algorithms

Article

Permuted Pattern Matching Algorithms on Multi-Track Strings

Diptarama Hendrian *, **Yohei Ueki**, **Kazuyuki Narisawa**, **Ryo Yoshinaka** and **Ayumi Shinohara**

Graduate School of Information Sciences, Tohoku University, Miyagi 980-8579, Japan; yohei_ueki@shino.ecei.tohoku.ac.jp (Y.U.); narisawa@ecei.tohoku.ac.jp (K.N.); ryoshinaka@tohoku.ac.jp (R.Y.); ayumis@tohoku.ac.jp (A.S.)

* Correspondence: diptarama@tohoku.ac.jp

Received: 28 February 2019; Accepted: 3 April 2019; Published: 8 April 2019

Abstract: A multi-track string is a tuple of strings of the same length. Given the pattern and text of two multi-track strings, the permuted pattern matching problem is to find the occurrence positions of all permutations of the pattern in the text. In this paper, we propose several algorithms for permuted pattern matching. Our first algorithm, which is based on the Knuth–Morris–Pratt (KMP) algorithm, has a fast theoretical computing time with $O(mk)$ as the preprocessing time and $O(nk \log \sigma)$ as the matching time, where n, m, k, σ, and occ denote the length of the text, the length of the pattern, the number of strings in the multi-track, the alphabet size, and the number of occurrences of the pattern, respectively. We then improve the KMP-based algorithm by using an automaton, which has a better experimental running time. The next proposed algorithms are based on the Boyer–Moore algorithm and the Horspool algorithm that try to perform pattern matching. These algorithms are the fastest experimental algorithms. Furthermore, we propose an extension of the AC-automaton algorithm that can solve dictionary matching on multi-tracks, which is a task to find multiple multi-track patterns in a multi-track text. Finally, we propose filtering algorithms that can perform permuted pattern matching quickly in practice.

Keywords: multi-track string; permuted pattern matching; AC-automaton

1. Introduction

The pattern matching problem on strings is to find all occurrence positions of a pattern string in a text string. Pattern matching algorithms such as the Knuth–Morris–Pratt (KMP) algorithm [1], the Boyer–Moore algorithm [2], and the Horspool algorithm [3] perform pattern matching quickly by preprocessing the pattern. On the other hand, pattern matching can be accelerated by preprocessing the text into some indexing structure such as suffix trees [4], suffix arrays [5], or position heaps [6].

The *permuted pattern matching problem*, proposed by Katsura et al. [7,8], is a generalization of the pattern matching problem, where we compare tuples of strings. We call a tuple of strings of the same length a *multi-track string*. The permuted pattern matching problem is as follows: given two multi-track strings $\mathbb{T} = (T_1, T_2, \ldots, T_k)$ and $\mathbb{P} = (P_1, P_2, \ldots, P_k)$ where $|T_1| = \cdots = |T_k| = n \geq |P_1| = \cdots = |P_k| = m$, to find all positions i such that $(T_1[i : i+m-1], \ldots, T_k[i : i+m-1])$ is a permutation of $\mathbb{P}$. The problem can be solved by constructing indexing structures from the text such as multi-track suffix trees [7], multi-track position heaps [8], and filtering multi-set trees [9], or by preprocessing the pattern such as the AC-automaton-based algorithm [7].

Multi-track strings can model various types of real data when sequences in the data have the same meaning. For example, data from a three-dimensional accelerometer in a smartphone can be considered a multi-track string. Consider two smartphones with the same position, and rotate one of them several times where each rotation is 90 degrees on the x, y, or z axis. If we perform the same motion on both smartphones, the generated data from three-dimensional accelerometer of one phone is a permutation of the other one. By using permuted pattern matching, we can find occurrences of a pattern on both smartphones without permuting the pattern.

In this paper, we propose several algorithms that solve permuted pattern matching quickly. These algorithms can be classified into three groups. The first group consists of algorithms that are based on the KMP algorithm [1]. This group includes the *multi-track KMP*, *multi-track AC-automaton*, and *multi-track permuted matching automaton* algorithms. The second group consists of algorithms that are based on the Boyer–Moore algorithm [2] and the Horspool algorithm [3]. This group includes the *multi-track Boyer–Moore* and *multi-track Horspool* algorithms. The third group consists of filtering algorithms. Moreover, we conduct experiments and show that our algorithms perform permuted pattern matching faster than existing algorithms. The worst case running times of the 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.

Table 1. Comparison of the algorithms for permuted pattern matching. Multi-track AC-automaton can find occurrences of *multiple* patterns.

Algorithm	Preprocessing Time	Matching Time
AC-automaton-based [7]	$O(mk\log\sigma)$	$O(nk\log\sigma)$
Multi-track KMP	$O(mk)$	$O(nk)$
MTAC-automaton	$O(dk\log\sigma)$	$O(nk\log\sigma)$
MT permuted matching automaton	$O(mk\log\sigma)$	$O(nk\log\sigma)$
MT Boyer–Moore	$O(m(k\log\sigma+\sigma))$	$O(nk(m+\log\sigma)+n(k+\sigma))$
MT Horspool	$O(m(k\log\sigma+\sigma))$	$O(nk(m+\log\sigma)+n(k+\sigma))$

Preliminary versions of this work appeared in [10,11].

2. Notation and Definition on Multi-Track Strings

Let Σ be a *totally-ordered alphabet* and $\sigma = |\Sigma|$ be the size of the alphabet. An element of Σ^* is called a *string*. For a string $W \in \Sigma^*$, the length of W is denoted by $|W|$. The *empty string*, denoted by ε, is a string of length zero. For a string $W \in \Sigma^*$ of length n, $W[i]$ denotes the i^{th} symbol of W, and $W[i:j] = W[i]W[i+1]\dots W[j]$ denotes a substring of W that begins at i and ends at j for $1 \le i \le j \le n$. For convenience, we abbreviate $W[1:i]$ to $W[:i]$ and $W[i:n]$ to $W[i:]$; these terms are known as the *prefix* and the *suffix* of W, respectively. Moreover, let $W[i:j] = \varepsilon$ if $i > j$. The reverse string of W is denoted by $W^R = W[n]W[n-1]\dots W[2]W[1]$. For two strings, X and Y, $X \prec Y$ denotes that X is lexicographically smaller than Y, and $X \preceq Y$ denotes that either $X = Y$ or $X \prec Y$.

A *multi-track symbol* $\mathbb{C} = (c_1, c_2, \dots, c_k)$ is a k-tuple of symbols $c_i \in \Sigma$. A *multi-track string* (or *multi-track* for short) $\mathbb{W} = (W_1, W_2, \dots, W_k)$ is a k-tuple of strings $W_i \in \Sigma^*$ where $|W_1| = |W_2| = \cdots = |W_k|$, and each W_i is called the i^{th} *track* of $\mathbb{W}$. The length n of strings in $\mathbb{W}$ is called the *length* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{len}$, and the number k of tracks in $\mathbb{W}$ is called the *track count* of $\mathbb{W}$ and is denoted by $|\mathbb{W}|_{num}$. For a multi-track symbol $\mathbb{C} = (c_1, c_2, \dots, c_k)$, we define $\mathbb{C}[i] = c_i$. For a multi-track $\mathbb{W}$ of track count $k = |\mathbb{W}|_{num}$, $\mathbb{W}[i] = (W_1[i], W_2[i], \dots, W_k[i])$ denotes the i^{th} multi-track symbol, and $\mathbb{W}[i][j]$ denotes $W_j[i]$ for $1 \le i \le |\mathbb{W}|_{len}$ and $1 \le j \le |\mathbb{W}|_{num}$. Moreover, $\mathbb{W}[i:j] = (W_1[i:j], W_2[i:j], \dots, W_k[i:j])$ denotes a substring of $\mathbb{W}$ that begins at i and ends at j for $1 \le i \le j \le |\mathbb{W}|_{len}$. Similarly to the notation for strings, $\mathbb{W}[:i]$ and $\mathbb{W}[i:]$ mean $\mathbb{W}[1:i]$ and $\mathbb{W}[i:|\mathbb{W}|_{len}]$, which are called the *prefix* and the *suffix* of $\mathbb{W}$, respectively.

Let $\mathbf{r} = (r_1, r_2, \dots, r_k)$ be a permutation of $(1, 2, \dots, k)$. For a multi-track $\mathbb{W} = (W_1, W_2, \dots, W_k)$, $\mathbb{W}\langle\mathbf{r}\rangle = \mathbb{W}\langle r_1, r_2, \dots, r_k\rangle = (W_{r_1}, \dots, W_{r_k})$ is called a *permuted multi-track* of $\mathbb{W}$. The *sorted index* $\mathsf{SI}(\mathbb{W})$ of a multi-track $\mathbb{W}$ is a permutation $(r_1, \dots, r_k)$ such that $W_{r_i} \preceq W_{r_{i+1}}$ for any $1 \le i < k$, where we assume $r_i < r_{i+1}$ in the case $W_{r_i} = W_{r_{i+1}}$. The *sorted multi-track* $\mathsf{sort}(\mathbb{W})$ is defined as $\mathbb{W}\langle\mathsf{SI}(\mathbb{W})\rangle$. The *reverse* of a multi-track is $\mathbb{W} = (W_1, \ldots, W_k)$ is $\mathbb{W}^R = (W_1^R, \ldots, W_k^R)$. The sorted index of the reverse multi-track, denoted by $\mathsf{RI}(\mathbb{W})$, is a permutation $(r_1, \ldots, r_k)$ such that $w_{r_i}^R \preceq w_{r_{i+1}}^R$ for any $1 \le i < k$.

Lemma 1 ([7]). *Let $SI_{\mathbb{W}}$ be a two-dimensional array such that $SI_{\mathbb{W}}[i] = \mathsf{SI}(\mathbb{W}[i:])$ for $1 \le i \le n$. $SI_{\mathbb{W}}$ can be computed in $O(nk)$ time* offline, *where $n = |\mathbb{P}|_{len}$ and $k = |\mathbb{P}|_{num}$.*

Lemma 2. *Let $RI_{\mathbb{W}}$ be a two-dimensional array such that $RI_{\mathbb{W}}[i] = \mathsf{RI}(\mathbb{W}[i:])$ for $1 \le i \le n$. $RI_{\mathbb{W}}$ can be computed in $O(n(k+\sigma))$ time* online, *where $n = |\mathbb{P}|_{len}$ and $k = |\mathbb{P}|_{num}$.*

Proof. Clearly, $RI_{\mathbb{W}}$ can be computed in $O(n(k+\sigma))$ by using a radix sort algorithm. □

For two multi-tracks $\mathbb{X} = (X_1, X_2, \ldots, X_k)$ and $\mathbb{Y} = (Y_1, Y_2, \ldots, Y_k)$ of the same length, we say that $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$ if $\mathbb{X} = \mathbb{Y}\langle \mathbf{r} \rangle$ for some permutation $\mathbf{r}$, and it is denoted by $\mathbb{X} \bowtie \mathbb{Y}$.

Lemma 3 ([7]). *For two multi-tracks $\mathbb{X}$ and $\mathbb{Y}$, $\mathbb{X} \bowtie \mathbb{Y}$ if and only if $\mathsf{sort}(\mathbb{X}) = \mathsf{sort}(\mathbb{Y})$.*

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

Definition 1 (Permuted pattern matching [7]). *Given a multi-track text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$, compute all positions* i *that satisfy $\mathbb{P} \bowtie \mathbb{T}[i : i+m-1]$. We call such positions* occurrence positions *of $\mathbb{P}$ in $\mathbb{T}$.*

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 $\mathbb{P}$ matches $\mathbb{T}$ at two because $\mathbb{T}[2:4] = \mathbb{P}$. Moreover, the pattern $\mathbb{P}$ permuted-matches $\mathbb{T}$ at six, since $\mathbb{P}\langle 3,2,1 \rangle = \begin{pmatrix} \texttt{aaa}, \\ \texttt{baa}, \\ \texttt{aba} \end{pmatrix} = \mathbb{T}[6:8]$. Therefore, we should output two and six in this case.

We remark that Katsura et al. [7] defined a more general problem called the *sub-permuted pattern matching*, where we have $|\mathbb{T}|_{num} \ge |\mathbb{P}|_{num}$, and our task is to find a partial permutation $(r_1, \ldots, r_{|\mathbb{P}|_{num}})$ of $(1, \ldots, |\mathbb{T}|_{num})$ and a position i for which $\mathbb{P} \bowtie \mathbb{T}\langle r_1, \ldots, r_{|\mathbb{P}|_{num}} \rangle [i : i+m-1]$ holds. However, in this paper, we only consider the case $|\mathbb{T}|_{num} = |\mathbb{P}|_{num}$, which was called *full-permuted pattern matching* in [7].

3. KMP-Based Permuted Pattern Matching Algorithms

In this section, we propose three algorithms for permuted pattern matching based on the KMP algorithm. In Section 3.1, we introduce our first algorithm, the *multi-track KMP algorithm* (MTKMP algorithm) which uses the border array of a multi-track pattern. We remark that this algorithm is a variant of the generalized KMP algorithm proposed by Matsuoka et al. [12]. They showed that the KMP algorithm can be applied to string classes that satisfy some equality and border properties. In this paper, we show some properties of multi-track strings so that the KMP algorithm can be applied to multi-track strings and give the exact computation time of the algorithm.

In Section 3.2, we extend it to the multi-track AC-automaton algorithm, which can perform dictionary matching on multi-tracks. Last, in Section 3.3, we introduce another extension of the MTKMP algorithm, the permuted matching automaton algorithm, which can perform permuted pattern matching faster than the MTKMP algorithm.

3.1. Multi-Track KMP Algorithm

Similar to the original KMP algorithms, the *multi-track KMP algorithm* uses *multi-track border arrays* to compute the shift amount when the algorithm finds a mismatch. First, we define borders for multi track strings.

Definition 2 (Borders of multi-tracks)**.** *A* border *of a multi-track $\mathbb{P}$ is any multi-track that permuted-matches with a prefix and a suffix of $\mathbb{P}$. Moreover, a border $\mathbb{B}$ of $\mathbb{P}$ is called* proper *if $\mathbb{B} \not\bowtie \mathbb{P}$.*

We can easily confirm the following lemma.

Lemma 4. *For any two borders $\mathbb{B}_1$ and $\mathbb{B}_2$ of $\mathbb{P}$, if $|\mathbb{B}_1|_{len} < |\mathbb{B}_2|_{len}$, then $\mathbb{B}_1$ is a border of $\mathbb{B}_2$.*

Let $\mathbb{T}[i : i+j] \bowtie \mathbb{P}[1 : j+1]$ for some i and j. If $\mathbb{T}[i+x : i+x+m-1] \bowtie \mathbb{P}[1 : m]$ for some $x \leq j$, then $\mathbb{T}[i : i+(j-x)] \bowtie \mathbb{P}[1 : (j-x)+1]$ and $\mathbb{T}[i+x : i+j] \bowtie \mathbb{P}[1 : (j-x)+1]$ hold, i.e., $\mathbb{P}[1 : (j-x)+1]$ is a proper border of $\mathbb{P}[1 : j+1]$. Therefore, if $\mathbb{T}[i : i+j] \bowtie \mathbb{P}[1 : j+1]$ and $\mathbb{T}[i : i+j] \not\bowtie \mathbb{P}[1 : j+1]$, we can safely shift $\mathbb{P}$ by x, where $(j-x+1)$ is the length of the longest proper border of $\mathbb{P}[1 : j+1]$. We store the length of the longest proper border of $\mathbb{P}[1 : j+1]$ for $1 \leq j \leq m$ as *the border array* of $\mathbb{P}$.

Definition 3 (Multi-track border array)**.** *Given a multi-track $\mathbb{P}$ of $|\mathbb{P}|_{len} = m$,* the border array *$Border_{\mathbb{P}}$ of $\mathbb{P}$ is an array of length $m+1$, where the i^{th} element of $Border_{\mathbb{P}}$ is the length of the longest proper border of $\mathbb{P}[1 : i]$. Formally,*

$$Border_{\mathbb{P}}[i] = \max\{j \mid \mathbb{P}[1 : j] \bowtie \mathbb{P}[i-j+1 : i], 0 \leq j < i\}. \qquad (1)$$

For algorithmic purpose, we define $Border_{\mathbb{P}}[0] = -1$. Algorithm 1 constructs the multi-track border array of an input $\mathbb{P}$.

Lemma 5. *Given a multi-track $\mathbb{P}$ of $|\mathbb{P}|_{len} = m$ and $|\mathbb{P}|_{num} = k$, Algorithm 1 constructs the border array of $\mathbb{P}$ in $O(mk)$ time.*

Proof. First, we show the correctness of the algorithm by induction. Assume we have computed the longest proper border of $\mathbb{P}[: j]$ and stored it in $Border[j]$ for $1 \leq j \leq i$. Let $\mathbb{P}[: b]$ be the longest proper border of $\mathbb{P}[: i+1]$. Clearly, $\mathbb{P}[: b-1]$ is a border of $\mathbb{P}[: i]$. Using Lemma 4, we can find $b-1$ by finding $\max\{j \mid \mathbb{P}[: j]$ is a proper border of $\mathbb{P}[: i]$ and $\mathbb{P}[: j+1] \bowtie \mathbb{P}[i-j+1 : i+1]\}$. Since $Border[j]$ is the longest proper border of $\mathbb{P}[: j]$, the algorithm can find $b-1$ by updating $j \leftarrow Border[j]$ from $j = i$ until $\mathbb{P}[: Border[j]+1] \bowtie \mathbb{P}[i - Border[j]+1 : i+1]$ holds. Therefore, Algorithm 1 computes the border array of $\mathbb{P}$ correctly.

Next, we show the computation time of the algorithm. Using Lemma 1, *SI* can be computed in $O(mk)$. Both **while** loops are called $O(m)$ times at most, since $i - j \leq i \leq m+1$, and the value of i always increases each time the outer loop is called, while the value of $i-j$ always increases each time the inner loop is called. Each comparison of $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[1 :]) \rangle$ and $\mathbb{P}[i]\langle \mathsf{SI}(\mathbb{P}[i-j :]) \rangle$ consumes $O(k)$ time; hence, Algorithm 1 runs in $O(mk)$ time. □

Algorithm 2 shows the pseudocode of the MTKMPalgorithm. The MTKMP algorithm performs permuted pattern matching from left to right of the pattern and the text. The sorted pattern $\mathbb{P}_{sort}$ and the array *SI* save the sorted indices of suffixes of text used to perform permuted-matching between the pattern and a substring of the text. If symbols on the text and the pattern are mismatched, we shift the matching position of the pattern using *Border*. If the pattern matches a substring of the text, then the MTKMP algorithm outputs the occurrence position at the text and shifts the pattern according to *Border*.

Algorithm 1: Multi-track border array construction algorithm.

```
Function constructMTBorderArray($\mathbb{P}$)
    compute $SI[i] \leftarrow \mathsf{SI}(\mathbb{P}[i:])$ for $1 \leq i \leq m$;
    $i \leftarrow 1$; $j \leftarrow 0$; $Border[0] \leftarrow -1$;
    while $i \leq m$ do
        while $j \geq 0$ and $\mathbb{P}[j+1]\langle SI[1]\rangle \neq \mathbb{P}[i]\langle SI[i-j]\rangle$ do
            $j \leftarrow Border[j]$;
        $i \leftarrow i+1$; $j \leftarrow j+1$;
        $Border[i] \leftarrow j$;
    return $Border[i]$;
```

Algorithm 2: The multi-track KMP algorithm.

```
Function MTKMP($\mathbb{T}, \mathbb{P}$)
    compute $SI[i] \leftarrow \mathsf{SI}(\mathbb{T}[i:])$ for $1 \leq i \leq n$;
    compute $\mathbb{P}_{sort} \leftarrow \mathsf{SI}(\mathbb{P})$;
    $Border[i] \leftarrow$ constructMTBorderArray($\mathbb{P}$);
    $i \leftarrow 1$; $j \leftarrow 0$;
    while $i \leq n$ do
        while $j \geq 0$ and $\mathbb{T}[i]\langle SI[i-j]\rangle \neq \mathbb{P}_{sort}[j+1]$ do
            $j \leftarrow Border[j]$;
        $i \leftarrow i+1$; $j \leftarrow j+1$;
        if $j = m$ then
            output $(i-j)$;
            $j \leftarrow Border[j]$;
```

Theorem 1. *Given a text $\mathbb{T}$ and a pattern $\mathbb{P}$, the MTKMP algorithm computes all occurrence positions of $\mathbb{P}$ in $\mathbb{T}$ in $O(nk + mk)$ time.*

Proof. According to the above arguments, we can shift $\mathbb{P}$ by x, where $(j - x + 1)$ is the length of the longest proper border of $\mathbb{P}[1 : j+1]$ if $\mathbb{T}[i : i+j] \bowtie \mathbb{P}[1 : j+1]$ and $\mathbb{T}[i : i+j] \not\bowtie \mathbb{P}[1 : j+1]$. Since $Border[i]$ is the longest proper border of $\mathbb{P}[: i]$, the algorithm finds the occurrence position of $\mathbb{P}$ in $\mathbb{T}$ correctly.

Using Lemma 1, SI can be computed in $O(nk)$. From Lemma 5, constructMTBorderArray($\mathbb{P}$) runs in $O(mk)$. Both **while** loops are called $O(n)$ times at most, since $i - j \leq i \leq n+1$, and the value of i always increases each time the outer loop is called, while the value of $i - j$ always increases each time the inner loop is called. Each comparison of $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[1:])\rangle$ and $\mathbb{T}[i]\langle \mathsf{SI}(\mathbb{T}[i-j:])\rangle$ consumes $O(k)$ time. Hence, Algorithm 2 runs in $O(nk + mk)$ time. □

3.2. Multi-Track AC-Automaton

In this subsection, we introduce a data structure called a multi-track AC-automaton that can perform dictionary matching on multi-tracks, namely *permuted dictionary matching*. Let $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ be a set of patterns of the same track count k, called a *dictionary*, and let $d = \sum_{i=1}^{r} m_i$ be the total length of the patterns in D, where $m_i = |\mathbb{P}_i|_{len}$. The multi-track AC-automaton of D, denoted by $\mathsf{MTAC}(D)$, consists of a set of states and three functions: *goto*, *failure*, and *output* functions. Figure 1 shows an example of $\mathsf{MTAC}(D)$.

For a dictionary D, the states of $\mathsf{MTAC}(D)$ are $\mathcal{S} = \{\mathsf{sort}(\mathbb{P}_i)[: j] \mid \mathbb{P}_i \in D, 0 \leq j \leq m_i\}$. Each state corresponds to a prefix of $\mathsf{sort}(\mathbb{P}_i)$ for some $\mathbb{P}_i \in D$. Unlike the original AC-automaton, the multi-track AC-automaton uses a multi-track symbol, instead of a single symbol, to define the goto function. The goto, failure, and output functions of $\mathsf{MTAC}(D)$ are defined as follows.

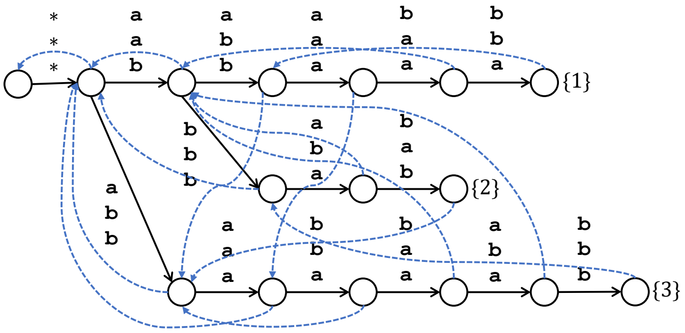


Figure 1. The multi-track AC-automaton $\mathsf{MTAC}(D)$ for $D = \{\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3\}$, where $\mathbb{P}_1 = (\mathtt{aaabb}, \mathtt{abaab}, \mathtt{bbaaa})$, $\mathbb{P}_2 = (\mathtt{abab}, \mathtt{abba}, \mathtt{bbab})$, and $\mathbb{P}_3 = (\mathtt{aabbab}, \mathtt{bababb}, \mathtt{baaaab})$. The asterisk "*" is a special symbol that matches any symbol in Σ.

Definition 4 (Goto function)**.** *We define the goto function δ of* $\mathsf{MTAC}(D)$ *by* $\delta(\mathsf{sort}(\mathbb{P}_i)[: j]), \mathbb{C}) = \mathsf{sort}(\mathbb{P}_i)[: j+1]$ *iff* $\mathbb{C} = \mathsf{sort}(\mathbb{P}_i)[j+1]$ *for a state* $\mathsf{sort}(\mathbb{P}_i)[: j]$ *and a multi-track symbol* $\mathbb{C}$.

Definition 5 (Failure function)**.** *The failure link of a state* $\mathsf{sort}(\mathbb{P}_i)[: j]$ *is defined as* $\mathsf{flink}(\mathsf{sort}(\mathbb{P}_i)[: j]) = \mathsf{sort}(\mathbb{P}_i[l : j])$*, where $\mathbb{P}_i[l : j]$ is the longest proper suffix of $\mathbb{P}_i[: j]$ such that $\mathbb{P}_i[l : j]$ permuted-matches a prefix of some pattern in D.*

Definition 6 (Output function)**.** *The output* $\mathsf{output}(\mathbb{P}_i[: j])$ *of a state* $\mathsf{sort}(\mathbb{P}_i)[: j]$ *is the set of patterns that permuted-matches $\mathbb{P}_i[l : j]$ for some $1 \le l \le j$.*

Next, we explain how to construct $\mathsf{MTAC}(D)$. We use *multi-track symbol tries* to implement the goto function.

Definition 7 (Multi-track symbol trie)**.** *Let $\mathcal{C} = \{\mathbb{C}_1, ..., \mathbb{C}_z\}$ be a set of multi-track symbols. The multi-track symbol trie of $\mathcal{C}$ is the trie for the set of strings $\{\mathbb{C}_1[1]\mathbb{C}_1[2]\ldots\mathbb{C}_1[|\mathbb{C}_1|_{num}], \ldots, \mathbb{C}_z[1]\mathbb{C}_z[2]\ldots\mathbb{C}_z[|\mathbb{C}_z|_{num}]\}$.*

For a state $s \in \mathcal{S}$, we use the multi-track symbol trie of a set $\{\mathbb{C} \mid \delta(s, \mathbb{C}) \ne \mathsf{NULL}\}$ and each leaf of multi-track symbol tries labeled by $\delta(s, \mathbb{C})$. Since a child of a node in multi-track symbol tries can be searched in $O(\log \sigma)$ time, $\delta(s, \mathbb{C})$ can be computed in $O(k \log \sigma)$ time. The goto function δ can be constructed by using Algorithm 3.

Algorithm 3: Multi-track AC-automaton goto function construction algorithm.

```
Function constructGotoFunction($D$)
    compute $SI[i][j] \leftarrow \mathsf{SI}(\mathbb{P}_i[j:])$ for $1 \le i \le r$ and $1 \le j \le m_i$;
    create states $root$ and $\perp$;
    $\mathsf{depth}(root) \leftarrow 0$; $\mathsf{depth}(\perp) \leftarrow -1$;
    $\delta(\perp, \mathbb{C}) \leftarrow root$ for any multi-track symbol $\mathbb{C}$ of track count $k$;
    for $i \leftarrow 1$ to $r$ do
        $v \leftarrow root$;
        for $1 \le j \le m_i$ do
            if $\delta(v, \mathbb{P}_i[j]\langle SI[i][1]\rangle) \ne \mathsf{NULL}$ then
                $v \leftarrow \delta(v, \mathbb{P}_i[j]\langle SI[i][1]\rangle)$;
            else
                create $newState$;
                $\delta(v, \mathbb{P}_i[j]\langle SI[i][1]\rangle) \leftarrow newState$;
                $\mathsf{label}(newState) \leftarrow i$;
                $\mathsf{depth}(newState) \leftarrow \mathsf{depth}(v) + 1$;
                $v \leftarrow newState$;
```

Lemma 6. *Algorithm 3 constructs the goto function of the multi-track AC-automaton of a dictionary* $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ *in* $O(dk \log \sigma)$ *time.*

Proof. The goto function $\delta(s, \mathbb{C})$ is executed $O(d)$ times, and $\delta(s, \mathbb{C})$ can be computed in $O(k \log \sigma)$ time. □

After constructing the goto function, we can construct the failure function of $\mathsf{MTAC}(D)$. Algorithm 4 shows a construction algorithm for the failure function of a multi-track AC-automaton. In order to simplify the construction algorithm, we use a special state that reads any multi-track symbol to get to the root state.

Algorithm 4: Multi-track AC-automaton failure function construction algorithm.

```
Function constructFailureFunction($D$)
    compute $SI[i][j] \leftarrow \mathsf{SI}(\mathbb{P}_i[j:])$ for $1 \le i \le r$ and $1 \le j \le m_i$;
    $\mathsf{flink}(root) \leftarrow \bot$;
    push $root$ to $queue$;
    while $queue \neq \emptyset$ do
        pop $v$ from $queue$;
        for $\mathbb{C}$ such that $\delta(v, \mathbb{C}) = u \neq \mathsf{NULL}$ do
            push $u$ to $queue$;
            $s \leftarrow \mathsf{flink}(v)$;
            $i \leftarrow \mathsf{label}(u)$;
            while $\delta(s, \mathbb{P}_i[\mathsf{depth}(u)]\langle SI[i][\mathsf{depth}(u) - \mathsf{depth}(s)]\rangle) = \mathsf{NULL}$ do
                $s \leftarrow \mathsf{flink}(s)$;
            $\mathsf{flink}(u) \leftarrow \delta(s, \mathbb{P}_i[\mathsf{depth}(u)]\langle SI[i][\mathsf{depth}(u) - \mathsf{depth}(s)]\rangle)$;
```

Lemma 7. *Algorithm 4 constructs the failure function of the multi-track AC-automaton of a dictionary* $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ *in* $O(dk \log \sigma)$ *time.*

Proof. We can bound the running time of Algorithm 4 by counting the number of executions of $\delta(s, \mathbb{C})$. For each pattern $\mathbb{P}_i$, let $s_{i,j}$ be a state such that $s_{i,j} = \delta(root, \mathbb{P}_i[:j])$ for $1 \le j \le m_i$. Let $f_{i,j}$ be the number of executions of flink when finding $\mathsf{flink}(s_{i,j})$. The maximum value of $f_{i,j}$ is bounded by $\mathsf{depth}(\mathsf{flink}(s_{i,j-1})) + 1$. Because the depth of $\mathsf{flink}(s_{i,j})$ is, at most, $\mathsf{depth}(\mathsf{flink}(s_{i,j-1})) - f_{i,j} + 1$, we get $f_{i,j} \le \mathsf{depth}(\mathsf{flink}(s_{i,j-2})) - f_{i,j-1} + 2$ recursively. By solving this formula, we get $\sum_{j=1}^{m_i} f_{i,j} \le 2m_i$ and $\sum_{i=1}^{r} \sum_{j=1}^{m_i} f_{i,j} \le \sum_{i=1}^{r} 2m_i = 2d$. Moreover, each $\delta(s, \mathbb{C})$ is executed in $O(k \log \sigma)$ time. □

Last, by using the goto and failure functions, Algorithm 4 constructs the output function of a multi-track AC-automaton.

Lemma 8. *Algorithm 5 constructs the output function of the multi-track AC-automaton of a dictionary* $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ *in* $O(dk \log \sigma)$ *time.*

Proof. Updating the output function can be performed in $O(1)$ time by using lists to save the output function and updating it by concatenating the list (see [13]). The proof is similar to the proofs of Lemmas 3 and 4. □

From Lemmas 6–8, we get the following theorem.

Theorem 2. *The multi-track AC-automaton of a dictionary* $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ *can be constructed in* $O(dk \log \sigma)$ *time.*

Algorithm 5: Multi-track AC-automaton output function construction algorithm.

```
Function constructOutputFunction($D$)
    compute $SI[i][j] \leftarrow \mathsf{SI}(\mathbb{P}_i[j:])$ for $1 \leq i \leq r$ and $1 \leq j \leq m_i$;
    for $i \leftarrow 1$ to $r$ do
        $v \leftarrow root$;
        for $1 \leq j \leq m_i$ do
            $v \leftarrow \delta(v, \mathbb{P}_i[j]\langle SI[i][1]\rangle)$;
            if $j = m_i$ then
                $\mathsf{output}(v) \leftarrow \mathsf{output}(v) \cup \{i\}$;
    push $root$ to $queue$;
    while $queue \neq \emptyset$ do
        pop $v$ from $queue$;
        $\mathsf{output}(v) \leftarrow \mathsf{output}(v) \cup \mathsf{output}(\mathsf{flink}(v))$;
        for $\mathbb{C}$ such that $\delta(activeState, \mathbb{C}) = s \neq \mathsf{NULL}$ do
            push $s$ to $queue$;
```

By using the multi-track AC-automaton of D, we can perform permuted dictionary matching on a text $\mathbb{T}$, as shown in Algorithm 6. Let v be the current state of the multi-track AC-automaton. For each position i on $\mathbb{T}$, Algorithm 6 uses the sorted index of $\mathbb{T}[i - \mathsf{depth}(v):]$ to determine the permutation of $\mathbb{T}[i]$ that is used in the goto function.

Algorithm 6: Permuted dictionary matching algorithm by using multi-track AC-automaton.

```
Function MTACADictionaryMatching($\mathbb{T}$)
    compute $SI[i] \leftarrow \mathsf{SI}(\mathbb{T}[i:])$ for $1 \leq i \leq n$;
    $v \leftarrow root$;
    for $1 \leq i \leq n$ do
        while $\delta(v, \mathbb{T}[i]\langle SI[i - \mathsf{depth}(v) + 1]\rangle) = \mathsf{NULL}$ do $v \leftarrow \mathsf{flink}(v)$;
        $v \leftarrow \delta(v, \mathbb{T}[i]\langle SI[i - \mathsf{depth}(v) + 1]\rangle)$;
        for $k \in \mathsf{output}(v)$ do output $(k, i - m_k + 1)$;
```

Theorem 3. *Permuted dictionary matching on a multi-track text $\mathbb{T}$ can be performed in $O(nk \log \sigma)$ time by using the multi-track AC-automaton of D.*

Proof. The running time of Algorithm 6 can be evaluated by counting the number of executions of $\delta(s, \mathbb{C})$. First, for each i, $\delta(s, \mathbb{C})$ is executed at least once on the v transition. Next, $\delta(s, \mathbb{C})$ is executed to check whether the transition is NULL or not. In this case, the number of executions of $\delta(s, \mathbb{C})$ is the same as that of flink. The latter is, at most, n, because whenever $\delta(s, \mathbb{C})$ is executed, the depth of v is increased by one, and whenever flink is executed, the depth of v is decreased by at least one. Therefore, the number of executions of $\delta(s, \mathbb{C})$ is $O(n)$. □

3.3. Multi-Track Permuted Matching Automaton

In this subsection, we 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 $\mathbb{P}$. For a multi-track pattern $\mathbb{P} = (P_1, P_2, ..., P_k)$, the multi-track permuted matching automaton of $\mathbb{P}$, denoted by $\mathsf{MTPMA}(\mathbb{P})$, consists of a set of states and two functions, goto and failure functions. In addition, each state of the multi-track permuted matching automaton has a weight in order to determine whether flink should be executed or not. Figure 2 shows an example of a multi-track permuted matching automaton.

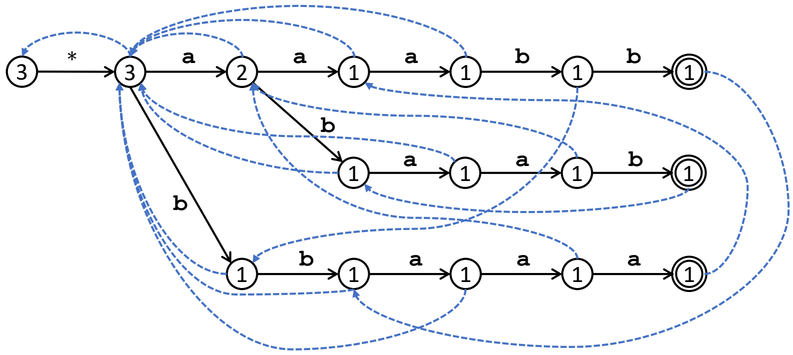


Figure 2. The multi-track permuted matching automaton $\mathsf{MTPMA}(\mathbb{P})$ for $\mathbb{P} = (\mathtt{aaabb}, \mathtt{abaab}, \mathtt{bbaaa})$. The asterisk is a special symbol that matches with any symbols in Σ.

Let $\mathcal{S}$ be the set of states of $\mathsf{MTPMA}(\mathbb{P})$. Each state of $\mathsf{MTPMA}(\mathbb{P})$ corresponds to a prefix of P_i, i.e., $\mathcal{S} = \{P_i[:j] \mid 1 \le i \le k,\ 0 \le j \le m\}$. Each state s has a weight, which is the number of tracks containing s as a prefix. Moreover, a state s is called an *accept state* if $s = P_i$ for some i. The goto and failure functions of $\mathsf{MTPMA}(\mathbb{P})$ are defined as follows.

Definition 8 (Goto function). *We define the goto function δ as $\delta(P_i[:j], c) = P_i[:j+1]$ iff $c = P_i[j+1]$ for each state $P_i[:j]$ and symbol c.*

Definition 9 (Failure function). *Let $S_j = \{P_i[:j] \mid 1 \le i \le k\}$ be the set of states whose depth is j. The failure link of the state is $\mathsf{flink}(P_i[:j]) = P_x[:y] \in S_y$ such that $P_x[:y]$ is a proper suffix of $P_i[:j]$ and $\mathbb{P}[:y]$ is the longest prefix of $\mathbb{P}$ that permuted-matches with a proper suffix of $\mathbb{P}[:j]$. The latter condition is similar to that of the multi-track KMP algorithm.*

Next, we explain how to construct $\mathsf{MTPMA}(\mathbb{P})$. Algorithm 7 constructs the goto function of a multi-track permuted matching automaton. We add a weight on each state each time we visit the state when reading each track.

Lemma 9. *Algorithm 7 constructs the goto function of $\mathsf{MTPMA}(\mathbb{P})$ in $O(mk \log \sigma)$ time.*

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

After constructing the goto function, Algorithm 8 constructs the failure function of $\mathsf{MTPMA}(\mathbb{P})$. For each state s, we use a multi-track border array to determine the depth of $\mathsf{flink}(s)$.

Lemma 10. *Algorithm 8 constructs the failure function of $\mathbb{P}$ in $O(mk \log \sigma)$ time.*

Proof. Similarly to the proof of Theorem 7, the failure and goto functions are executed $O(mk)$ times. Moreover, the execution time of the failure function is $O(1)$, and that of the goto function is $O(\log \sigma)$. □

From Lemmas 9 and 10 we get the following theorem.

Theorem 4. *The multi-track permuted matching automaton of a pattern $\mathbb{P}$ can be constructed in $O(mk \log \sigma)$ time.*

Finally, by using $\mathsf{MTPMA}(\mathbb{P})$, Algorithm 9 can perform permuted pattern matching on a multi-track text $\mathbb{T}$. Algorithm 9 uses k pointers *activeStates* to point to the current states. Note that all *activeStates* always have the same depth. Algorithm 9 uses two conditions to determine whether it should execute the failure function or not. 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. If any of the *activeStates* fail, then all of the *activeStates* execute the failure function, otherwise *activeStates* executes the goto function.

Theorem 5. *By using* MTPMA($\mathbb{P}$)*, Algorithm 9 performs permuted pattern matching on a multi-track string* $\mathbb{T}$ *in* $O(nk \log \sigma)$ *time.*

Proof. Similarly to the proof of Theorem 3, the number of executions of the failure and goto function is $O(nk)$. Since the execution time of the failure function is $O(1)$ and the goto function is $O(\log \sigma)$, Algorithm 9 runs in $O(nk \log \sigma)$ time. □

Algorithm 7: Multi-track permuted matching automaton goto function construction algorithm.

```
Function constructGotoFunction($\mathbb{P}$)
    create states $root$ and $\bot$;
    $\delta(\bot, c) \leftarrow root$ for all symbol $c \in \Sigma$;
    $newState \leftarrow root$;
    weight($\bot$) $\leftarrow$ weight($root$) $\leftarrow k$;
    for $1 \leq i \leq k$ do
        $v \leftarrow root$;
        for $1 \leq j \leq m$ do
            if $\delta(v, \mathbb{P}[j][i]) =$ NULL then
                create $newState$;
                $\delta(v, \mathbb{P}[j][i]) \leftarrow newState$;
                weight($newState$) $\leftarrow 1$;
                $v \leftarrow newState$;
            else
                $v \leftarrow \delta(v, \mathbb{P}[j][i])$;
                weight($v$) $\leftarrow$ weight($v$) $+ 1$;
            if $j = m$ then  set $v$ as an accept state;
```

Algorithm 8: Multi-track permuted matching automaton failure function construction algorithm.

```
Function constructFailureFunction($\mathbb{P}$)
    $activeStates \leftarrow \{root\}$;
    flink($root$) $\leftarrow \bot$;
    $Border_{\mathbb{P}}[i] \leftarrow$ constructMTBorderArray($\mathbb{P}$);
    for $1 \leq i \leq m$ do
        $temp \leftarrow \emptyset$;
        for $v \in activeStates$ do
            $u \leftarrow$ flink($v$);
            while depth($u$) $+ 1 \neq Border_{\mathbb{P}}[i]$ do  $u \leftarrow$ flink($v$);
            for $c$ such that $\delta(v, c) = s \neq$ NULL do
                $temp \leftarrow temp \cup \{s\}$;
                flink($s$) $\leftarrow \delta(u, c)$;
        $activeStates \leftarrow temp$;
```

Algorithm 9: Multi-track permuted matching automaton matching algorithm.

```
Function MTPMAMatching($\mathbb{T}$)
    $activeStates[i] \leftarrow root$ for $1 \leq i \leq k$;
    for $1 \leq i \leq n$ do
        $failFlag \leftarrow$ true;
        while $failFlag =$ true do
            $failFlag \leftarrow$ false;
            $failFlag \leftarrow$ isFail($activeStates, \mathbb{T}, i$);
            if $failFlag =$ true then
                for $j = 1$ to $k$ do $activeStates[j] \leftarrow$ flink($activeStates$);
            else
                for $j = 1$ to $|activeStates|$ do
                    $activeStates[j] \leftarrow \delta(activeStates[j], \mathbb{T}[i][j])$;
        if $activeStates[1]$ is an accept state then output $i - m + 1$;
Function isFail($activeStates, \mathbb{T}, i$)
    for $j = 1$ to $k$ do
        if $\delta(activeStates[j], \mathbb{T}[i][j]) =$ NULL then return true;
        else
            $nextState = \delta(activeStates[j], \mathbb{T}[i][j])$;
            temp($nextState$) $\leftarrow$ temp($nextState$) $+ 1$;
            if temp($nextState$) $>$ weight($nextState$) then return true;
    return false;
```

4. Multi-Track Boyer–Moore and Horspool Algorithms

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. The original Boyer–Moore algorithm uses two functions GoodSuf (good suffixes) and BadSym (bad symbols), and the Horspool algorithm only uses BadSym 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. For multi-track strings, we use a permuted match to define GoodSuf instead of a complete match, and we use sorted multi-track symbols for BadSym instead of symbols. Formally, those functions are defined as follows on multi-tracks.

Definition 10 (Good suffixes function). *For a multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, let $A[i] = \{0 < s < i \mid \mathbb{P}[i-s+1 : m-s] \bowtie \mathbb{P}[i+1 : m], \mathbb{P}[i-s : m-s] \not\bowtie \mathbb{P}[i : m]\}$ and $B[i] = \{i \leq s < m \mid \mathbb{P}[1 : m-s] \bowtie \mathbb{P}[s+1 : m]\}$. The good suffixes function is defined as $\mathsf{GoodSuf}_{\mathbb{P}}[m] = 1$ and $\mathsf{GoodSuf}_{\mathbb{P}}[i] = \min A[i] \cup B[i] \cup \{m\}$ for $0 \leq i < m$.*

Definition 11 (Bad symbols function). *For a multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$ and a multi-track symbol $\mathbb{C}$, $\mathsf{BadSym}_{\mathbb{P}}(\mathbb{C})$ is the first occurrence position of $\mathsf{sort}(\mathbb{C})$ in $\mathbb{P}^R[2:]$. The function $\mathsf{BadSym}_{\mathbb{P}}(\mathbb{C})$ returns m if there is no occurrence of $\mathsf{sort}(\mathbb{C})$ in $\mathbb{P}^R[2:]$.*

In the implementation, GoodSuf is computed by using an array, while BadSym can be computed by using a trie of the multi-track symbols. We perform permuted-match instead of exact matching when computing GoodSuf. We also use another array suf to compute GoodSuf.

Definition 12 (Suffixes). *For a multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, $\mathsf{suf}_{\mathbb{P}}[i]$ is the maximum value of l such that $\mathbb{P}[i-l+1 : i] \bowtie \mathbb{P}[m-l+1 : m]$ for $1 \leq i \leq m$.*

Algorithm 10 shows how to construct GoodSuf and BadSym. The array GoodSuf is computed by ComputeGoodSuf, which uses array suf computed by ComputeSuf. Note that we compute $\mathsf{RI}(\mathbb{P}[:\ i])$ at the beginning (Lines 2 and 26) of the algorithm and will not recompute them when we use the values later.

Algorithm 10: MT-BM and MT-H preprocessing functions.

```
1  Function ComputeSuf($\mathbb{P}$)
2      compute $RI[i] \leftarrow \mathsf{RI}(\mathbb{P}[:\ i])$ for $1 \le i \le m$;
3      $\mathsf{suf}[m] \leftarrow m, j \leftarrow m, l \leftarrow m$;
4      for $i \leftarrow m-1$ to $1$ do
5          if $i > l$ and $\mathsf{suf}[m-(j-i)] < i-l$ then
6              $\mathsf{suf}[i] \leftarrow \mathsf{suf}[m-(j-i)]$;
7          else
8              if $i < l$ then $l \leftarrow i$;
9              $j \leftarrow i$;
10             while $l > 0$ and $\mathbb{P}[l]\langle RI[j]\rangle = \mathbb{P}[l+m-j]\langle RI[m]\rangle$ do
11                 $k \leftarrow l-1$;
12             $\mathsf{suf}[i] \leftarrow j-l$ ;
13     return suf;
14 Function ComputeGoodSuf($\mathbb{P}$)
15     suf $\leftarrow$ ComputeSuf($\mathbb{P}$);
16     $j \leftarrow 1$;
17     for $i \leftarrow 1$ to $m$ do  $\mathsf{GoodSuf}[i] \leftarrow m$;
18     for $i \leftarrow m$ to $1$ do
19         if $\mathsf{suf}[i] = i$ then
20             while $j \le m-i$ do
21                 if $\mathsf{GoodSuf}[j] = m$ then $\mathsf{GoodSuf}[j] \leftarrow m-i$;
22                 $j \leftarrow j+1$;
23     for $i \leftarrow 1$ to $m-1$ do $\mathsf{GoodSuf}[m-\mathsf{suf}[i]] \leftarrow m-i$;
24     return GoodSuf;
25 Function ComputeBadSym($\mathbb{P}$)
26     compute $RI[i] \leftarrow \mathsf{RI}(\mathbb{P}[:\ i])$ for $1 \le i \le m$;
27     for $i \leftarrow 1$ to $m-1$ do
28         if $\mathsf{BadSym}(\mathbb{P}[i]\langle RI[i]\rangle) = m$ then
29             BadSym.add($\mathbb{P}[i]\langle RI[i]\rangle, m-i$);
30         else
31             $\mathsf{BadSym}(\mathbb{P}[i]\langle RI[i]\rangle) \leftarrow m-i$;
32     return BadSym;
```

Lemma 11. *The function ComputeSuf computes the array* suf *in $O(m(k+\sigma))$ time.*

Proof. First, RI can be computed in $O(m(k+\sigma))$ time by using radix sort. The **for** loop is executed $m-1$ times, and the **while** loop in Line 11 is executed m times at most through the whole run, because k is always reduced in each loop. A comparison of two multi-track symbols of the pattern that was executed in each loop can be computed in $O(k)$ time. □

Lemma 12. *The function ComputeGoodSuf computes* GoodSuf *in $O(m)$ time.*

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

Lemma 13. *The function ComputeBadSym computes* BadSym *in* $O(m(k \log \sigma + \sigma))$ *time.*

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

By using both $\mathsf{GoodSuf}_{\mathbb{P}}$ and $\mathsf{BadSym}_{\mathbb{P}}$, 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 11.

Algorithm 11: Multi-track Boyer–Moore algorithm.

```
1  Function MTBM($\mathbb{T}$, $\mathbb{P}$)
2      compute $RI[i] \leftarrow \mathsf{RI}(\mathbb{T}[:i])$ for $1 \le i \le n$;
3      compute $\mathbb{P}_{sort} \leftarrow \mathsf{RI}(\mathbb{P})$;
4      BadSym $\leftarrow$ ComputeBadSym($\mathbb{P}$);
5      GoodSuf $\leftarrow$ ComputeGoodSuf($\mathbb{P}$);
6      $j \leftarrow 0$;
7      while $j \le n - m + 1$ do
8          $i \leftarrow m$;
9          while $i > 0$ and $\mathbb{T}[i+j]\langle RI[j+m]\rangle = \mathbb{P}_{sort}[i]$ do
10             $i \leftarrow i - 1$;
11         if $i \le 0$ then
12             output $j + 1$;
13             $j \leftarrow j + \mathsf{GoodSuf}[0]$;
14         else
15             $j \leftarrow j + \max(\mathsf{GoodSuf}[i], \mathsf{BadSym}(\mathbb{T}[i+j]\langle RI[i+j]\rangle) - (m - i))$;
```

Theorem 6. *Given a multi-track text* $\mathbb{T}$ *and a pattern* $\mathbb{P}$*, MT-BM outputs all occurrence positions of* $\mathbb{P}$ *in* $\mathbb{T}$ *in* $O(nk(m + \log \sigma) + n(k + \sigma))$ *time with* $O(m(k \log \sigma + \sigma))$ *preprocessing time.*

Proof. From Lemmas 11–13, Algorithm 11 needs $O(m(k \log \sigma + \sigma))$ time for preprocessing. Next, RI can be computed in $O(n(k + \sigma))$ time by using radix sort. In the outer **while** loop starting at Line 7, the value of j is increased by at least one, so the loop is executed $n - m + 2$ times at most. In each execution of the outer loop, the inner **while** loop is executed m times at most, where multi-track symbols of the pattern and the text can be compared in $O(k)$ time. BadSym can be accessed in $O(k \log \sigma)$ time, and GoodSuf can be executed in $O(1)$ time. □

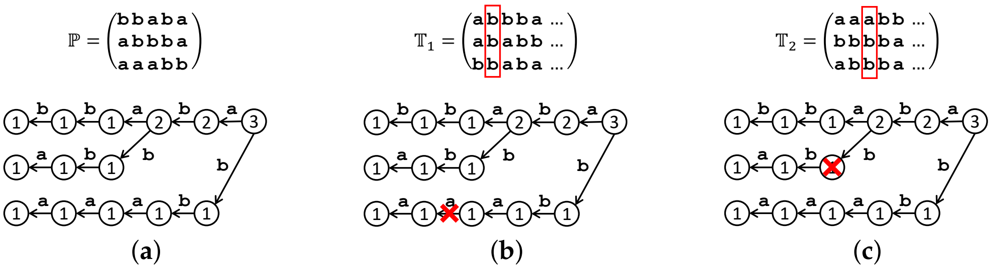
Similarly to the original Horspool algorithm, the multi-track Horspool algorithm (MT-H) uses BadSym to shift the pattern.

Theorem 7. *Given a multi-track text* $\mathbb{T}$ *and a pattern* $\mathbb{P}$*, MT-H outputs all occurrence positions of* $\mathbb{P}$ *in* $\mathbb{T}$ *in* $O(nk(m + \log \sigma) + n(k + \sigma))$ *time with* $O(m(k \log \sigma + \sigma))$ *preprocessing time.*

Proof. Similar to the proof of Theorem 6. □

Boyer–Moore and Horspool Matching Algorithms with Track Trie

The two algorithms presented in the previous subsections determine if two multi-tracks permuted-match by sorting them. In this subsection, we present another idea for this task using a data structure called a *track trie*. The track trie $\mathsf{TrackTrie}(\mathbb{P})$ of a multi-track $\mathbb{P}$ stores all the reversed strings of the tracks of $\mathbb{P}$, that is $\{P_1^R, P_2^R, \ldots, P_k^R\}$. Figure 3a shows the track trie of a multi-track pattern $\mathbb{P} = (\texttt{bbaba}, \texttt{abbba}, \texttt{aaabb})$.



ℙ = (bbaba / abbba / aaabb)
𝕋1 = (abbba ... / ababb ... / bbaba ...)
𝕋2 = (aaabb ... / bbbba ... / abbba ...)
(a)
(b)
(c)

Figure 3. (**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.

Algorithm 12 is the construction algorithm of TrackTrie($\mathbb{P}$). For a node s of TrackTrie($\mathbb{P}$) and a symbol $c \in \Sigma$, the goto function $\delta(s, c)$ returns the child of s that has an edge labeled c. We naturally extend it to the domain Σ^* by $\delta(s, \varepsilon) = s$ and $\delta(s, aw) = \delta(\delta(s, a), w)$ 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 explain later.

Algorithm 12: Track trie construction algorithm.

```
1  Function constructTrackTrie($\mathbb{P}$)
2      $newNode \leftarrow root$;
3      weight$(root) \leftarrow k$;
4      for $i \leftarrow 1$ to $k$ do
5          $activeNode \leftarrow root$;
6          for $j \leftarrow m$ to 1 do
7              if $\delta(activeNode, \mathbb{P}[j][i]) =$ NULL then
8                  $newNode \leftarrow newNode + 1$;
9                  weight$(newNode) \leftarrow 1$;
10                 $\delta(activeNode, \mathbb{P}[j][i]) \leftarrow newNode$;
11                 $activeNode \leftarrow newNode$;
12             else
13                 $activeNode \leftarrow \delta(activeNode, \mathbb{P}[j][i])$;
14                 weight$(activeNode) \leftarrow$ weight$(activeNode) + 1$;
```

Theorem 8. *Algorithm 12 constructs TrackTrie($\mathbb{P}$) in $O(mk \log \sigma)$ 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 6), Algorithm 12 executes *goto* to check the 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 12 accesses the child node and then increases its weight by one. The total number of iterations of the inner loop is mk. □

For a given multi-track text $\mathbb{T}$ and a position i, Algorithm 13 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 $\delta(root, w)$. Those mismatch conditions are illustrated in Figure 3b,c, respectively. Figure 3b shows that the track trie cannot find a transition for the second symbol b of the third track. On the other hand, Figure 3c 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.

Algorithm 13: Track trie matching algorithm.

```
Function matchTrackTrie($\mathbb{T}, j$)
    $activeNode[l] \leftarrow root$ for $1 \leq l \leq k$;
    $temp(node) \leftarrow 0$ for all $node$ in TrackTrie($\mathbb{P}$);
    for $i \leftarrow m$ to 1 do
        for $l \leftarrow 1$ to $k$ do
            if $\delta(activeNodes[l], \mathbb{T}[i+j][l]) =$ NULL then return $i$;
            else
                $activeNodes[l] \leftarrow \delta(activeNodes[k], \mathbb{T}[i+j][l])$;
                $temp(activeNodes[l]) \leftarrow temp(activeNodes[l]) + 1$;
                if $temp(activeNodes[l]) >$ weight$(activeNodes[l])$ then
                    return $i$;
    return 0;
```

Theorem 9. *Given a multi-track text $\mathbb{T}$ and a position j, Algorithm 13 finds a mismatch position in the pattern in $O(mk \log \sigma)$ time.*

Proof. For each position $i + j$ on the text, Algorithm 13 executes *goto* to check whether $activeNodes[l]$ has a child node with an edge labeled $\mathbb{T}[i + j][l]$ for $1 \leq l \leq k$. If there is no child node with an edge labeled $\mathbb{T}[i + j][l]$, then Algorithm 13 considers it a mismatch and returns the mismatch position. On the other hand, if there is such a child node, Algorithm 13 updates $activeNodes[l]$ to the child node and then checks whether the number of tracks of $\mathbb{T}[i + j :]$ that contain $\mathbb{T}[i + j : i + m][l]$ as a prefix is more than the weight of the child node. If the number of tracks exceeds the weight, then Algorithm 13 treats it as mismatch and returns the mismatch position. The total number of iterations of the inner loop is, at most, mk. □

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 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 10 of Algorithm 11 by matchTrackTrie($\mathbb{T}, j$) to find a mismatch position.

5. Filtering Algorithm on a Multi-Track String

In this section, we propose filtering algorithms for permuted pattern matching. The filtering algorithm uses a function to transform a multi-track string into another sequence. Then, the algorithm implements a pattern matching algorithm on the transformed pattern and text to get candidate positions where the pattern may permuted-match. Finally, every candidate position is checked for whether the pattern is permuted-matched in each position or not.

The filtering algorithm uses a function ϕ that inputs a multi-track $\mathbb{W}$ and outputs a sequence of length $|\mathbb{W}|_{len}$. The function ϕ must have a *false-positive* property, that is, for two multi-tracks $\mathbb{X}$ and $\mathbb{Y}$, if $\mathbb{X} \bowtie \mathbb{Y}$, then $\phi(\mathbb{X}) = \phi(\mathbb{Y})$. We can use any hash function such as the Karp–Rabin fingerprint [14] and locality-sensitive hashing [15]. We describe the filtering algorithm by using a simple function β that is defined as follows.

Definition 13 (Bucket function β)**.** *For $\mathbb{W} = (W_1, W_2, \ldots, W_k)$, $\beta(\mathbb{W}) = (B_1, B_2, \ldots, B_\sigma)$ such that $B_j[i]$ is the number of the j^{th} symbol in $\mathbb{Z}[i]$.*

For example, for a multi-track $\mathbb{W} = (\texttt{abab}, \texttt{bbac}, \texttt{aabb}, \texttt{cabb}, \texttt{abba})$, $\beta(\mathbb{W})$ is as follows,

$$\begin{pmatrix} \texttt{a} & \texttt{b} & \texttt{a} & \texttt{b} \\ \texttt{b} & \texttt{b} & \texttt{a} & \texttt{c} \\ \texttt{a} & \texttt{a} & \texttt{b} & \texttt{b} \\ \texttt{c} & \texttt{a} & \texttt{b} & \texttt{b} \\ \texttt{a} & \texttt{b} & \texttt{b} & \texttt{a} \end{pmatrix}, \ \beta(\mathbb{Z}) = \begin{pmatrix} 3 & 2 & 2 & 1 \\ 1 & 3 & 3 & 3 \\ 1 & 0 & 0 & 1 \end{pmatrix}. \tag{2}$$

$\beta(\mathbb{W})$ can be computed in $O(n(k+\sigma))$ time by counting all symbols in $\mathbb{W}$.

Algorithm 14 shows an example of filtering algorithm for permuted pattern matching. This algorithm uses β as a hash function, and the KMP algorithm as a patten matching algorithm. First, the algorithm computes $\mathbb{P}' = \beta(\mathbb{P})$ and $\mathbb{T}' = \beta(\mathbb{T})$ and then constructs the border array $Border_{\mathbb{P}'}$ using the same procedure as the KMP algorithm. Next, the algorithm performs pattern matching on $\mathbb{P}'$ and $\mathbb{T}'$. When a mismatch occurs, the pattern is shifted by using $Border_{\mathbb{P}'}$. If $\mathbb{P}'$ matches $\mathbb{T}'[i:j]$, then the algorithm checks whether the pattern $\mathbb{P}$ permuted-matches $\mathbb{T}[i:j]$ or not, and it outputs the position if true.

Algorithm 14: Filtering algorithm for permuted pattern matching.

```
Function multiTrackFiltering($\mathbb{P}$)
    $\mathbb{T}' \leftarrow \beta(\mathbb{T})$; $\mathbb{P}' \leftarrow \beta(\mathbb{P})$;
    constructBorderArray($\mathbb{P}'$);
    $i \leftarrow 1$; $j \leftarrow 1$;
    while $i \leq n$ do
        while $j > 0$ and $\mathbb{T}'[i] \neq \mathbb{P}'[j]$ do $j \leftarrow Border[j]$;
        $i \leftarrow i+1$; $j \leftarrow j+1$;
        if $j > m$ then
            if $\mathbb{T}[i-j+1 : i-1] \bowtie \mathbb{P}$ then output $(i-j+1)$;
            $j \leftarrow Border[j]$;

Function constructBorderArray($\mathbb{P}'$)
    $i \leftarrow 1$; $j \leftarrow 1$;
    $Border[1] \leftarrow 0$;
    while $i \leq m$ do
        while $j > 0$ and $\mathbb{P}'[i] \neq \mathbb{P}'[j]$ do $j \leftarrow Border[j]$;
        $i \leftarrow i+1$; $j \leftarrow j+1$;
        if $i \leq m$ and $\mathbb{P}'[i] = \mathbb{P}'[j]$ then $Border[j] \leftarrow Border[i]$;
        else $Border[j] \leftarrow i$;
```

Theorem 10. *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, Algorithm 14 computes all occurrence positions of $\mathbb{P}$ in $\mathbb{T}$ in $O((m+n)(\sigma+k)+cmk)$ time, where c is the number of candidates.*

Proof. As described above, $\mathbb{P}' = \beta(\mathbb{P})$ and $\mathbb{T}' = \beta(\mathbb{T})$ can be computed in $O(m(\sigma+k))$ and $O(n(\sigma+k))$, respectively. The border array $Border_{\mathbb{P}'}$ can be constructed in $O(m\sigma)$ time, and pattern matching can be performed in $O(n\sigma)$ time, since the comparison $\mathbb{P}'[i] = \mathbb{P}'[i]$ needs $O(\sigma)$ time. In addition, the algorithm takes $O(cmk)$ time to check the candidates. □

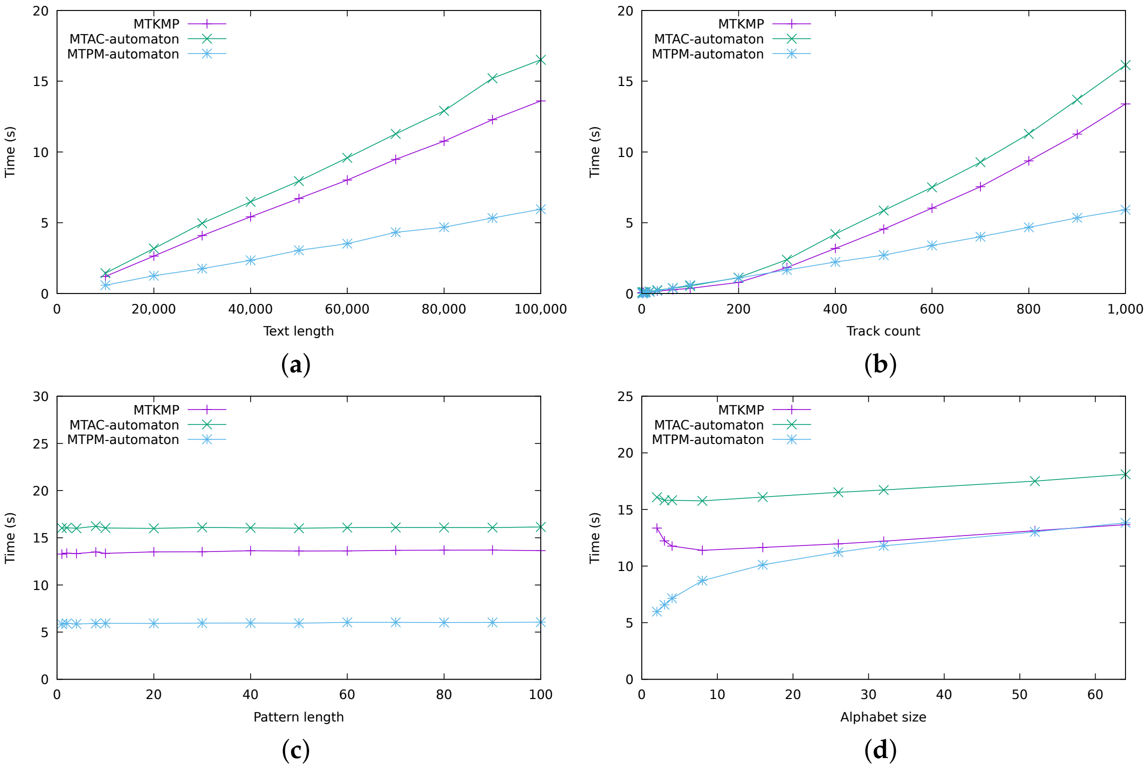
6. Experiments

We performed some sets of experiments to evaluate the practical performance of the proposed algorithms. We measured the running time of our algorithms and the AC-automaton-based algorithm [7]. All experiments were performed on a computer with an Intel Xeon CPU E5-2609 8 core, 2.40 GHz, 256 GB memory, and a Debian Wheezy operating system. We set the parameter values as n = 100,000, m = 10, k = 1000, and σ = 2, and we changed one of the parameters in each experiment to see the relation between the parameters and the running time of the algorithms.

We used randomly-generated texts and patterns. The source code of our algorithm is available at https://github.com/ushitora/permutedpatternmatching.

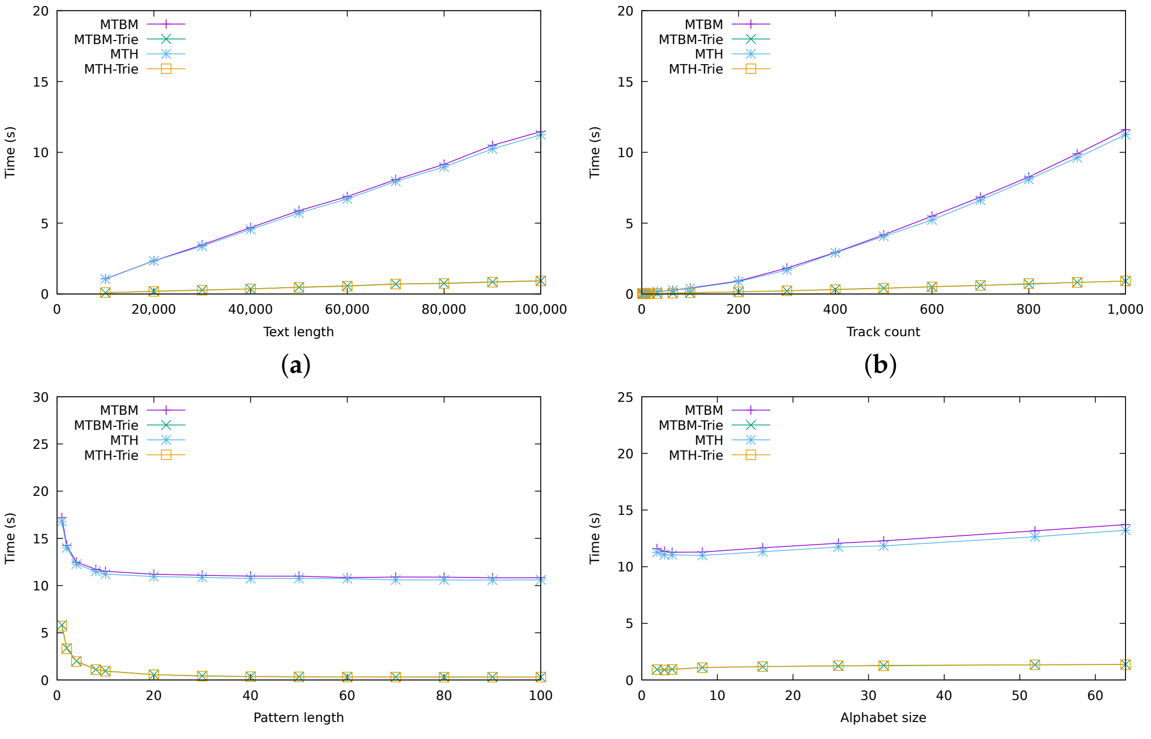
The first set of experiments compared the running times of KMP-based algorithms, namely the multi-track KMP algorithm, the multi-track AC-automaton algorithm, and the multi-track permuted matching automaton algorithm. The purpose of this experiment was to see the changes in the running time of the multi-track KMP algorithm when extended to the multi-track AC-automaton algorithm and the multi-track permuted matching automaton algorithm. The results of the experiment are shown in Figure 4. From the results, we can see that the multi-track AC-automaton algorithm was slower than the multi-track KMP algorithm. The reason for this is that the multi-track AC-automaton algorithm uses state transitions instead of comparing multi-track symbols directly. This time difference is not significant considering that the multi-track AC-automaton algorithm focuses on dictionary matching instead of single pattern matching. Next, we can see that the multi-track permuted matching automaton algorithm is faster than the multi-track KMP algorithm. The multi-track permuted matching automaton algorithm can perform matching faster because the multi-track permuted matching automaton algorithm does not need to sort the text.

The next experiment compared the running time of the multi-track Boyer–Moore algorithm and the multi-track Horspool algorithm. We also checked the effectiveness of track tries in reducing the matching times of both algorithms. Figure 5 shows the result of the experiment. While the multi-track Horspool was slightly faster than the multi-track Boyer–Moore algorithm, there was no significant difference between these algorithms. We can also see that track tries significantly reduced the matching times of both algorithms, since we do not need to sort the text when using track tries.



(**a**) (**b**) (**c**) (**d**)

Figure 4. Running time of the multi-track KMP (MTKMP), multi-track AC-automaton (MTACA), multi-track permuted matching automaton (MTPMA) algorithms on permuted pattern matching with respect to (**a**) the text length, (**b**) the track count, (**c**) the pattern length, and (**d**) the alphabet size.



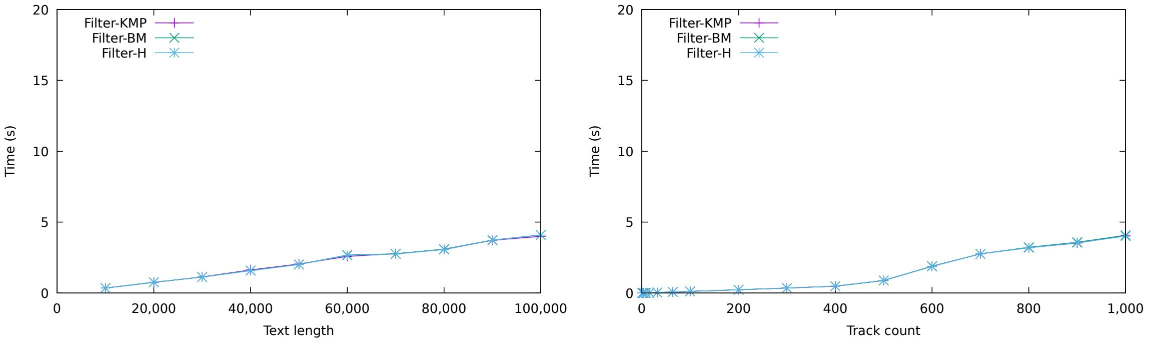
(**a**) (**b**)

(**c**) (**d**)

Figure 5. Running time of the multi-track Buyer–Moore (MTBM) and multi-track Horspool (MTH) algorithms on permuted pattern matching with respect to (**a**) the text length, (**b**) the track count, (**c**) the pattern length, and (**d**) the alphabet size.

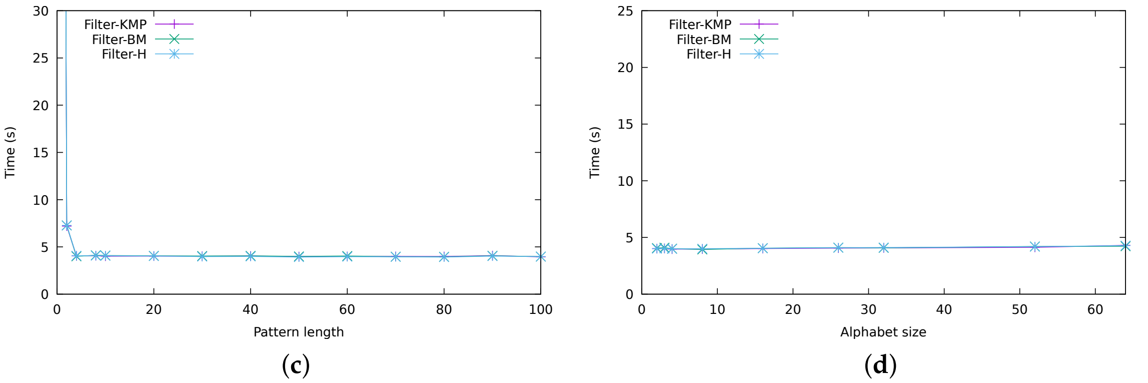
The third experiment set compared the running times of the filtering algorithms. We used three algorithms, the KMP algorithm, the Boyer–Moore algorithm, and the Horspool algorithm, as filtering algorithms. Figure 6 shows the running times of the filtering algorithms. There was no significant difference in running time between the filtering algorithms. We can see that the filtering algorithms were fast apart from when the pattern length was $m = 1$. The algorithms became very slow, because there were many occurrence position candidates when $m = 1$. The occurrence position candidates will be fewer if the pattern length is longer.

Lastly, we compared the proposed algorithms with the AC-automaton-based algorithm [7]. Figure 7 shows the running time of the AC-automaton-based algorithm [7] and the proposed algorithms. We chose the fastest algorithms among the groups. We can see that the proposed algorithms were faster than the AC-automaton-based algorithm overall. The multi-track Boyer–Moore algorithm with track trie was the fastest among the algorithms.



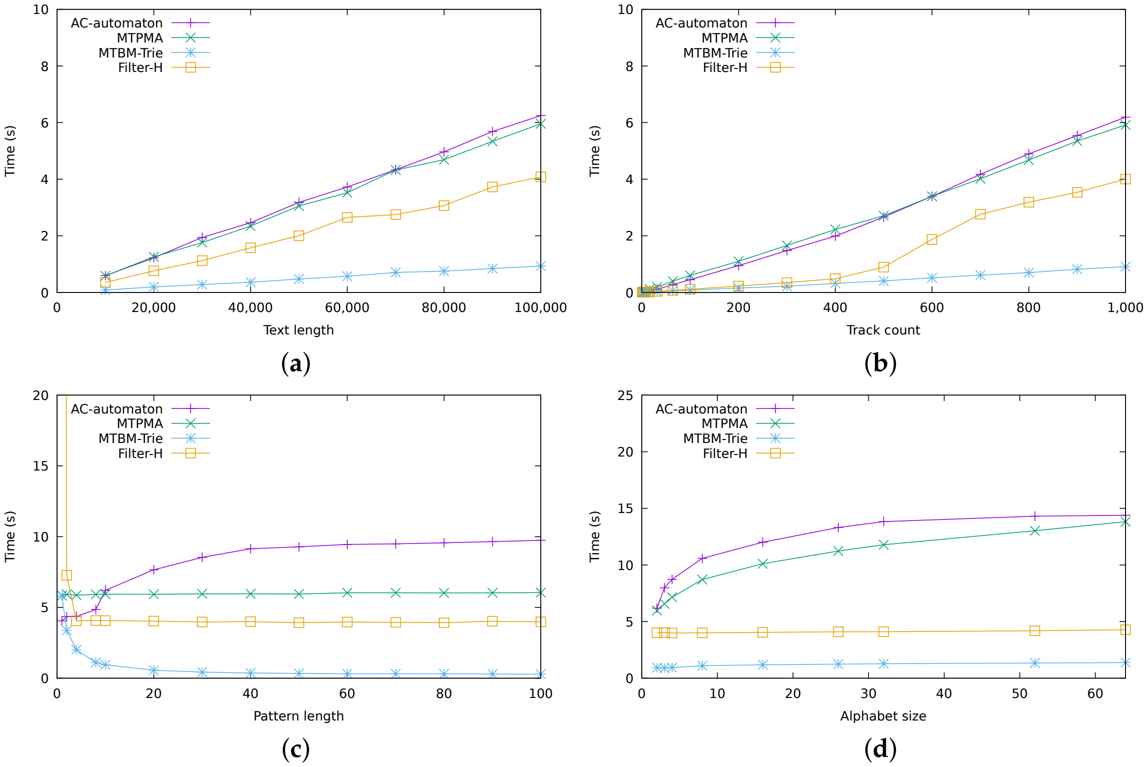
(**a**) (**b**)

Figure 6. *Cont.*



(**c**) (**d**)

Figure 6. Running time of the filtering algorithms (Knuth–Morris–Pratt (Filter-KMP), Boyer–Moore (Filter-BM), and Horspool (Filter-H)) for permuted pattern matching with respect to (**a**) the text length, (**b**) the track count, (**c**) the pattern length, and (**d**) the alphabet size.



(**a**) (**b**)

(**c**) (**d**)

Figure 7. Comparison of the running time of the AC-automaton-based algorithm with the proposed algorithms for permuted pattern matching with respect to (**a**) the text length, (**b**) the track count, (**c**) the pattern length, and (**d**) the alphabet size.

7. Conclusions

We proposed some algorithms for permuted pattern matching on multi-track strings. We primarily focused on the algorithms that preprocess the pattern before performing permuted pattern matching, instead of constructing indexing structures from the text. We showed that our proposed algorithms are faster than the AC-automaton-based algorithm. Moreover, we proposed an algorithm for dictionary matching on multi-track strings, which, to the best of our knowledge, is the first algorithm for this problem. Because of its considerable difficulty owing to a larger number of permutations, only a few algorithms for sub-permuted pattern matching have been proposed. Therefore, developing an algorithm for the sub-permuted pattern matching problem must be addressed in future work.

Author Contributions: Conceptualization, D.H., K.N., R.Y. and A.S.; methodology, D.H., Y.U., R.Y. and A.S.; software, D.H.; validation, D.H., R.Y. and A.S.; formal analysis, D.H., K.N., R.Y. and A.S.; investigation, D.H.; data curation, D.H.; writing—original draft preparation, D.H.; writing—review and editing, D.H., R.Y. and A.S.; visualization, D.H.; funding acquisition, D.H. and A.S.

Funding: This work was supported by JSPS KAKENHI Grant Numbers JP15H05706, JP24106010, JP25240003, and JP19K20208. This work was also supported by ImPACT Program of the Council for Science, Technology and Innovation (Cabinet Office, Government of Japan). Diptarama Hendrian was supported by a research grant from Tohoku University Division for International Advanced Research and Education.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Knuth, D.E.; Morris, J.H., Jr.; Pratt, V.R. Fast Pattern Matching in Strings. *SIAM J. Comput.* **1977**, *6*, 323–350. [CrossRef]
2. Boyer, R.S.; Moore, J.S. A fast string searching algorithm. *Commun. ACM* **1977**, *20*, 762–772. [CrossRef]
3. Horspool, R.N. Practical fast searching in strings. *Softw. Pract. Exp.* **1980**, *10*, 501–506. [CrossRef]
4. Weiner, P. Linear pattern matching algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973), Iowa City, IA, USA, 15–17 October 1973; pp. 1–11. [CrossRef]
5. Manber, U.; Myers, G. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* **1993**, *22*, 935–948. [CrossRef]
6. Ehrenfeucht, A.; McConnell, R.M.; Osheim, N.; Woo, S.W. Position heaps: A simple and dynamic text indexing data structure. *J. Discret. Algorithms* **2011**, *9*, 100–121. [CrossRef]
7. Katsura, T.; Narisawa, K.; Shinohara, A.; Bannai, H.; Inenaga, S. Permuted Pattern Matching on Multi-track Strings. In Proceedings of the SOFSEM 2013: 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, 26–31 January 2013; pp. 280–291.
8. Katsura, T.; Otomo, Y.; Narisawa, K.; Shinohara, A. Position Heaps for Permuted Pattern Matching on Multi-Track Strings. In Proceedings of the Student Research Forum Papers and Posters at SOFSEM 2015, Pec pod Snezkou, Czech Republic, 24–29 January 2015; pp. 41–53.
9. Narisawa, K.; Katsura, T.; Ota, H.; Shinohara, A. Filtering Multi-set Tree: Data Structure for Flexible Matching Using Multi-track Data. *Interdiscip. Inf. Sci.* **2015**, *21*, 37–47. [CrossRef]
10. Diptarama, Y.U.; Narisawa, K.; Shinohara, A. KMP based pattern matching algorithms for multi-track strings. In Proceedings of the Student Research Forum Papers and Posters at SOFSEM 2016, Harrachov, Czech Republic, 23–28 January 2016; Volume 2016, pp. 100–107.
11. Diptarama, Y.U.; Yoshinaka, R.; Shinohara, A. Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings. In Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, 29–31 August 2016; pp. 7–21.
12. Matsuoka, Y.; Aoki, T.; Inenaga, S.; Bannai, H.; Takeda, M. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theor. Comput. Sci.* **2016**, *656*, 225–233. [CrossRef]
13. Aho, A.V.; Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Commun. ACM* **1975**, *18*, 333–340. [CrossRef]
14. Karp, R.M.; Rabin, M.O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* **1987**, *31*, 249–260. [CrossRef]
15. Indyk, P.; Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing—STOC '98, Dallas, TX, USA, 24–26 May 1998; ACM Press: New York, NY, USA, 1998; pp. 604–613. [CrossRef]