1]), \mathbb{P}_i[k])$ as $goto(s, \mathbb{P}_i[j:k])$. For a state s and a multi-track character $\mathbb{C}$, $goto(s, \mathbb{C})$ can be implemented by using multi-track character trie of depth at most M nodes, thus $goto(s, \mathbb{C})$ can be executed in $O(M \log \sigma)$ time. The function *goto* can be constructed by using Algorithm 6.

Next, the failure function of a state $S(\mathbb{P}_i[:j])$ is defined as $\mathit{flink}(S(\mathbb{P}_i[:j])) = S(sort(\mathbb{P}_i[k:j]))$, where $\mathbb{P}_i[k:j]$ is the longest proper suffix of $\mathbb{P}_i[:j]$ such that $\mathbb{P}_i[k:j]$ is a prefix of some $sort(\mathbb{P}_\ell)$ with $\mathbb{P}_\ell \in D$. Algorithm 7 shows a construction algorithm for the failure function of a multi-track AC-automaton.

Finally, the output function of the multi-track AC-automaton is similar to the original AC-Automaton. For a state $S(\mathbb{P}_i[:j])$, the output of the state

$output(S(\mathbb{P}_i[:\, j]))$ is the set of patterns $\mathbb{P}_\ell \in D$ such that $\mathbb{P}_\ell \cong \mathbb{P}_i[k : j]$ for some $1 \le k \le j$. The initial output function is constructed by Algorithm 6, and then updated by Algorithm 7 to get the final output function. Fig. 2 shows an example of $MTAC(D)$. In order to simplify the construction algorithm, we use a special state that reads any multi-track character to get to the root state.

Theorem 12. *Algorithm 6 constructs the goto and initial output functions of a set $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ of multi-track patterns in $O(dM \log \sigma)$ time.*

Proof. The total number of executions of *goto* is $d = \Sigma_{i=1}^{r} m_i$ and $goto(s, \mathbb{C})$ is executed in $O(M \log \sigma)$ time. □

Theorem 13. *Algorithm 7 constructs the failure and output functions of a set $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ of multi-track patterns in $O(dM \log \sigma)$ time.*

Proof. Updating the output function can be performed in $O(1)$ time by using list to save the output function, and update it by concatenate the list. Therefore, we can bound the running time of Algorithm 7 by counting the number of executions of *goto*. For each pattern $\mathbb{P}_i$, let $s_{i,j}$ be a state such that $s_{i,j} = goto(root, \mathbb{P}_i[:\, j])$ for $1 \le j \le m_i$. Let $f_{i,j}$ be the number of executions of *failure* when finding $failure(s_{i,j})$. The maximum value of $f_{i,j}$ is bounded by $depth(failure(s_{i,j-1})) + 1$. Because the depth of $failure(s_{i,j})$ is at most $depth(failure(s_{i,j-1})) - f_{i,j} + 1$, we get $f_{i,j} \le depth(failure(s_{i,j-2})) - f_{i,j-1} + 2$ recursively. By solving this formula, we get $\Sigma_{j=1}^{m_i} f_{i,j} \le 2m_i$, and $\Sigma_{i=1}^{r} \Sigma_{j=1}^{m_i} f_{i,j} \le \Sigma_{i=1}^{r} 2m_i = 2d$. Moreover, each *goto* is executed in $O(M \log \sigma)$ time. □

By using the goto, output, and failure functions, the multi-track AC-automaton can perform permuted pattern matching on a text $\mathbb{T}$ as shown in Algorithm 8. Let *activeState* be the current state of the multi-track AC-automaton and d be the depth of *activeState*. For each position i on $\mathbb{T}$, Algorithm 8 uses the sorted index of $\mathbb{T}[i-d\,:]$ to determine permutation of $\mathbb{T}[i]$ used in the goto function.

Theorem 14. *Algorithm 8 performs permuted pattern matching on a multi-track text $\mathbb{T}$ in $O(nN \log \sigma)$ time.*

Proof. The running time of Algorithm 8 can be evaluated by counting the number of executions of *goto*. First, for each i, *goto* is executed at least once on *activeState* transition. Next, *goto* is executed to check whether the transition is **fail** or not. In this case, the number of executions of *goto* is the same as that of *failure*. The latter is at most n, because whenever *goto* is executed, the depth of *activeState* is increased by one, and whenever *failure* is executed, the depth of *activeState* is decreased by at least one. Therefore, the number of executions of *goto* is $O(n)$. □

5 Multi-track permuted matching automaton

In this section, we will describe a data structure called a multi-track permuted matching automaton that can perform permuted pattern matching on a multi-track text $\mathbb{T}$ online, by preprocessing a multi-track pattern $\mathbb{P}$. A multi-track permuted matching automaton is constructed by two functions, *goto* and *failure*. In addition, similarly to a track-trie, each state of the multi-track permuted matching automaton has a

Algorithm 9: Multi-track permuted matching automaton goto function construction algorithm

Input: Multi-track $\mathbb{P}$
Output: Goto function

```
1  create states rootState and ⊥;
2  goto(⊥, w) ← rootState for all character w ∈ Σ;
3  newState ← rootState;
4  weight(⊥) ← weight(rootState) ← M;
5  for i ← 1 to M do
6      activeState ← rootState;
7      for 1 ≤ j ≤ m do
8          if goto(activeNode, ℙ[j][i]) = Null then
9              newState ← newState + 1;
10             weight(newState) ← 1;
11             activeState ← newState;
12         else
13             activeState ← goto(activeState, ℙ[j][i]);
14             weight(activeState) ← weight(activeState) + 1;
15         if k = m then set activeState as an accept state ;
```

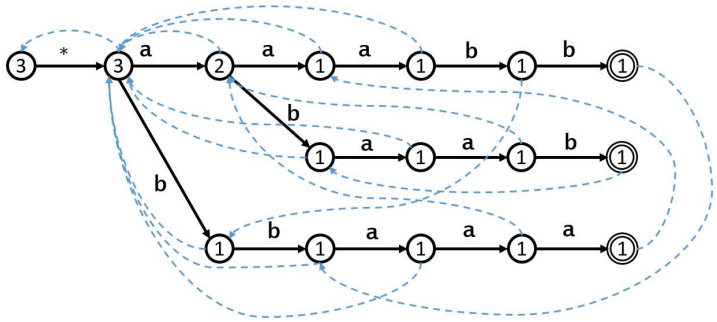

Figure 3. Multi-track permuted matching automaton of $\mathbb{P} = (\texttt{aaabb}, \texttt{abaab}, \texttt{bbaaa})$. The asterisk '*' is a special character that matches with any characters in Σ.

weight in order to determine whether *failure* should be executed or not. Fig. 3 shows an example of a multi-track permuted matching automaton.

For a multi-track pattern $\mathbb{P} = (p_1, p_2, ..., p_m)$, the multi-track permuted matching automaton of the pattern is denoted by $MTPMA(\mathbb{P})$. The goto function of the multi-track permuted matching automaton is similar to that of an AC-automaton, thus, each state in $MTPMA(\mathbb{P})$ represents a prefix of p_i, which is denoted by $S(w)$, where w is the string obtained by concatenating the labels of the edges from the root to the state. Each state $S(w)$ has a weight, which is a number of tracks containing w as a prefix. Moreover, a state $S(w)$ is called an *accept state* if $w = p_i$ for some i. Algorithm 9 constructs the goto function of a multi-track permuted matching automaton.

Theorem 15. *Algorithm 9 constructs the goto function of a multi-track pattern* $\mathbb{P}$ *in* $O(mM \log \sigma)$ *time.*

Proof. For each track, the number of executions of *goto* is m and there are M tracks in a pattern $\mathbb{P}$. Moreover, *goto* can be executed in $O(\log \sigma)$ time. □

Next, we will define the failure function of a multi-track permuted matching automaton. Let S_j be the set of states that have depth j and $S(p_i[:j]) \in S_j$ be a state

Algorithm 10: Multi-track permuted matching automaton failure function construction algorithm

```
   Input: Multi-track ℙ, goto function
   Output: Failure function
 1 activeStates[i] ← rootState for 1 ≤ i ≤ M;
 2 failure(rootState) ← ⊥;
 3 for i ← 1 to m do
 4     tempStates ← activeStates;
 5     failFlag ← true;
 6     while failFlag = true do
 7         failFlag ← false;
 8         vector failStates;
 9         for j ← 1 to |tempStates| do failStates[j] ← failure(tempStates[j]) ;
10         failFlag ← isFail(activeStates, failStates);
11         if failFlag = true then tempStates ← failStates ;
12         else
13             for j ← 1 to |activeStates| do
14                 for a such that goto(activeStates[j], a) = s ≠ fail do
15                     failure(s) ← goto(failStates[j], a);
16     clear tempStates;
17     for j ← 1 to |activeStates| do
18         for a such that goto(activeStates[j], a) = s ≠ fail do tempStates.add(s) ;
19     activeStates ← tempStates;
20 Function isFail(activeStates, failStates)
21     for j ← 1 to |activeStates| do
22         for a such that goto(activeStates[j], a) = s ≠ fail do
23             if goto(failStates[j], a) = fail then return true ;
24             else
25                 nextState ← goto(failStates[j], a);
26                 temp(nextState) ← temp(nextState) + weight(s);
27                 if temp(nextState) > weight(nextState) then return true ;
28     return false;
```

of depth j. The failure function of the state is $failure(S(p_i[:j])) = S(p_k[:\ell]) \in S_\ell$ such that $p_k[:\ell]$ is a proper suffix of $p_i[:j]$ and $\mathbb{P}[:\ell]$ permuted matches with a suffix of $\mathbb{P}[:j]$. Note that the definition of this failure function is similar to that of the multi-track KMP algorithm introduced in [3].

Algorithm 10 constructs the failure function of a multi-track permuted matching automaton. We use a state pointer for each track in the pattern. Similarly to a track-trie, there are two conditions that are considered as failure in a multi-track permuted matching automaton. The first condition is when it cannot find the goto transition, and the second condition is when the number of state pointers in the state is more than the weight of the state.

Theorem 16. *Algorithm 10 constructs the failure function of a multi-track permuted matching automaton in $O(mM \log \sigma)$ time.*

Algorithm 11: Multi-track permuted matching automaton matching algorithm

```
   Input: goto and failure functions
   Output: Permuted match positions
1  activeStates[i] ← rootState for 1 ≤ i ≤ N;
2  for 1 ≤ i ≤ n do
3      failFlag ← true;
4      while failFlag = true do
5          failFlag ← false;
6          failFlag ← isFail(activeStates, 𝕋, i);
7          if failFlag = true then
8              for j = 1 to N do activeStates[j] ← failure(activeStates) ;
9          else
10             for j = 1 to |activeStates| do activeStates[j] ← goto(activeStates[j], 𝕋[i][j]) ;
11     if activeStates[1] is an accept state then output i − m + 1 ;
12 Function isFail(activeStates, 𝕋, i)
13     for j = 1 to N do
14         if goto(activeStates[j], 𝕋[i][j]) = fail then return true ;
15         else
16             nextState = goto(activeStates[j], 𝕋[i][j]);
17             temp(nextState) = temp(nextState) + 1;
18             if temp(nextState) > weight(nextState) then return true ;
19     return false;
```

Proof. Similar to the proof of Theorem 13, the failure and goto functions are executed $O(mM)$ times. Moreover, execution time of the failure function is $O(1)$ and that of the goto function is $O(\log \sigma)$. □

Finally, by using the goto and failure functions, Algorithm 11 can perform permuted pattern match on a multi-track text $\mathbb{T}$. Algorithm 11 uses N pointers *activeStates* to point the current states. Note that all *activeStates* always have the same depth. Similarly to Algorithm 10, Algorithm 11 also uses two conditions to determine whether it should execute the failure function or not. If any of the *activeStates* is fail, then all of the *activeStates* execute the failure function, otherwise *activeStates* execute the goto function.

Theorem 17. *Algorithm 11 performs permuted pattern match on a multi-track string* $\mathbb{T}$ *in* $O(nN \log \sigma)$ *time.*

Proof. Similarly to the proof of Theorem 14, the number of executions of the failure and goto function is $O(nN)$. Since the execution time of the failure function is $O(1)$ and the goto function is $O(\log \sigma)$, Algorithm 11 runs in $O(nN \log \sigma)$ time. □

6 Experiments

We evaluate performance of our algorithms by conducting experiments on full-permuted pattern matching. We compared the running time of our algorithms with existing algorithms, AC automaton based algorithm [6] and KMP based algorithm [3]. We ran the algorithms on a computer with Intel Xeon CPU E5-2609 8 cores 2.40 GHz, 256 GB memory, and Debian Wheezy operating system.

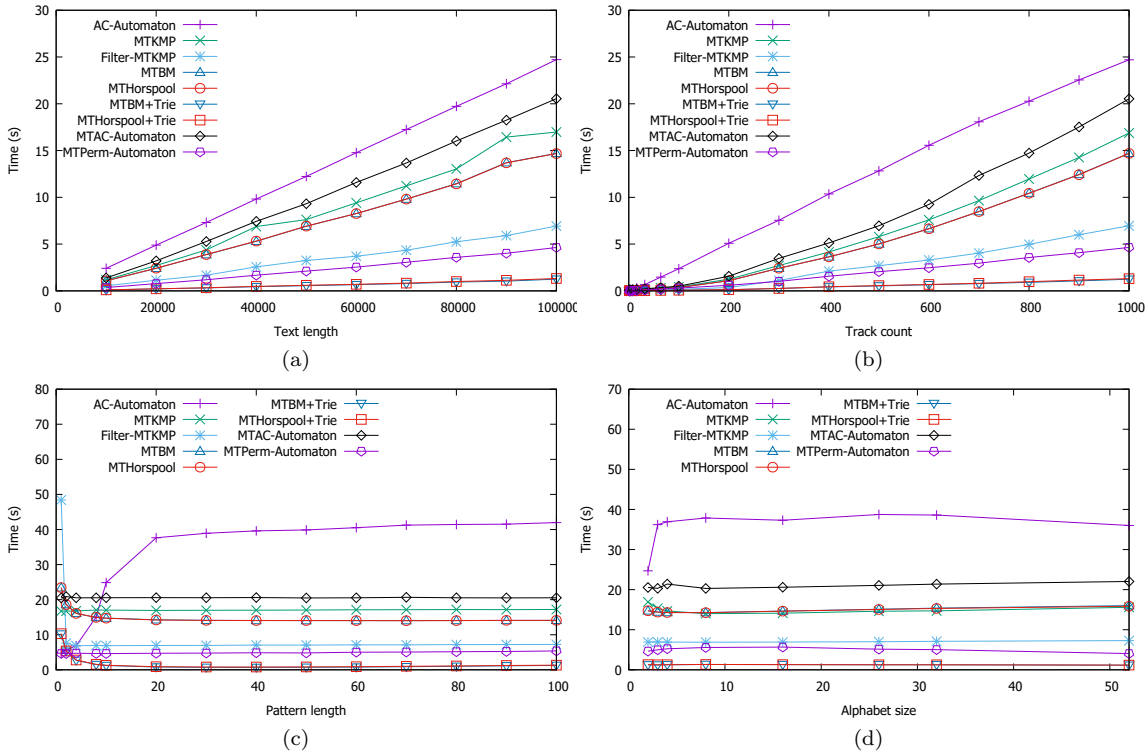

Figure 4. Running time of the algorithms on full-permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

We set the parameter values as follows, $n = 100000$, $m = 10$, $N = M = 1000$, and $\sigma = 2$, and changed one of the parameters in each experiment to see the running time of the algorithms with respect to the parameters. We used randomly generated texts and patterns, and inserted 50 occurrences of a pattern into each text to make sure that there are occurrences of the pattern in the text.

The result of the experiments are shown in Fig. 4 (a)–(d), where one of the parameters n, N, m, and σ is changed respectively. First, we can see that the running time of the algorithms increase linearly with respect to the length and track count of the text, and is not much affected by the pattern length or the alphabet size. The running times of MT-BM and MT-H are almost the same, and the running times of these algorithms are faster when a track-trie is used.

Multi-track AC-automaton is slower than the MTKMP algorithm on a single pattern matching, although it can support dictionary matching on multi-track strings. We can also see that multi-track permuted matching automaton runs faster than the MTKMP and Filter-MTKMP algorithms, as it is an improvement of MTKMP algorithm.

7 Concluding remarks

In this paper, we focused on *full* permuted pattern matching problems, where the track count N of a text equals to the track count M of a pattern. In general, the permuted pattern matching problem is more difficult if $N > M$. For example, when we construct $GS[i]$ for the full-permuted pattern matching problem, we compute the minimum value of s such that $\mathbb{P}[i-s+1:m-s] \cong \mathbb{P}[i+1:m]$, because we know that if a substring $\mathbb{T}[j:j+m-i-1]$ of the text does not match with $\mathbb{P}[i+1:m]$,

then $\mathbb{T}[j : j + m - i - 1] \ncong \mathbb{P}[i - k + 1 : m - k]$ for $0 < k < s$. However, in the case where $N > M$, there is a possibility that $\mathbb{P}[i - k + 1 : m - k]$ matches with the $\mathbb{T}[j : j + m - i - 1]$, and we might miss the occurrences of the pattern if we use the same shift as in the case of full permuted pattern matching. This problem is also arises in multi-track AC-automaton and multi-track permuted matching automaton when we try to construct the failure function. We should find another condition to define the failure function for these algorithms.

Acknowledgments This work is supported by Tohoku University Division for Interdisciplinary Advance Research and Education, JSPS KAKENHI Grant Numbers JP15H05706, JP24106010, and ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan).

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
3. Diptarama, Y. Ueki, K. Narisawa, and A. Shinohara: *KMP based pattern matching algorithms for multi-track strings*, in Proceedings of Student Research Forum Papers and Posters at SOFSEM 2016, 2016, pp. 100–107.
4. A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo: *Position heaps: A simple and dynamic text indexing data structure.* Journal of Discrete Algorithms, 9(1) 2011, pp. 100–121.
5. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
6. T. Katsura, K. Narisawa, A. Shinohara, H. Bannai, and S. Inenaga: *Permuted pattern matching on multi-track strings*, in SOFSEM, 2013, pp. 280–291.
7. T. Katsura, Y. Otomo, K. Narisawa, and A. Shinohara: *Position heaps for permuted pattern matching on multi-track strings*, in Proceedings of Student Research Forum Papers and Posters at SOFSEM 2015, 2015, pp. 41–531.
8. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
9. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in CPM, 2003, pp. 200–210.
10. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
11. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
12. P. Weiner: *Linear pattern matching algorithms*, in SWAT, 1973, pp. 1–11.