

Permuted Pattern Matchings on Multi-track Strings (マルチトラック文字列上の順列パターン照合)



東北大学

Takashi Katsura

Graduate School of Information Sciences
Tohoku University, Japan

A thesis submitted in partial fulfillment of the requirements
for the degree of *Doctor of Philosophy in Engineering*

January 2015

Acknowledgment

First of all, I would like to express my sincere gratitude to my supervisor, Professor Ayumi Shinohara. He gave me a lot of advice and guidance during my research and study at Tohoku University. This thesis would not have been possible without his continual help and encouragement.

I would also like to express my deepest appreciation to Professor Takeshi Tokuyama and Professor Xiao Zhou, who are the members of my graduate committee, for their insightful suggestions and comments.

I would like to convey my gratitude and appreciation to Research Associate Kazuyuki Narisawa. He gave me valuable suggestions and provided a good research environment. Without his help, this thesis would never have been completed. I would also like to specially thank Associate Professor Hideo Bannai and Associate Professor Shunsuke Inenaga of Kyushu University, for their helpful suggestions and discussions.

My deepest appreciation also goes to staffs and colleagues of Shinohara Laboratory. Naomi Ogasawara and Ryoko Arijii provided a lot of helpful supports. All the colleagues in the lab made my life more precious. In particular, Dr. Kazuhiko Kusano taught me not only about researches but also how to enjoy my student life, for example programming contests. Yuhei Otomo, Hiroyuki Ota and Ryosuke Okuta helped me with writing papers. Yusuke Sato and Diptarama made my lab life more joyful.

I would also like to express my gratitude to Japan Student Services Organization for their financial support.

Finally, I would like to express special thanks to my family and friends for their moral support and warm encouragements.

Takashi Katsura

January 22, 2015

Abstract

In this thesis, we propose a new variant of pattern matching called permuted pattern matching. A multi-set of strings over an alphabet Σ is called a *multi-track string* or *multi-track*, denoted by $\mathbb{T} = \{t_1, \dots, t_N\}$. The permuted pattern matching problem is that given a multi-track text $\mathbb{T} = \{t_1, \dots, t_N\}$ of length n and a multi-track pattern $\mathbb{P} = \{p_1, \dots, p_M\}$ of length m , outputs all positions i such that $\{p_1, \dots, p_M\} \subseteq \{t_1[i : i + m - 1], \dots, t_N[i : i + m - 1]\}$, where $t_j[b : e]$ is a substring of t_j that begins at b and ends at e . The permuted pattern matching provides a new analysis of data represented as multiple sequences, such that multi-sensor data, polyphonic music data and traffic data.

At first we address an exact version of permuted pattern matching problems. We consider three settings of the problem with respect to whether the text and the pattern are allowed to be preprocessed or not. For each setting, we propose efficient algorithms.

Next we tackle an approximate version of permuted pattern matchings. We consider multi-track sequences of numbers, called *multi-track numerical sequence* or *numerical multi-track*. Then, we define a distance D between such two multi-track numerical sequences. The approximate permuted pattern matching problem is that given a multi-track text $\mathbb{T} = \{t_1, \dots, t_N\}$ of length n , a multi-track pattern $\mathbb{P} = \{p_1, \dots, p_M\}$ of length m , and a criterion $\delta \geq 0$, outputs all positions i such that $D(\mathbb{T}[i : i + m - 1], \mathbb{P}) \leq \delta$ for $1 \leq i \leq n$. For this problem, we propose a new data structure that is based on hash functions. By using our data structure, both of the exact and approximate permuted pattern matching problems can be solved efficiently.

Contents

1	Introduction	1
1.1	Background	1
1.2	Previous work	4
1.3	Contributions	5
1.4	Organization of the thesis	6
2	Preliminaries	8
2.1	Strings	8
2.2	Tries	8
2.3	Sequence hash trees	9
2.4	Various queries on rooted trees	10
2.5	Indexing structures for strings	11
2.6	Spectral Bloom filter	12
2.7	Hash functions	15
2.7.1	Rolling hash	15
2.7.2	Locality sensitive hashing	15
3	Multi-tracks	16
3.1	Multi-track strings	16
3.2	Numerical multi-track sequences	19

CONTENTS

4	Exact permuted pattern matching algorithms	22
4.1	Algorithm based on the generalized suffix array	23
4.2	Algorithm based on the AC automaton	25
4.3	Indexing structures for multi-track strings	27
4.3.1	Multi-track suffix trees	27
4.3.2	Multi-track position heaps	33
4.3.3	Contracted multi-track position heaps	42
5	Approximate permuted pattern matching algorithm	46
5.1	Filtering multi-set trees	47
5.2	Selection of hash function	50
6	Experiments	51
6.1	Exact permuted pattern matching algorithms	51
6.2	Indexing structures	55
6.3	Approximate permuted pattern matching algorithms	59
6.3.1	Construction time on random data	60
6.3.2	Search time on random data	60
6.3.3	Construction time and search time on traffic data	61
6.4	Comparison of search times of indexing structures and FILM tree	62
7	Conclusion	68
	References	74
	List of Publications	75

List of Figures

1.1	Examples of multiple sequence data	2
2.1	Spectral Bloom filters $SBF_H(Q_1)$ and $SBF_H(Q_2)$ using $H = \{h_1, h_2\}$ for $Q_1 = \{a, a, b, c, c\}$ and $Q_2 = \{a, b, b, b, c\}$, respectively.	12
4.1	The generalized suffix array for all tracks of a text $\mathbb{T} = (\text{ababaab}\$1, \text{aaababa}\$2, \text{babaaab}\$3)$ and a pattern $\mathbb{P} = (\text{aba}\$4, \text{baa}\$5, \text{aba}\$6)$	24
4.2	The AC automaton of a multi-track $\mathbb{P} = (p_1, p_2, p_3) = (\text{aba}, \text{baa}, \text{aba})$. The broken lines denote transitions of the failure function.	26
4.3	The multi-track suffix tree $MTST(\mathbb{T})$ of a multi-track $\mathbb{T} = (\text{ababaab}\$1,$ $\text{aaababa}\$2, \text{babaaab}\$3)$	28
4.4	The column concatenated strings $s_i = CS_{Sort(\mathbb{T}[i:])}$ in the left, $MTPH(\mathbb{T})$ in the middle, and $CMTPH(\mathbb{T})$ in the right for $\mathbb{T} = (\text{aabbaabbbaaabbbaa}, \text{ababababbababba})$. Maximal-reach pointers $mrp(v)$ in MTPH and CMTPH are drawn as bro- ken lines, where they are omitted if $mrp(v) = v$ for clarity. In indexing nodes, its associated positions (either one or two) are written.	34
5.1	Example of a FILM tree, using only one hash function ($k = 1$).	47
6.1	Comparison of the construction time on random data.	65
6.2	Comparison of the search time on random data.	66

LIST OF FIGURES

6.3	Comparisons of construction time and search time on the real traffic data.	
	The size ω of SBF varied from 10000 to 100000, and the pattern length	
	$m = \mathbb{P} _{len}$ was changed to 36, 72, 144, and 288.	67

List of Tables

1.1	Data structures for the permuted pattern matching	6
1.2	Comparisons of algorithms using data structures for permuted-pattern match- ings	6
6.1	Running times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$	53
6.2	Running times (sec) for N with $n = 100000$, $m = 10$, and $M = N$	53
6.3	Running times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$. . .	54
6.4	Running times (sec) for M with $n = 100000$, $N = 1000$, and $m = 10$	54
6.5	Running times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$	56
6.6	Required memory (MBytes) for n with $N = 1000$, $m = 10$, and $M = 1000$.	56
6.7	Running times (sec) for N with $n = 100000$, $m = 10$, and $M = N$	57
6.8	Required memory (MBytes) for N with $n = 100000$, $m = 10$, and $M = N$.	57
6.9	Running times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$. .	58
6.10	Required memory (MBytes) for m with $n = 100000$, $N = 1000$, and $M =$ 1000	58
6.11	Search times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$	63
6.12	Search times (sec) for N with $n = 100000$, $m = 10$, and $M = N$	64
6.13	Search times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$	64

Chapter 1

Introduction

1.1 Background

Time-series data are observed and stored in various areas; thus, it is necessary to analyze these data, e.g., stock data, weather data, vehicle sensor data, network traffic data, music data, and so on. We may obtain some benefits by studying these data, e.g., discovering useful knowledge or predicting and detecting the characteristic patterns of future data. However, the amount of data is continuing to increase; therefore, methods for the analysis these data are required to realize a real-time property and efficiency.

Recently, it seems that time-series data stored as multiple sequences have increased, and such data may provide advanced knowledge by considering the effect of the combination of sequences (see Figure 1.1). For example, the analysis of multi-sensor data enables advanced event detection. Modern smartphones are equipped with multi-axis acceleration sensors. By focusing on the combination of measurements from three sensors, the smartphone detects its state: stopping, moving, rolling, shaking, and so on. In contrast, it is quite difficult to detect these states by analyzing only one sensor. Modern cars also have many sensors to support drivers. By using sensor data, the cars provide useful functionality to drivers, such as collision detection, parking support, and car navigation. Moreover, the analysis of the action histories of users of various services is useful for identifying

1.1 Background

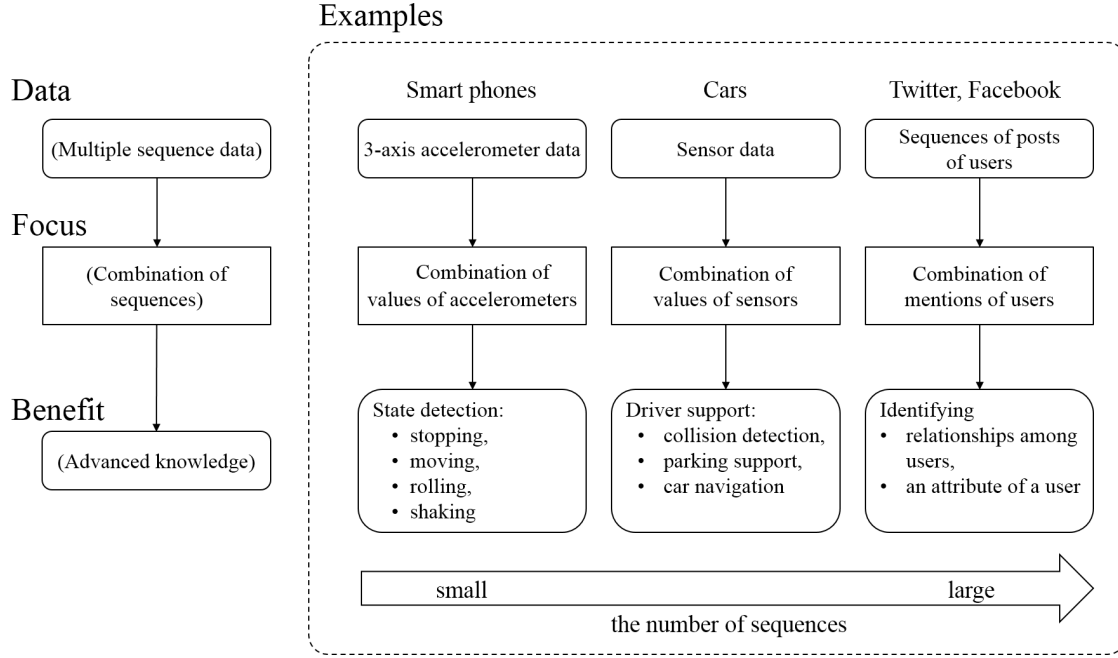


Figure 1.1: Examples of multiple sequence data

the relationships among users or clustering users. For example, the sequences of posts of users on micro-blog services such as Twitter and Facebook can be regarded as multiple time-series data. We can reveal the relationships among users by focusing on the mentions between users. Although the number of sequences in multi-sensor data of a smartphone or car is at most a few dozen or hundreds, the number of users of the micro-blog services can be greater than ten thousand. To analyze such a large set of sequences considering the combination of sequences, scalable methods are required.

String processing is the most fundamental technique for data processing because all data are electronically stored as binary strings on computers. Because numerical data can be transformed into strings by various methods [18, 25, 35], string processing can be widely applied to sequence data. Because of the large number of applications, string processing has been studied for a long time [16, 24], including pattern matching, text indexing, text compression, and so on. The efficiencies of the algorithms for string processing are guaranteed theoretically.

The *pattern matching problem* is the most basic problem in string processing, where a

1.1 Background

text string t and a pattern string p over an alphabet Σ are given to find the occurrences of p in t ; thus it has been studied by many researchers [1, 9, 31, 36, 43]. In addition to the above standard pattern matching problem, there are some varieties of pattern matching for industrial applications. For example, parameterized pattern matching [4, 41] focuses on the structure of strings and is applied to software maintenance or genome analysis. Two-dimensional pattern matching [2, 5, 8, 23] is used for image processing.

Our purpose is to develop a basic analysis technique for multiple time-series data that is theoretically guaranteed to have a real-time property and scalability. We call multi-sets of strings of the same length over Σ a *multi-track string*, such that $\mathbb{T} = \{t_1, t_2, \dots, t_N\}$, where $t_i \in \Sigma^n$ for $1 \leq i \leq N$. In our study, we use the multi-track string as a model of multiple time-series data. We aim to build an analysis method for multiple time-series data by developing efficient algorithms for the multi-track string.

In this thesis, we particularly address the development of algorithms that solve the pattern matching problem on the multi-track string because the pattern matching algorithm is considered as to be the most fundamental tool, similar to standard string processing. In such a problem, text and pattern are given as multiple strings that are different from the standard pattern matching problem. The two-dimensional pattern matching problem also considers multiple-string inputs. It regards a multiple-string input as a tuple of strings; however, our problem is concerned with multi-sets of strings. This is a new formulation of the pattern matching problem that has not been studied in previous works. We must consider a permutation of strings to determine whether or not a pattern matches at a specific position in the text because both of the text and pattern are both multi-sets of strings. Thus, we call our problem the *permuted pattern matching problem* on the multi-track string.

We also address how to perform pattern matching of multiple numerical sequences directly. We call such sequences *multi-track numerical sequences* in this study. Although we mentioned that numerical data can be transformed into strings above, there are problems

1.2 Previous work

associated with the cost of the transformation and the lack of information. Generally, the matching of numerical sequences is performed approximately by using the proper similarity or dissimilarity between sequences. In this dissertation, we define a distance between multi-track numerical sequences. We propose the *approximate permuted pattern matching problem* by using the distance, and try to develop algorithms for solving it efficiently.

1.2 Previous work

The string matching problem is that given a text string t of length n and a pattern string p of length m , outputs all positions of occurrences of the pattern in the text. KMP algorithm [31], BM algorithm [9], and AC algorithm [1] are well known as the sequential search approaches. If preprocessing of the text is allowed, the problem can be solved efficiently by building the indexing structures for the text. The classical indexing structures for the text, suffix trees [43] and suffix arrays [36], require $O(n)$ space and can be built in $O(n)$ time on a constant-size alphabet [22, 28, 32, 37, 38, 42]. By using suffix trees and suffix arrays, all occurrences of a pattern can be reported in $O(m + occ)$ and $O(m \log n + occ)$ time, respectively, where occ is the total number of occurrences of the pattern in the text. Ehrenfeucht et al. [21] proposed more space efficient indexing structure called position heaps, which requires $O(n)$ space but the number of nodes in the position heaps is at most $n + 1$ although that of the suffix tree is at most $2n - 1$. Kucherov [33] showed an Ukkonen-like on-line $O(n)$ -time algorithm for constructing position heaps. By using position heaps, the occurrences of the pattern can be found in $O(m^2 + occ)$ time. To improve its time bound to $O(m + occ)$, Ehrenfeucht et al. [21] proposed $O(n)$ -space auxiliary structure, called the maximal-reach pointers (shortly MRPs).

1.3 Contributions

The contribution of this paper is as follows: We define the permuted pattern matching between multi-track strings which are multi-sets of strings, and propose the permuted pattern matching problem. We consider three settings of the problem :

1. neither of the text and pattern are allowed to be preprocessed,
2. only the pattern can be preprocessed,
3. only the text can be preprocessed.

For each setting, we propose efficient algorithms:

1. we propose an algorithm by using the generalized suffix array that runs in $O(nN)$ time and space for integer alphabets,
2. we propose an algorithm by using AC-automaton that runs in $O(nN \log |\Sigma|)$ time and $O(mM + N)$ space for general alphabets,
3. we propose a new indexing structure *multi-track suffix tree* that enables us to solve the permuted pattern matching problem in $O(mN \log |\Sigma| + occ)$ time and $O(nN)$ space when assuming $N = M$ and general alphabets. We also propose a memory-efficient structure *contracted multi-track position heap*, and show an algorithm that runs in $O(m^2 N^2 \log |\Sigma| + occ)$ time and $O(n)$ space by using it.

We also address an approximate version of permuted pattern matchings. We consider multi-track sequences over \mathcal{R} and define a distance D between such two multi-track numerical sequences. Then, the approximate permuted pattern matching problem is that given a multi-track text $\mathbb{T} = \{t_1, \dots, t_N\}$ of length n , a multi-track pattern $\mathbb{P} = \{p_1, \dots, p_M\}$ of length m , and a criterion $\delta \geq 0$, outputs all positions i such that $D(\mathbb{T}[i : i + m - 1], \mathbb{P}) \leq \delta$ for $1 \leq i \leq n$. For this problem, we propose a new data structure called a filtering multi-set tree (shortly FILM tree) that each node represents a spectral Bloom filter. By

1.4 Organization of the thesis

Table 1.1: Data structures for the permuted pattern matching

data structure	space	# of nodes	construction	search
GSA	$O(nN)$		$O(nN)$	$O(nN)$
AC automaton	$O(mM)$	$mM + 1$	$O(mM \log \Sigma)$	$O(nN \log \Sigma)$
MTST	$O(nN)$	$2n - 1$	$O(nN \log \Sigma)$	$O(mN \log \Sigma + occ)$
MTPH	$O(nN)$	$nN + 1$	$O(nN \log \Sigma)$	$O(m^2 N \log \Sigma + occ)$
CMPH	$O(n)$	$n + 1$	$O(nN \log \Sigma)$	$O(m^2 N^2 \log \Sigma + occ)$
FILM tree	$O(n\omega)$	$2^{\lceil \log_2 n \rceil + 1} - 1$	$O(knN + n\omega)$	$O(kmM + n\omega)$

Table 1.2: Comparisons of algorithms using data structures for permuted-pattern matchings

Data structure	Permutation type				Constructed for
	String		Numerical		
	sub	full	sub	full	
GSA	✓	✓			text and pattern
AC automaton	✓	✓			pattern
MTST		✓			text
MTPH		✓			text
CMPH		✓			text
FILM tree	✓	✓	✓	✓	text

using the FILM tree, both exact and approximate matching problems can be solved in $O(kmM + n\omega)$ time and space, where k is the number of hash functions for constructing the spectral Bloom filter and ω is the size of the spectral Bloom filter. Note that, outputs of search using the FILM tree can contain the false-positive errors.

At last, we demonstrate the performance of our approach experimentally.

The contributions of this paper is summarized in Table 1.1 and Table 1.2.

1.4 Organization of the thesis

The rest of this thesis is organized as follows.

In Chapter 2, we present some notations and results in previous works that we will use in the paper.

In Chapter 3, we propose the multi-track string and the permuted pattern matching

1.4 Organization of the thesis

problem on multi-track strings. We also define the numerical version of the multi-track and the Euclidean distance between two numerical multi-track sequences. Then, we show the definition of the approximate permuted pattern matching problem.

In Chapter 4, we consider three settings of the exact permuted pattern matching. First is that both of the multi-track text and pattern can not be preprocessed. We propose an algorithm using the generalized suffix array for a text and a pattern, which runs in $O(nN)$ time for integer alphabets. Second is that it is allowed to preprocess the multi-track pattern. For this setting, we propose an algorithm based on the Aho-Corasick (AC) automaton, and show that it solves the problem in $O(nN \log |\Sigma|)$ time for general alphabets. Third is that it is allowed to preprocess the multi-track text. We propose the three indexing structures for the multi-track string, that are, the multi-track suffix tree, the multi-track position heap and the contracted multi-track position heap. We show that each structure can be constructed in $O(nN \log |\Sigma|)$ time for general alphabets and the full-permuted pattern matching can be solved efficiently by using these structures.

In Chapter 5, we propose the filtering multi-set tree for the approximate permuted pattern matching.

In Chapter 6, we show the results of the experiments for our algorithms.

Chapter 2

Preliminaries

2.1 Strings

Let Σ be a finite set of symbols, called an *alphabet*. An element of Σ^* is called a *string*. Let Σ^n be the set of strings of length n . For a string $w = xyz$, strings x , y , and z are called *prefix*, *substring*, and *suffix* of w , respectively. $|w|$ denotes the length of w , and $w[i]$ denotes the i -th symbol of w for $1 \leq i \leq |w|$. Let $x \cdot y$, briefly denote xy , be the *concatenation* of strings x and y . Then, $w = w[1]w[2] \dots w[|w|]$. The substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$, i.e., $w[i : j] = w[i] w[i + 1] \dots w[j]$. Moreover, let $w[: i] = w[1 : i]$ and $w[i :] = w[i : |w|]$ for $1 \leq i \leq |w|$. The empty string is denoted by ε , that is, $|\varepsilon| = 0$. For convenience, let $w[i : j] = \varepsilon$ if $i > j$. For two strings x and y , we denote by $x \prec y$ if x is lexicographically smaller than y , and by $x \preceq y$ if $x \prec y$ or $x = y$. For a set S , we denote the cardinality of S by $|S|$. For a multi-set S , let $\#_S(x)$ denote the multiplicity of element $x \in S$.

2.2 Tries

Definition 2.1 (Tries). A trie on Σ is a rooted tree that has the following two properties:

(1) each edge is labeled by a character $c \in \Sigma$, and (2) for each node u and a character

2.3 Sequence hash trees

$c \in \Sigma$, u has at most one edge that is labeled by c from u to a child of u .

Let $T = (V, E)$ be a trie, where V and E are sets of nodes and edges, respectively. The root node of T is denoted by $root$. Each edge $e \in E$ is denoted by (u, c, v) , where $c \in \Sigma$ is the label of e , and v is a child node of a node u . Note that, we assume that the time required to find the child of a node on the child edge labeled by $c \in \Sigma$ is $O(\log |\Sigma|)$ in this paper. For any node v in T , the sets of ancestors and descendants of v are denoted by $Anc(v)$ and $Des(v)$, respectively. For nodes u and $v \in Des(u)$, the sequence of nodes and edges from u to v is called the *path* from u to v and denoted by $path(u, v)$. For any node v , let a path $path(root, v)$ be $root, e_1, v_1, e_2, v_2, \dots, e_d, v$ and let e_i be labeled by $c_i \in \Sigma$ for $i = 1, 2, \dots, d$. Then, we say that the string $w = c_1 c_2 \dots c_d$ is *represented in T* , and denote the node v by \bar{w} and the string w by $label(v)$. Thus, $root = \bar{\varepsilon}$ and $label(root) = \varepsilon$. The number of edges d on $path(root, v)$ is called the *depth* of v , denoted by $depth(v)$.

2.3 Sequence hash trees

A *sequence hash tree* [14] is a trie for hashing a set of strings.

Definition 2.2 (Sequence hash trees [14]). *Let $W = \{w_1, w_2, \dots, w_k\}$ be an ordered set of strings, where $w_i \in \Sigma^*$. For $1 \leq i \leq k$, $SHT_i(W) = (V_i, E_i)$ is a trie recursively defined by $(V_0, E_0) = (\{root\}, \emptyset)$, and*

$$SHT_i(W) = (V_{i-1} \cup \{\bar{q_i}\}, E_{i-1} \cup \{(\overline{q_i[:|q_i| - 1]}, c, \bar{q_i})\}),$$

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

For any i , $SHT_i(W)$ is obtained by adding at most one node and one edge. Thus, $SHT(W) = SHT_k(W)$ consumes $O(k)$ space. $SHT_i(W)$ is obtained by adding a node

2.4 Various queries on rooted trees

corresponding to w_i into $SHT_{i-1}(W)$. When the node corresponding to w_i is added into $SHT_{i-1}(W)$, we say that w_i is *inserted to* $SHT_{i-1}(W)$.

2.4 Various queries on rooted trees

We show some results for the rooted tree T of n nodes and σ degree that are used in the sequel of this thesis. We first describe the *lowest common ancestor query* problem that is one of most fundamental problems on trees. For any two nodes u and v in T , the lowest common ancestor $LCA(u, v)$ is the ancestor of u and v and the farthest node from the root. The following result is known:

Lemma 2.3 (Lowest common ancestor query [6, 40]). *For any given two nodes u and v , the lowest common ancestor $LCA(u, v)$ of u and v in T can be answered in $O(1)$ time, after an $O(n)$ time and space preprocessing of T .*

We next explain about the *nearest marked ancestor query*. The semi-dynamic nearest marked ancestor query problem is the on-line problem of satisfying the following queries on a rooted tree T : (1) insert a leaf v into T ; (2) mark a node v in T ; (3) find the nearest marked ancestor $NMA(v)$ of a node v in T . It was shown that this problem can be solved efficiently.

Lemma 2.4 (Nearest marked ancestor query [3, 44]). *The semi-dynamic nearest marked ancestor problem can be solved in the following time bounds after an $O(n \log \sigma)$ time and $O(n)$ space preprocessing of T : (1) inserting a leaf v in amortized $O(1)$ time; (2) marking a node v in worst-case $O(1)$ time; (3) finding the nearest marked ancestor $NMA(v)$ of a node v in worst-case $O(1)$ time.*

In addition to above two queries, we will use the following query.

Lemma 2.5 (Level ancestor query [7]). *For any given node v and an integer $d \geq 0$, the ancestor $LevA(v, d)$ of v at depth d can be answered in $O(1)$ time, after an $O(n)$ time and space preprocessing of T .*

2.5 Indexing structures for strings

In the sequel of this thesis, we will propose algorithms and data structures that are based on indexing structures for standard strings. In front of our proposals, we describe such data structures in this section.

Definition 2.6 (Suffix trees [43]). *For a text string t of length n over an alphabet Σ , a suffix tree $ST(t)$ of t is a compacted trie for all suffixes of t .*

From the definition, $ST(t)$ has just n leaves and each leaf represents one suffix of t . Since each internal node has at least two children, the number of nodes in $ST(t)$ is at most $2n - 1$, and thus $ST(t)$ requires $O(n)$ space. It is known that $ST(t)$ can be constructed in linear time for the length of t : $O(n)$ time for integer alphabets [22] and $O(n \log |\Sigma|)$ time for general alphabets [37, 42]. By using $ST(t)$, the pattern matching on t can be solved in $O(m \log |\Sigma| + occ)$ time, where m is the length of a pattern.

Definition 2.7 (Suffix arrays [36]). *For a text string t of length n over an alphabet Σ , a suffix array $SA(t)$ of t is an array such that if $SA(t)[i] = j$ then $t[j:]$ is the i -th smallest suffix in lexicographical order for $1 \leq i, j \leq n$.*

$SA(t)$ provides an order of suffixes that are sorted lexicographically. Obviously, $SA(t)$ needs $O(n)$ space. $SA(t)$ can be also constructed in linear time: $O(n)$ time for integer alphabets [22, 28, 32, 38] and $O(n \log n)$ time for general alphabets [11, 34, 36]. By using $SA(t)$, the pattern matching on t is performed in $O(m \log n + occ)$ time with a binary search on $SA(t)$. This time bound can be improved by using a longest common prefix (LCP) array defined as follows:

Definition 2.8 (LCP arrays). *For a text string t of length n over an alphabet Σ , an LCP array $LCP(t)$ of t is an array such that $LCP(t)[0] = -1$ and $LCP(t)[i]$ is the length of the longest common prefix between $t[SA[i-1]:]$ and $t[SA[i]:]$ for $1 < i \leq n$.*

By using $SA(t)$ and $LCP(t)$, the pattern matching is done in $O(m + \log n + occ)$ time.

2.6 Spectral Bloom filter

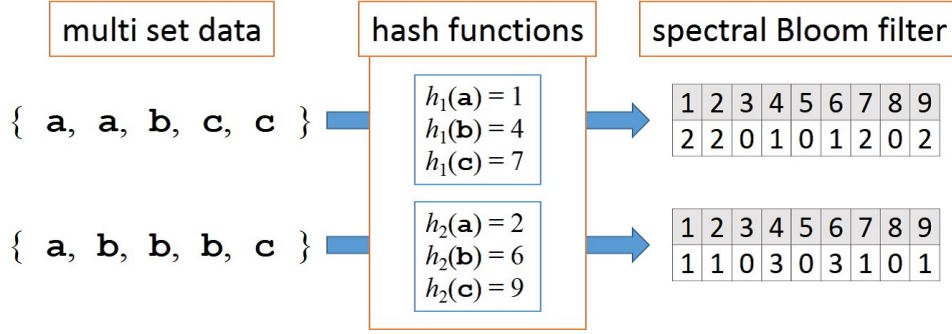


Figure 2.1: Spectral Bloom filters $SBF_H(Q_1)$ and $SBF_H(Q_2)$ using $H = \{h_1, h_2\}$ for $Q_1 = \{a, a, b, c, c\}$ and $Q_2 = \{a, b, b, b, c\}$, respectively.

Definition 2.9 (Position heaps [21, 33]). *For a text string t of length n over an alphabet Σ , let S be an ordered set $t[1:], t[2:], \dots, t[n:]$. A position heap $PH(t)$ is a sequence hash tree for S , i.e., $SHT(S)$.*

The position heap was recently proposed by Ehrenfeucht et al. [21]. It is a trie structure and each node corresponds to each suffix. Thus, the position heap has just $n+1$ nodes with assuming that the root node does not correspond to any suffix. Kucherov proposed the on-line linear-time construction algorithm of the position heap [33] that is based on Ukkonen's suffix tree construction algorithm [42]. The naive pattern matching by using $PH(t)$ requires $O(m^2 \log |\Sigma| + occ)$ time. Ehrenfeucht et al. proposed an $O(n)$ -space data structure maximal reach pointer, and developed an efficient $O(m \log |\Sigma| + occ)$ -time matching algorithm by using it.

2.6 Spectral Bloom filter

A spectral Bloom filter [15] is a data structure that estimates the multiplicity of an element in a multi-set. The spectral Bloom filter for a multi-set Q is useful to answer the query of whether $R \subseteq Q$ for any given multi-set R .

Definition 2.10 (SBF [15]). *Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of k hash functions, $h_i : U \rightarrow \{0, \dots, u-1\}$, where the domain U is either Σ^* or \mathcal{R}^* . Let Q be a multi-set*

2.6 Spectral Bloom filter

over U . A spectral Bloom filter $SBF_H(Q)$ for Q using H is an integer array of length ω , defined by $SBF_H(Q) = (C_Q[1], C_Q[2], \dots, C_Q[\omega])$ such that

$$C_Q[i] = \sum_{h \in H} \sum_{q \in Q} \llbracket h(q) \pmod{\omega} + 1 = i \rrbracket,$$

where $\llbracket P \rrbracket$ is 1 if the predicate P is true and 0 otherwise.

For example, Fig. 2.1 shows two SBFs using two hash functions. The multiplicity of an element $x \in U$ in Q can be estimated by \min_x that is given as following:

$$\min_x = \min\{C_Q[h_1(x) \pmod{\omega} + 1], C_Q[h_2(x) \pmod{\omega} + 1], \dots, C_Q[h_k(x) \pmod{\omega} + 1]\}.$$

Note that there may be *false positive error* for the estimation by using \min_x because of the collision between hash values.

For two SBFs, we define the following operations.

Definition 2.11. For two SBFs, $SBF_H(Q_1)$ and $SBF_H(Q_2)$ of the same size ω , we define their addition and subtraction by

$$\begin{aligned} SBF_H(Q_1) \oplus SBF_H(Q_2) &= (C_{Q_1}[1] + C_{Q_2}[1], C_{Q_1}[2] + C_{Q_2}[2], \dots, C_{Q_1}[\omega] + C_{Q_2}[\omega]), \\ SBF_H(Q_1) \ominus SBF_H(Q_2) &= (C_{Q_1}[1] - C_{Q_2}[1], C_{Q_1}[2] - C_{Q_2}[2], \dots, C_{Q_1}[\omega] - C_{Q_2}[\omega]). \end{aligned}$$

We can easily verify the following properties.

Property 2.1. For any two multi-sets Q and R ,

- (1) $SBF_H(Q) \oplus SBF_H(R) = SBF_H(Q \cup R)$,
- (2) if $R \subseteq Q$, then $\min(SBF_H(Q) \ominus SBF_H(R)) \geq 0$.

We note that the converse of (2) in Property 2.1 does not always hold. For instance, let us consider $Q = \{\mathbf{a}, \mathbf{b}\}$, $R = \{\mathbf{c}\}$ and $H = \{h_1, h_2\}$ such that $h_1(\mathbf{a}) = 1$, $h_2(\mathbf{a}) = 2$,

2.6 Spectral Bloom filter

$h_1(\mathbf{b}) = 3$, $h_2(\mathbf{b}) = 4$, $h_1(\mathbf{c}) = 1$, $h_2(\mathbf{c}) = 3$. Then we have $SBF_H(Q) = (1, 1, 1, 1)$ and $SBF_H(R) = (1, 0, 1, 0)$, which yield that $\min(SBF_H(Q) \ominus SBF_H(R)) = 0$, although $R \not\subseteq Q$. Therefore, if we use the condition $\min(SBF_H(Q) \ominus SBF_H(Q')) \geq 0$ to reply a subset query $Q' \subseteq Q$, we may encounter a false positive.

We now show the probability of the false positive error of $SBF_H(Q)$. We assume that every hash function $h_j \in H$ outputs each value with equal probability, and Q contains N elements. Let f_x be the number of occurrences of $x \in U$ in Q , and $\min_x = \min\{C_Q[h_1(x)(\bmod \omega) + 1], C_Q[h_2(x)(\bmod \omega) + 1], \dots, C_Q[h_k(x)(\bmod \omega) + 1]\}$. Then, $f_x \leq \min_x$ holds.

First we show the probability that the estimation of the multiplicity of an element $x \in U$ in Q is incorrect. This is encountered when $f_x \neq \min_x$. It is shown in [15] that the probability E_x of $f_x \neq \min_x$ is same to that of the false positive error of the Bloom filter and obtained as follows:

$$E_x \simeq (1 - e^{-kN/\omega})^k. \quad (2.1)$$

For $k = \frac{\omega}{N} \ln 2$, the formula (2.1) is minimized as

$$E_{\min} = 2^{-k} \simeq 0.6185^{\frac{\omega}{N}}.$$

For example, if we take $\omega = 8N$, the probability that SBF replies a correct answer is 98%.

We next consider the probability E_{sbf} that the estimation of $R \subseteq Q$ by using $SBF_H(Q)$ for given a multi-set R is incorrect. It may occur when $f_x \neq \min_x$ holds for all $x \in R$. Let R' be a set such that $x \in R'$ for any $x \in R$ and $x \notin R'$ for any $x \notin R$. Then, E_{sbf} is given as follows by using the formula (2.1):

$$E_{sbf} = \prod_{x \in R'} E_x$$

2.7 Hash functions

2.7.1 Rolling hash

The rolling hash is a classical hash function found in Karp-Rabin string matching algorithm [30].

Definition 2.12. A rolling hash $h_{rh}(w)$ for a string w of length l is

$$h_{rh}(w) = w[1]a^{l-1} + w[2]a^{l-2} + \cdots + w[l]a^0 \pmod{q},$$

where a and q are positive integers.

In practice, it is strongly preferred that $1 < a < q$ and a, q are mutually prime numbers to reduce hash collisions.

2.7.2 Locality sensitive hashing

The locality sensitive hashing (LSH) is a hashing algorithm for the nearest neighbor search problem [10, 13, 17, 27]. The hash values of a hash function in a LSH family are in a collision with high probability if input data are similar. Different LSH families can be used for different distance functions. In this paper, we use the following hash function, that is one of the standard LSH families for Euclidean metric space.

Definition 2.13. We define a locality sensitive hashing function $h_{lsh} : \mathcal{R}^m \rightarrow \mathcal{N}$ by

$$h_{lsh}(\mathbf{v}) = \left\lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{r} \right\rfloor,$$

where each entry of $\mathbf{a} \in \mathcal{R}^m$ is selected independently from a stable distribution, r is a positive real number, and b is selected uniformly from the range $(0, r]$.

Chapter 3

Multi-tracks

As was mentioned in the introductory chapter, our pattern matching problem is defined over a multi-set of strings. For ease of presentation, however, in what follows we assume an arbitrary order of the strings in the multi-set and define a multi-track string as a tuple. Then, we define the problems that we address in this thesis formally.

3.1 Multi-track strings

Let Σ_N be the set of all N -tuples (a_1, a_2, \dots, a_N) with $a_i \in \Sigma$ for $1 \leq i \leq N$, called a *multi-track alphabet*. An element of Σ_N is called a *multi-track character* (mt-character), and an element of Σ_N^* is called a *multi-track string* (or simply *multi-track*), where the concatenation between two multi-track strings is defined as $(a_1, a_2, \dots, a_N) \cdot (b_1, b_2, \dots, b_N) = (a_1b_1, a_2b_2, \dots, a_Nb_N)$. For a multi-track $\mathbb{T} = (t_1, t_2, \dots, t_N) \in \Sigma_N^n$, the i -th element t_i of \mathbb{T} is called the i -th *track*, the length of multi-track \mathbb{T} is denoted by $|\mathbb{T}|_{len} = |t_1| = |t_2| = \dots = |t_N| = n$, and the number of tracks in multi-track \mathbb{T} or the *track count* of \mathbb{T} , is denoted by $|\mathbb{T}|_{num} = N$. Let Σ_N^+ be the set of all multi-tracks of length at least 1, and let $\mathbb{E}_N = (\varepsilon, \varepsilon, \dots, \varepsilon)$ denotes the *empty multi-track of track count N* . Then, $\Sigma_N^* = \Sigma_N^+ \cup \{\mathbb{E}_N\}$. For a multi-track $\mathbb{T} = \mathbb{X}\mathbb{Y}\mathbb{Z}$, multi-track \mathbb{X} , \mathbb{Y} , and \mathbb{Z} are called *prefix*, *substring*, and *suffix* of \mathbb{T} , respectively. We call a prefix, substring and suffix of a

3.1 Multi-track strings

multi-track by an *mt-prefix*, *mt-substring* and *mt-suffix* of the multi-track, respectively. $\mathbb{T}[i]$ denotes the i -th mt-character of \mathbb{T} for $1 \leq i \leq |\mathbb{T}|_{len}$, i.e., $\mathbb{T} = \mathbb{T}[1]\mathbb{T}[2] \dots \mathbb{T}[|\mathbb{T}|_{len}]$. The mt-substring of \mathbb{T} that begins at position i and ends at position j is denoted by $\mathbb{T}[i : j] = (t_1[i : j], t_2[i : j], \dots, t_N[i : j])$ for $1 \leq i \leq j \leq |\mathbb{T}|_{len}$. Moreover, let $\mathbb{T}[:i] = \mathbb{T}[1 : i]$ and $\mathbb{T}[i:] = \mathbb{T}[i : |\mathbb{T}|_{len}]$, respectively.

Definition 3.1 (Permuted multi-track). *Let $\mathbb{X} = (x_1, x_2, \dots, x_N)$ be a multi-track of track count N . Let $\mathbf{r} = (r_1, r_2, \dots, r_K)$ be a sub-permutation of $(1, \dots, N)$, where $1 \leq K \leq N$. A permuted multi-track of \mathbb{X} specified by \mathbf{r} is a multi-track $(x_{r_1}, x_{r_2}, \dots, x_{r_K})$, denoted by either $\mathbb{X}\langle r_1, r_2, \dots, r_K \rangle$ or $\mathbb{X}\langle \mathbf{r} \rangle$. If $K = N$, \mathbf{r} is called a full-permutation and $\mathbb{X}\langle \mathbf{r} \rangle$ is called a full-permuted multi-track of \mathbb{X} .*

Definition 3.2 (Permuted-match). *For any multi-tracks $\mathbb{X} = (x_1, x_2, \dots, x_{|\mathbb{X}|_{num}})$ and $\mathbb{Y} = (y_1, y_2, \dots, y_{|\mathbb{Y}|_{num}})$ with $|\mathbb{X}|_{len} = |\mathbb{Y}|_{len}$ and $|\mathbb{X}|_{num} \leq |\mathbb{Y}|_{num}$, we say that \mathbb{X} permuted-matches \mathbb{Y} , denoted by $\mathbb{X} \sqsubseteq \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}'$ for some permuted multi-track \mathbb{Y}' of \mathbb{Y} . In particular, if $|\mathbb{X}|_{num} = |\mathbb{Y}|_{num}$, then we say that \mathbb{X} full-permuted-matches \mathbb{Y} , and denote it by $\mathbb{X} \sqsubseteq \mathbb{Y}$. Otherwise, i.e., if $|\mathbb{X}|_{num} < |\mathbb{Y}|_{num}$, then we say that \mathbb{X} sub-permuted-matches \mathbb{Y} .*

The problem we consider is formally defined as:

Problem 1 (Permuted pattern matching). *Given multi-track text $\mathbb{T} = (t_1, \dots, t_N)$ where $|\mathbb{T}|_{len} = n$, and multi-track pattern $\mathbb{P} = (p_1, \dots, p_M)$, where $M \leq N$ and $|\mathbb{P}|_{len} = m \leq n$, output all positions i that satisfy $\mathbb{P} \sqsubseteq \mathbb{T}[i : i + m - 1]$. If $|\mathbb{T}|_{num} = |\mathbb{P}|_{num}$, then the problem is called the full-permuted pattern matching problem. Otherwise, i.e., if $|\mathbb{T}|_{num} > |\mathbb{P}|_{num}$, it is called the sub-permuted pattern matching problem.*

To determine whether $\mathbb{X} \sqsubseteq \mathbb{Y}$ or not for two multi-tracks \mathbb{X} and \mathbb{Y} , we can use $Sort(\mathbb{X})$ and $Sort(\mathbb{Y})$ defined as follows:

Definition 3.3. *For a multi-track $\mathbb{X} = (x_1, x_2, \dots, x_N)$, let $SortIndex(\mathbb{X}) = (r_1, r_2, \dots, r_N)$*

3.1 Multi-track strings

be a full-permutation such that $x_{r_i} \preceq x_{r_j}$ for any $1 \leq i \leq j \leq N$. $\text{Sort}(\mathbb{X})$ is defined as $\mathbb{X} \langle \text{SortIndex}(\mathbb{X}) \rangle$.

It holds that $\mathbb{X} \cong \mathbb{Y}$ if and only if $\text{Sort}(\mathbb{X}) = \text{Sort}(\mathbb{Y})$. We show examples of permuted matching in Example 1.

Example 1. Consider multi-tracks $\mathbb{T} = (t_1, t_2, t_3) = (\text{abab}, \text{abbb}, \text{abba})$, $\mathbb{X} = (x_1, x_2, x_3) = (\text{abba}, \text{abab}, \text{abbb})$, and $\mathbb{Y} = (y_1, y_2) = (\text{ba}, \text{ab})$. $\mathbb{X} \cong \mathbb{T}$ holds because $\text{Sort}(\mathbb{T}) = \text{Sort}(\mathbb{X}) = (\text{abab}, \text{abba}, \text{abbb})$, where $\text{SortIndex}(\mathbb{T}) = (1, 3, 2)$ and $\text{SortIndex}(\mathbb{X}) = (2, 1, 3)$. Moreover, $\mathbb{Y} \sqsubseteq^{\infty} \mathbb{T}[3 : 4]$ holds because $\text{Sort}(\mathbb{T}[3 : 4] \langle (1, 3) \rangle) = \text{Sort}(\mathbb{Y}) = (\text{ab}, \text{ba})$, where $\text{SortIndex}(\mathbb{T}[3 : 4] \langle (1, 3) \rangle) = (1, 2)$ and $\text{SortIndex}(\mathbb{Y}) = (2, 1)$.

All $\text{SortIndex}(\mathbb{T}[i :])$ for $1 \leq i \leq n$ can be computed in $O(nN)$ time since the following lemma holds.

Lemma 3.4. Given a multi-track $\mathbb{T} = (t_1, t_2, \dots, t_N)$ of length n , the permutations $\text{SortIndex}(\mathbb{T}[i :])$ for all $(1 \leq i \leq n)$ can be computed in $O(nN)$ time for an integer alphabet, or in $O(nN \log |\Sigma|)$ time for a general alphabet.

Proof: We can determine the lexicographic order of all the nN suffixes $t_1[1:], \dots, t_N[1:], t_1[2:], \dots, t_N[2:], t_1[n:], \dots, t_N[n:]$ as follows: let s be a string $t_1\$1t_2\$2 \dots t_N\$N$, where each $\$_i$ is smaller than any $c \in \Sigma$ and $\$_N \prec \$_{N-1} \prec \dots \prec \$_2 \prec \$_1$. We construct a suffix tree $ST(s)$ or a suffix array $SA(s)$. For integer alphabets, $ST(s)$ (e.g., [22]) or $SA(s)$ (e.g., [29]) can be constructed in $O(nN)$ time and space, and for general alphabets, $ST(s)$ (e.g., [42]) can be in $O(nN \log |\Sigma|)$ time and $O(nN)$ space.

The permutation $\text{SortIndex}(\mathbb{T}[i :])$ depends on the lexicographic order of suffixes $s[i:], s[(n+1)+i:], s[2(n+1)+i:], \dots, s[(N-1)(n+1)+i:]$. This lexicographic order corresponds to the order in which each suffix is encountered during a depth first traversal of $ST(s)$ or the simple linear scan of $SA(s)$. Such traversal or scan require $O(nN \log |\Sigma|)$ or $O(nN)$ time, respectively. Since all suffixes of s must be encountered in the traversal

3.2 Numerical multi-track sequences

or scan, the permutation $SortIndex(\mathbb{T}[i :])$ for all i 's can be obtained by the one-time traversal or scan. Thus the lemma holds. ■

From Lemma 3.4, Problem 1 can be solved in $O(nmN)$ time and $O(nN)$ space by storing all $SortIndex(\mathbb{T}[i :])$ naively.

3.2 Numerical multi-track sequences

We say that a multi-track $\mathbb{T} = (t_1, \dots, t_N)$ is a *multi-track numerical sequence* or *numerical multi-track* if each entry $t_i[j]$ is a numerical value in \mathcal{R} . In this thesis, we address the approximate matching problem for multi-track numerical sequences, which is defined in a metric space with a distance function D for numerical multi-tracks as follows.

Problem 2 (Approximate permuted pattern matching problem). *Given a numerical multi-track text \mathbb{T} , a numerical multi-track pattern \mathbb{P} , and a criterion $\delta \geq 0$, output all positions i that satisfy $D(\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle, \mathbb{P}) \leq \delta$, where $\mathbf{r} = (r_1, r_2, \dots, r_K)$ is a sub-permutation of $(1, \dots, N)$. If $M < N$, then the problem is called the approximate sub-permuted pattern matching problem, whereas if $M = N$, it is called the approximate full-permuted pattern matching problem.*

If $D(\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle, \mathbb{P}) \leq \delta$ holds, we say $\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle$ *approximate permuted-matches* \mathbb{P} . In Problem 2, we must specify the distance function $D(\mathbb{X}, \mathbb{Y})$ for numerical multi-tracks in order to measure the similarity of \mathbb{X} and \mathbb{Y} . Several distance functions have been proposed for numerical data in previous studies. For example, the dynamic time warping (DTW) [39] is often used for time series data. In this thesis, we focus on a metric based on the Euclidean distance.

Definition 3.5 (Euclidean distance). *For two numerical tracks t_1 and t_2 of the same*

3.2 Numerical multi-track sequences

length n , the Euclidean distance $d(t_1, t_2)$ between t_1 and t_2 is defined by

$$d(t_1, t_2) = \sqrt{\sum_{i=1}^n (t_1[i] - t_2[i])^2}.$$

Next, we define the Euclidean distance for multi-track numerical sequences.

Definition 3.6 (Multi-track Euclidean distance). *For two numerical multi-tracks $\mathbb{X} = (x_1, x_2, \dots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \dots, y_N)$ where the length of x_i and y_i is n , the multi-track Euclidean distance is defined by*

$$D(\mathbb{X}, \mathbb{Y}) = \sum_{j=1}^N d(x_j, y_j).$$

We show an example of the approximate permuted pattern matching in Example 2.

Example 2. *Consider numerical multi-tracks $\mathbb{T} = (t_1, t_2, t_3) = ((5\ 6\ 8\ 4), (3\ 5\ 6\ 8), (2\ 4\ 5\ 7))$ and $\mathbb{P} = (p_1, p_2) = ((4\ 5\ 7), (5\ 6\ 8))$. For \mathbb{T} , \mathbb{P} , and given the criterion $\delta = 2$, \mathbb{P} approximate permuted-matches $\mathbb{T}[1 : 3]$ and $\mathbb{T}[2 : 4]$ because $D(\mathbb{T}[1 : 3]\langle 2, 1 \rangle, \mathbb{P}) = \sqrt{2}$ and $D(\mathbb{T}[2 : 4]\langle \mathbf{r} \rangle, \mathbb{P}) = 0$ hold.*

We present a naive algorithm for permuted pattern matching problems in Algorithm 1. $\Pi_{N,M}$ in Line 3 is the set of permutations $\mathbf{r} = (r_1, r_2, \dots, r_M)$, where $1 \leq r_i \leq N$ and $r_i \neq r_j$ for $1 \leq i, j \leq j$. Thus, $|\Pi_{N,M}| = \frac{N!}{(N-M)!}$. This algorithm computes all of the positions that \mathbb{P} permuted- or approximate permuted-matches $\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle$. The computation in Line 4 of Algorithm 1 requires $O(mM)$ time. This algorithm has two loops, where the outside loop requires $O(n)$ and the inside loop requires $O(\frac{N!}{(N-M)!})$. Thus, Algorithm 1 runs in $O(nmM \frac{N!}{(N-M)!})$ time.

As a slightly more efficient approach, we note an algorithm that reduces the permuted pattern matching problem to the minimum weight bipartite matching problem. For each position i on multi-track text \mathbb{T} , we consider a weighted bipartite graph $G_i = (A \cup B, A \times B)$, where $A = \{1, \dots, N\}$, $B = \{1, \dots, M\}$, and the weight of an edge $(j, k) \in$

3.2 Numerical multi-track sequences

Algorithm 1: Naive algorithm for permuted pattern matching problems

Input: multi-track text \mathbb{T} , multi-track pattern \mathbb{P} (and criterion δ)
Output: matching positions of \mathbb{P} in \mathbb{T}

```

1  $n = |\mathbb{T}|_{len}; N = |\mathbb{T}|_{num}; m = |\mathbb{P}|_{len}; M = |\mathbb{P}|_{num};$ 
2 for  $i = 1$  to  $n - m + 1$  do
3   foreach  $\mathbf{r} \in \Pi_{N,M}$  do
4     if  $\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle$  and  $\mathbb{P}$  (approximate) permuted-match then
5       output position  $i$ ;
6       break
7     end
8   end
9 end

```

$A \times B$ is the distance $d(t_j[i : i + m - 1], p_k[1 : m])$. The minimum weight perfect bipartite matching on G_i corresponds to the distance $D(\mathbb{T}[i : i + m - 1]\langle \mathbf{r} \rangle, \mathbb{P})$ for the best choice of the permutation \mathbf{r} . Because the construction of G_i for each position i requires $O(mNM)$ time, and the minimum weighted bipartite matching for G_i can be found in $O(M(N + M)^2)$ time by the minimum cost flow algorithm [12, 19], the total time of this method is $O(nM(mN + (N + M)^2))$.

In the following chapters, we will describe the algorithms for Problem 1 and Problem 2. In Chapter 4, we address Problem 1, that is, the exact permuted pattern matching problem. We consider three settings of the problem as was mentioned in Chapter 1: (1) no preprocessing, (2) preprocessing for a pattern, and (3) preprocessing for a text. In Section 4.1 and Section 4.2, we show two algorithms that are for the settings of (1) and (2) and based on the suffix array [36] and the Aho-Corasick automaton [1]. Next in Section 4.3, we consider the setting (3). We propose three indexing structures that are based on the suffix tree [43] and the position heap [21]. In Chapter 5, we address Problem 2, that is, the approximate permuted pattern matching problem. We propose a flexible data structure based on the spectral Bloom Filter [15] and show the matching algorithm by using it.

Chapter 4

Exact permuted pattern matching algorithms

In this chapter, we show algorithms for exact full-/sub-permuted pattern matching problems. As mentioned in previous chapters, we consider three settings with respect to the problem: (1) neither of a text and a pattern can be preprocessed, (2) a pattern is allowed to be preprocessed, and (3) a text can be preprocessed. First we will propose an algorithm for the setting (1) that is based on the suffix array [36] in Section 4.1. Given a multi-track text and a multi-track pattern on an integer alphabet, the full-/sub-permuted pattern matching can be performed in $O(nN)$ time and space by using this algorithm. Next we show an algorithm based on the AC automaton [1] for the setting (2) in Section 4.2. We show that after preprocessing for the pattern on an general alphabet in $O(mM \log |\Sigma|)$, the full-/sub-permuted pattern matching problem can be solved in $O(nN \log |\Sigma|)$. At last we address the setting (3) in Section 4.3. For this problem setting, we propose three indexing structures of multi-track strings. One of these structures, called a *multi-track suffix tree*, is based on the suffix tree [43]. The other structures are based on the position heap [21], called a *multi-track position heap* and a *contracted multi-track position heap*. The multi-track suffix tree needs $O(nN)$ space and can be constructed in $O(nN \log |\Sigma|)$ time. It

4.1 Algorithm based on the generalized suffix array

enables to perform fast full-permuted pattern matching running in $O(mN \log |\Sigma| + occ)$ time. On the other hand, the matching by using the contracted multi-track position heap takes $O(m^2 N^2 \log |\Sigma| + occ)$ time. However it only needs $O(n)$ space. We will show that the contracted multi-track position heap can be build in $O(nN \log |\Sigma|)$ time through the construction of the multi-track position heap.

4.1 Algorithm based on the generalized suffix array

We first describe an algorithm using the generalized suffix array for a text and a pattern and the longest common extension (LCE). The LCE between two positions i, j respectively in strings t and p , is $\max\{k \mid t[i : i + k - 1] = p[j : j + k - 1]\}$. Consider all strings in the text and pattern multi-tracks, and preprocess them in linear time so that LCE queries between arbitrary positions of arbitrary tracks can be answered in constant time (see e.g. [26]). The preprocessing can be done in $O(nN)$ time for integer alphabets. Notice that we can determine the lexicographic order between the two strings $t[i :]$ and $p[j :]$ in constant time as well, by comparing the characters $t[i + k]$ and $p[j + k]$ which come after the LCE. Next, we calculate $SortIndex(\mathbb{T}[1 :]), \dots, SortIndex(\mathbb{T}[n :]), SortIndex(\mathbb{P})$. This can be performed in $O(nN)$ time with Lemma 3.4.

The generalized suffix array enables us to calculate both LCE and $SortIndex(\mathbb{T}[1 :]), \dots, SortIndex(\mathbb{T}[n :]), SortIndex(\mathbb{P})$. As an example, we show the generalized suffix array for $\mathbb{T} = (\text{ababaab}\$, \text{aaababa}\$, \text{babaaab}\$)$ and $\mathbb{P} = (\text{aba}\$, \text{baa}\$, \text{aba}\$)$ in Figure 4.1. $\$$ is the special symbol that is lexicographically smaller than any symbol in Σ . For any two positive integers j and k , if $j \leq k$, then $\$ \preceq \$$. Let s be the string that is concatenation of all tracks of \mathbb{T} and \mathbb{P} . For $1 \leq i \leq nN$, $s[i :]$ corresponds to $t_j[k :]$ where $j = \frac{i-1}{n} + 1$ and $k = (i - 1 \bmod n) + 1$. For $nN + 1 \leq i \leq nN + mM$, $s[i :]$ corresponds to $p_j[k :]$ where $j = \frac{i-1-nN}{m} + 1$ and $k = (i - 1 - nN \bmod m) + 1$. Let SA and LCP be the suffix array and the longest common prefix array for s , respectively. LCE queries between arbitrary positions of arbitrary tracks can be answered by

4.1 Algorithm based on the generalized suffix array

$$\mathbb{T} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = \begin{pmatrix} a & b & a & b & a & a & b & \$_1 \\ a & a & a & b & a & b & a & \$_2 \\ b & a & b & a & a & a & b & \$_3 \end{pmatrix} \quad \mathbb{P} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} a & b & a & \$_4 \\ b & a & a & \$_5 \\ a & b & a & \$_6 \end{pmatrix}$$

i	$SA[i]$	$s[SA[i]]$	$LCP[i]$
1	8	$\$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \6	0
2	16	$\$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	0
3	24	$\$3a b a \$4b a a \$5a b a \6	0
4	28	$\$4b a a \$5a b a \$6$	0
5	32	$\$5a b a \6	0
6	36	$\$6$	0
7	15	$a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	0
8	27	$a \$4b a a \$5a b a \$6$	1
9	31	$a \$5a b a \6	1
10	35	$a \$6$	1
11	30	$a a \$5a b a \6	1
12	20	$a a a b \$3a b a \$4b a a \$5a b a \6	2
13	9	$a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	4
14	5	$a a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \6	2
15	21	$a a b \$3a b a \$4b a a \$5a b a \6	3
16	10	$a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	3
17	6	$a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \6	1
18	22	$a b \$3a b a \$4b a a \$5a b a \6	2
19	13	$a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	2
20	25	$a b a \$4b a a \$5a b a \$6$	3
21	33	$a b a \$6$	3
22	18	$a b a a a b \$3a b a \$4b a a \$5a b a \6	3
23	3	$a b a a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a$	4
24	11	$a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	3
25	1	$a b a b a a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$$	5
26	7	$b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \6	0
27	23	$b \$3a b a \$4b a a \$5a b a \6	1
28	14	$b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	1
29	26	$b a \$4b a a \$5a b a \$6$	2
30	34	$b a \$6$	2
31	29	$b a a \$5a b a \6	2
32	19	$b a a a b \$3a b a \$4b a a \$5a b a \6	3
33	4	$b a a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a$	3
34	12	$b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a b a \$6$	2
35	17	$b a b a a a b \$3a b a \$4b a a \$5a b a \6	4
36	2	$b a b a a b \$1a a a b a b a \$2b a b a a a b \$3a b a \$4b a a \$5a$	4

The LCE between $t_1[5:]$ and p_1 is 1.

We see $SI(\mathbb{T}[1:]) = (2, 1, 3)$ from SA .

Figure 4.1: The generalized suffix array for all tracks of a text $\mathbb{T} = (ababaab\$1, aaababa\$2, babaaab\$3)$ and a pattern $\mathbb{P} = (aba\$4, baa\$5, aba\$6)$.

4.2 Algorithm based on the AC automaton

Range Minimum Queries (RMQ) in *LCP*. For example in Figure 4.1, LCE between $t_1[5:]$ and p_1 is 1, because positions $5 = SA[14]$ and $25 = SA[20]$ correspond to $t_1[5:]$ and p_1 respectively, and the minimum value in *LCP* for $14 \leq i \leq 20$ is 1. The permutations $SortIndex(\mathbb{T}[k:])$ for any $1 \leq k \leq n$ or $SortIndex(\mathbb{P})$ can be calculated by scanning *SA*. Consider $SortIndex(\mathbb{T}[1:])$, for example. $t_1[3:]$, $t_2[3:]$, and $t_3[3:]$ correspond to $s[3:]$, $s[11:]$, and $s[19:]$, respectively. When we scan $SA[i]$ in ascending order of i from $i = 1$, we will encounter $SA[13] = 9$, $SA[25] = 1$, and $SA[35] = 17$ in this order. Because this order is lexicographical order, we see $SortIndex(\mathbb{T}[1:]) = (2, 1, 3)$ immediately.

Finally, for each position $1 \leq i \leq n - m + 1$ in the text, check whether or not $\mathbb{P} \stackrel{\exists}{=} \mathbb{T}[i:]$. This check can be conducted in $O(N + M)$ time for each position, by comparing each text track $\mathbb{T}[i:]$ and pattern track \mathbb{P} in the order of $SortIndex(\mathbb{T}[i:]) = (r_{i,1}, \dots, r_{i,N})$ and $SortIndex(\mathbb{P}) = (r_1, \dots, r_M)$ respectively, to see if there exists $1 \leq j_1 < \dots < j_M \leq N$ such that $t_{r_{i,j_k}}[i : i + m - 1] = p_{r_k}$ for all k ($1 \leq k \leq M$). Each pair of text and pattern tracks can be checked in constant time using an LCE query. Since the tracks are checked in lexicographic order, the number of checks required is at most $O(N + M) = O(N)$. The total time for the algorithm is thus $O(nN)$.

Theorem 4.1. *Given multi-track text $\mathbb{T} = (t_1, \dots, t_N)$ where $|\mathbb{T}|_{len} = n$, and multi-track pattern $\mathbb{P} = (p_1, \dots, p_M)$, where $M \leq N$ and $|\mathbb{P}|_{len} = m \leq n$, the permuted matching problem can be solved in $O(nN)$ time and space, assuming an integer alphabet.*

4.2 Algorithm based on the AC automaton

Second, we describe an algorithm based on the Aho-Corasick (AC) automaton [1]. The AC automaton is a well known pattern matching automaton that can find occurrences of multiple pattern strings in a given text string. The automaton is traversed with each character $T[k]$ ($1 \leq k \leq |T|$) of the text T , and if an accepting state is reached, it means that position k is the end position of patterns that can be identified by the state. (We

4.2 Algorithm based on the AC automaton

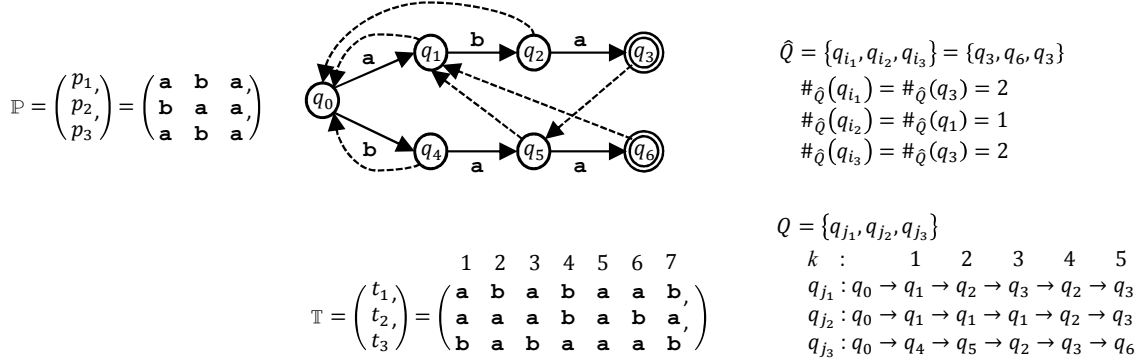


Figure 4.2: The AC automaton of a multi-track $\mathbb{P} = (p_1, p_2, p_3) = (\text{aba}, \text{baa}, \text{aba})$. The broken lines denote transitions of the failure function.

shall omit technical details of the traversal involving failure links in this paper, as they are identical to the original work.) It is known that the AC automaton can be constructed in time linear in the total length of the pattern strings [1, 20].

We solve the permuted matching problem as follows. First, construct the AC automaton for the strings in the pattern multi-track $\mathbb{P} = (p_1, \dots, p_M)$. Let $\hat{Q} = \{q_{i_1}, \dots, q_{i_M}\}$ denote the multi-set of accepting states corresponding to each track of the pattern. Next, we traverse the automaton with each track of the text multi-track in a parallel manner. Let $Q = \{q_{j_1}, \dots, q_{j_N}\}$ be the multi-set of states reached after traversing with $\mathbb{T}[k]$. Then, $\mathbb{P} \sqsubseteq \mathbb{T}[k - |\mathbb{P}|_{\text{len}} + 1 : k]$ if and only if all the M accepting states are included in Q , i.e., for any $q \in \hat{Q}$, $\#_Q(q) \geq \#_{\hat{Q}}(q)$. This can be easily checked in $O(N)$ time for each position. Since the traversal of the AC automaton can be conducted in $O(n \log |\Sigma|)$ time for each track, the total time required for the traversal is $O(nN \log |\Sigma|)$.

Theorem 4.2. *A multi-track pattern $\mathbb{P} = (p_1, \dots, p_M)$, where $|\mathbb{P}|_{\text{len}} = m$, can be preprocessed in $O(mM \log |\Sigma|)$ -time and $O(mM)$ space so that the permuted matching problem for any multi-track text $\mathbb{T} = (t_1, \dots, t_N)$ where $N \geq M$ and $|\mathbb{T}|_{\text{len}} = n \geq m$, can be solved in $O(nN \log |\Sigma|)$ time and $O(N)$ working space.*

Simple example of permuted matching using the AC automaton for $\mathbb{P} = (p_1, p_2, p_3) = (\text{aba}, \text{baa}, \text{aba})$ is shown in Figure 4.2. For \mathbb{P} , the multi-set of accepting states $\hat{Q} =$

4.3 Indexing structures for multi-track strings

$\{q_{i_1}, q_{i_2}, q_{i_3}\}$ is $\{q_3, q_6, q_3\}$. The multiplicities are obtained as $\#_{\hat{Q}}(q_{i_1}) = 2, \#_{\hat{Q}}(q_{i_2}) = 1, \#_{\hat{Q}}(q_{i_3}) = 2$. Given a multi-track text $\mathbb{T} = (t_1, t_2, t_3) = (\text{ababaab}, \text{aaababa}, \text{babaaab})$, $\mathbb{P} \stackrel{\boxtimes}{\subseteq} \mathbb{T}[3 : 5]$. Because, the multi-set of states $Q = \{q_{j_1}, q_{j_2}, q_{j_3}\}$ corresponding to t_1, t_2, t_3 after traversing $\mathbb{T}[5]$ are $\{q_3, q_3, q_6\}$ and satisfy $\#_Q(q) \geq \#_{\hat{Q}}(q)$ for any $q \in \hat{Q}$.

4.3 Indexing structures for multi-track strings

4.3.1 Multi-track suffix trees

In this section, we present a new data structure called a *multi-track suffix tree*, as well as an efficient algorithm for constructing it, with which we can solve the full-permuted multi-track matching problem efficiently. Namely, we will show that we can construct the multi-track suffix tree for any text multi-track $\mathbb{T} = (t_1, \dots, t_N)$ in $O(nN \log |\Sigma|)$ -time and $O(nN)$ space where $|\mathbb{T}|_{len} = n$. Using the multi-track suffix tree, we can conduct full permuted matching for any multi-track pattern $\mathbb{P} = (p_1, \dots, p_N)$, where $|\mathbb{P}|_{len} = m$, in $O(mN \log |\Sigma| + occ)$ time.

The construction algorithm is based on Ukkonen's suffix tree construction algorithm [42] for a single string. The suffix tree $ST(w)$ for a string w over Σ is a compacted trie of all suffixes of w . Using the suffix tree of a text, we can find occurrences of patterns in the text efficiently (see, e.g., [16, 24]). Ukkonen showed that $ST(w)$ is constructed in $O(n \log |\Sigma|)$ time.

We show the definition of the multi-track suffix tree (the mt-suffix tree, in short) in following. For multi-track $\mathbb{T} = (t_1, t_2, \dots, t_N)$ with $|\mathbb{T}|_{len} = n$ and $|\mathbb{T}|_{num} = N$, we assume that any track t_i terminates with a special symbol $\$i$, where each $\$i \notin \Sigma$ is lexicographically smaller than any character in Σ , and $\$1 \prec \$2 \prec \dots \prec \$N$. The mt-suffix tree of a multi-track text \mathbb{T} , denoted $MTST(\mathbb{T})$, is a compacted trie of all sorted mt-suffixes of \mathbb{T} , i.e. $Sort(\mathbb{T}[1 :]), Sort(\mathbb{T}[2 :]), \dots, Sort(\mathbb{T}[n :])$, that each edge is labeled by a mt-substring of \mathbb{T} . Fig 4.3 illustrates $MTST(\mathbb{T})$ with $\mathbb{T} = (\text{ababaab}\$, \text{aaababa}\$, \text{babaaab}\$)$, where

4.3 Indexing structures for multi-track strings

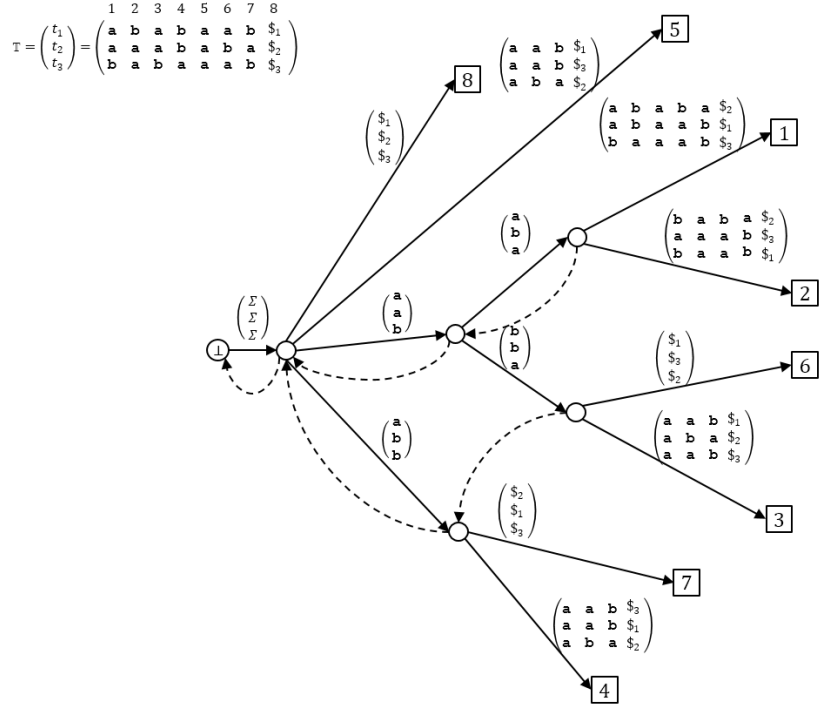


Figure 4.3: The multi-track suffix tree $MTST(\mathbb{T})$ of a multi-track $\mathbb{T} = (ababaab\$1, aaababa\$2, babaaab\$3)$.

dot-lines denote the suffix links that we will describe later.

The following lemma shows a simple but important observation regarding multi-track text \mathbb{T} .

Lemma 4.3. *For any multi-track strings $\mathbb{X} \in \Sigma_N^+$ and $\mathbb{Y}, \mathbb{Z} \in \Sigma_N^*$, $Sort(\mathbb{X}\mathbb{Y})[1 : |\mathbb{X}|_{len}] = Sort(\mathbb{X}\mathbb{Z})[1 : |\mathbb{X}|_{len}]$.*

Proof: Let $\mathbb{X} = (x_1, x_2, \dots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \dots, y_N)$. For any $1 \leq i \neq j \leq N$, we consider the following two cases: (1) if $x_i \neq x_j$: We assume w.l.o.g. that $x_i \prec x_j$. Then $x_i y_i \prec x_j y_j$. (2) if $x_i = x_j$: We assume w.l.o.g. that $y_i \preceq y_j$. Then although $x_i y_i \preceq x_j y_j$, we have $x_i y_i[1 : |\mathbb{X}|_{len}] = x_j y_j[1 : |\mathbb{X}|_{len}]$. In either case, the lexicographical order between $x_i y_i$ and $x_j y_j$ depends only on the prefixes x_i and x_j . It is same for $x_i z_i$ and $x_j z_j$. Thus, $Sort(\mathbb{X}\mathbb{Y})[1 : |\mathbb{X}|_{len}] = Sort(\mathbb{X}) = Sort(\mathbb{X}\mathbb{Z})[1 : |\mathbb{X}|_{len}]$. ■

Due to the above lemma, for any mt-substring \mathbb{X} of \mathbb{T} , there is a path that spells out $Sort(\mathbb{X})$ in $MTST(\mathbb{T})$. This also implies that a multi-track pattern \mathbb{P} with $|\mathbb{P}|_{num} = |\mathbb{T}|_{num}$

4.3 Indexing structures for multi-track strings

has an occurrence in \mathbb{T} iff there is a path that spells out $\text{Sort}(\mathbb{P})$ in $\text{MTST}(\mathbb{T})$. More in detail, we can solve the full-permuted matching as follows:

Theorem 4.4. *Let $\mathbb{T} = (t_1, \dots, t_N)$ with $|\mathbb{T}|_{\text{len}} = n$. We can augment $\text{MTST}(\mathbb{T})$ in $O(nN \log |\Sigma|)$ time and $O(nN)$ space so that the full-permuted matching problem for any given multi-track pattern $\mathbb{P} = (p_1, \dots, p_N)$ with $|\mathbb{P}|_{\text{len}} = m$ can be solved in $O(mN \log |\Sigma| + \text{occ})$ time.*

Proof: First we compute $\text{Sort}(\mathbb{P})$ in $O(mN \log |\Sigma|)$ time using a trie that represents all tracks in \mathbb{P} . We then traverse down $\text{MTST}(\mathbb{T})$ from the root to search for a path that corresponds to $\text{Sort}(\mathbb{P})$. At each edge of $\text{MTST}(\mathbb{T})$, comparisons of mt-characters between \mathbb{T} and \mathbb{P} can be done in $O(N)$ time provided that $\text{SortIndex}(\mathbb{T}[1 :]), \dots, \text{SortIndex}(\mathbb{T}[n :])$ have been already computed by Lemma 3.4 in $O(nN \log |\Sigma|)$ time and $O(nN)$ space. Since the number of children of a node in $\text{MTST}(\mathbb{T})$ is $O(n)$, a straightforward search for occurrences of \mathbb{P} takes a total of $O(mN \log n + \text{occ})$ time. However, we can reduce the cost to $O(mN \log |\Sigma| + \text{occ})$ by the following preprocessing on \mathbb{T} : In each node of $\text{MTST}(\mathbb{T})$ we maintain a trie that represents the first mt-characters of the labels of the node. Since each mt-character in \mathbb{T} is of length N , searching at each node can be done in $O(N \log |\Sigma|)$ time. Since there are $O(n)$ edges in $\text{MTST}(\mathbb{T})$, these tries for all nodes occupy a total of $O(nN)$ space and can be constructed in a total of $O(nN \log |\Sigma|)$ time. ■

Clearly $\text{MTST}(\mathbb{T})$ has n leaves and each internal node has more than one child node, and hence the number of all nodes of $\text{MTST}(\mathbb{T})$ is at most $2n - 1$ and the number of all edges is at most $2n - 2$. Each edge of $\text{MTST}(\mathbb{T})$ is labeled by \mathbb{S} such that $\mathbb{T}[b : e] \langle \text{SortIndex}(\mathbb{T}[j :]) \rangle = \mathbb{S}$, and the label is represented by a triple (b, e, j) . Thus we can represent $\text{MTST}(\mathbb{T})$ in a total of $O(nN)$ space. Note that representing edge labels using the triples does not increase the asymptotic time complexity of the pattern matching algorithm of Theorem 4.4.

We shall use the next lemma to show the correctness of our algorithm which constructs $\text{MTST}(\mathbb{T})$.

4.3 Indexing structures for multi-track strings

Lemma 4.5. *Let $\mathbb{A}\mathbb{X}$ be a substring of \mathbb{T} with $\mathbb{A} \in \Sigma_N$ and $\mathbb{X} \in \Sigma_N^*$. There are paths that spell out $\text{Sort}(\mathbb{A}\mathbb{X})$ and $\text{Sort}(\mathbb{X})$ from the root of $MTST(\mathbb{T})$.*

Proof: Let $\mathbb{Y} \in \Sigma_N^*$ be a multi-track s.t. $\mathbb{A}\mathbb{X}\mathbb{Y}$ is a suffix of \mathbb{T} . Then there is a path spelling out $\text{Sort}(\mathbb{A}\mathbb{X}\mathbb{Y})$ from the root of $MTST(\mathbb{T})$. By Lemma 4.3, $\text{Sort}(\mathbb{A}\mathbb{X}) = \text{Sort}(\mathbb{A}\mathbb{X}\mathbb{Y})[1 : \ell(\mathbb{A}\mathbb{X})]$, and therefore there is a path spelling out $\text{Sort}(\mathbb{A}\mathbb{X})$ from the root. Since $\mathbb{X}\mathbb{Y}$ is also a suffix of \mathbb{T} , paths spelling out $\text{Sort}(\mathbb{X}\mathbb{Y})$ and $\text{Sort}(\mathbb{X})$ from the root exist for the same reasoning as above. ■

Now we propose an efficient $O(nN \log |\Sigma|)$ -time algorithm that constructs the mt-suffix tree. The algorithm consists of two parts (a pseudo-code is shown in Algorithm 2). In the first part, we compute the permutation for the sorted mt-suffixes of \mathbb{T} of each length, $\text{SortIndex}(\mathbb{T}[1 :]), \dots, \text{SortIndex}(\mathbb{T}[n :])$. In the second part, we construct the mt-suffix tree by using the permutations above.

Assume that for a node v of $MTST(\mathbb{T})$ and a mt-substring \mathbb{X} of \mathbb{T} , $\text{label}(v)$ is a mt-prefix of $\text{Sort}(\mathbb{X})$. We represent $\text{Sort}(\mathbb{X})$ by a *reference pair* $(v, (b, e, j))$ such that $\text{Sort}(\mathbb{X}) = \text{label}(v)\mathbb{T}[b : e]\langle \text{SortIndex}(\mathbb{T}[j :]) \rangle$. If v is the deepest node such that $\text{label}(v)$ is a prefix of $\text{Sort}(\mathbb{X})$, then the pair $(v, (b, e, j))$ is said to be *canonical*.

We define the suffix link on the mt-suffix tree as follows:

Definition 4.6 (Multi-track suffix link). *Let $\text{slink}(\text{root}) = \perp$. For any non-root explicit node v of $MTST(\mathbb{T})$, if $\text{label}(v) = \text{Sort}(\mathbb{A}\mathbb{X})$ where $\mathbb{A} \in \Sigma_N$, $\mathbb{X} \in \Sigma_N^*$ and $\mathbb{A}\mathbb{X}$ is a mt-substring of \mathbb{T} , then $\text{slink}(v) = u$, where $\text{label}(u) = \text{Sort}(\mathbb{X})$.*

We remark the suffix link of a node of $MTST(\mathbb{T})$ may point to an implicit node: Let $\mathbb{A}\mathbb{X}$ be any substring of \mathbb{T} with $\mathbb{A} \in \Sigma_N$ and $\mathbb{X} \in \Sigma_N^*$. If $\text{Sort}(\mathbb{A}\mathbb{X})$ is an explicit node v of $MTST(\mathbb{T})$, then there are at least two distinct non-empty multi-track strings $\mathbb{Y}, \mathbb{Z} \in \Sigma_N^+$ for which there exist paths spelling out $\text{Sort}(\mathbb{A}\mathbb{X}\mathbb{Y})$ and $\text{Sort}(\mathbb{A}\mathbb{X}\mathbb{Z})$ from the root. It follows from Lemma 4.5 that there also exist paths from the root which spell out $\text{Sort}(\mathbb{X}\mathbb{Y})$ and $\text{Sort}(\mathbb{X}\mathbb{Z})$. However, $\text{Sort}(\mathbb{X}\mathbb{Y})[: |\mathbb{X}|_{\text{len}} + 1]$ may or may not be equal

4.3 Indexing structures for multi-track strings

Algorithm 2: Algorithm to construct multi-track suffix trees

Input: multi-track text \mathbb{T} of length n and of track count N

```

1  compute  $SortIndex(\mathbb{T}[i :])$  for all  $1 \leq i \leq n$ ;
2  create nodes  $root$  and  $\perp$ , and an edge with label  $\Sigma_N$  from  $\perp$  to  $root$ ;
3   $slink(root) := \perp$ ;
4   $(v, (b, e, j)) := (root, (1, 0, 1))$ ;  $oldu := root$ ;
5  foreach  $i = 1, \dots, n$  do
6      while  $j \leq n$  do
7          if  $b \leq e$  and  $\mathbb{T}[i] \langle SortIndex(\mathbb{T}[j :]) \rangle = \mathbb{T}[e + 1] \langle SortIndex(\mathbb{T}[b :]) \rangle$  then
8               $(v, (b, e, j)) := canonize(v, (b, e + 1, j))$ ;
9              break;
10         if  $b > e$  and there is a  $\mathbb{T}[i] \langle SortIndex(\mathbb{T}[j :]) \rangle$ -edge from  $v$  then
11              $(v, (b, e, j)) := canonize(v, (i, i, j))$ ;
12             break;
13          $u := v$ ;
14         if  $b \leq e$  then
15             let  $(v, (p, q, h), z)$  be the  $\mathbb{T}[b] \langle SortIndex(\mathbb{T}[j :]) \rangle$ -edge from  $v$ ;
16             split the edge to two edges  $(v, (p, p + e - b, h), u)$  and
17                  $(u, (p + e - b + 1, q, h), z)$ ;
18             if  $oldu \neq root$  then  $slink(oldu) := u$ ;
19         create a new edge  $(u, (i, \infty, j), \ell)$  with a new leaf  $\ell$ ;
20          $oldu := u$ ;
21         if  $slink(u)$  is defined then
22              $v := slink(u)$ ;  $j := j + 1$ ;
23         else  $(v, (b, e, j)) := canonize(slink(v), (b, e, j + 1))$ ;

```

to $Sort(\mathbb{XZ})[|\mathbb{X}|_{len} + 1]$. Thus, $Sort(\mathbb{X})$, which is pointed to by $slink(v)$, can be an implicit node. Note that, Lemma 4.5 guarantees that there always exists a node pointed by $slink(v)$ for an explicit node v even though the node may be an implicit node. Still, we can design an algorithm to construct $MTST(\mathbb{T})$ for a given multi-track text \mathbb{T} based on the Ukkonen algorithm [42], using similar techniques to construction of parameterized suffix trees [4].

In the initial phase, we create $MTST(\mathbb{E}_N)$ that consists only of the root node, the auxiliary node, and the edge and the suffix link between them (in Lines 2 and 3 of Algorithm 2). In phase $i \geq 1$, our algorithm updates $MTST(\mathbb{T}[1 : i - 1])$ to $MTST(\mathbb{T}[1 : i])$. The update operation starts at a location in $MTST(\mathbb{T}[1 : i - 1])$ called the *active*

4.3 Indexing structures for multi-track strings

Algorithm 3: Function *canonize*

Input: reference pair $(v, (b, e, j))$ for mt-substring $\mathbb{X} = \text{label}(v) \cdot \mathbb{T}[b : e] \langle \mathbb{T}[j :] \rangle$.

Output: canonical reference pair for mt-substring \mathbb{X} .

```

1 if  $b > e$  then return  $(v, (b, e, j))$ ;
2 find the  $\mathbb{T}[b] \langle \mathbb{T}[j : ] \rangle$ -edge  $(v, (p, q, h), u)$  from  $v$ ;
3 while  $q - p \leq e - b$  do
4    $b := b + q - p + 1$ ;  $v := u$ ;
5   if  $b \leq e$  then find the  $\mathbb{T}[b] \langle \mathbb{T}[j : ] \rangle$ -edge  $(v, (p, q, h), u)$  from  $v$ ;
6 return  $(v, (b, e, j))$ ;
```

point, which we shall refer to as AP. The second part basically follows Ukkonen's suffix tree construction algorithm, with a distinction that when we insert the j th suffix $\text{Sort}(\mathbb{T}[j :])$ into the tree, then the i th mt-character $\mathbb{T}[i]$ is read *with the permutation of the j th mt-suffix*, i.e., as $\mathbb{T}[i] \langle \text{SortIndex}(\mathbb{T}[j :]) \rangle$. The AP for the i th phase initially corresponds to the longest mt-suffix $\text{Sort}(\mathbb{T}[j : i - 1])$ of $\mathbb{T}[i - 1]$ that matches at least two positions $1 \leq k \neq h \leq i - 1$ of $\mathbb{T}[1 : i - 1]$, that is, $\text{Sort}(\mathbb{T}[k : k + i - j - 1]) = \text{Sort}(\mathbb{T}[j : i - 1])$ and $\text{Sort}(\mathbb{T}[h : h + i - j - 1]) = \text{Sort}(\mathbb{T}[j : i - 1])$. Then, we examine whether it is possible to traverse down from the AP with $\mathbb{T}[i] \langle \text{SortIndex}(\mathbb{T}[j :]) \rangle$. The examination is conducted in Line 7 when the AP is on an implicit node, and in Line 10 when the AP is on an explicit node. If it turns out to be possible, then after the reference pair is canonized, we go to the $(i + 1)$ th phase. If it turns out to be impossible, then we create a new node if the reference pair represents an implicit node (Line 14). Then a new leaf node representing $\text{Sort}(\mathbb{T}[j :])$ is inserted in Line 18, whose parent node is u . The AP is then moved using the suffix link of u if it exists (in Line 21), and using the suffix link of the parent v of u if the suffix link of u does not exist yet (in Line 22), and we go to the $(j + 1)$ th step to insert $\text{Sort}(\mathbb{T}[j + 1 :])$. It follows from the definition of the suffix link and from Lemma 4.5 that there always exists a path that spells out $\mathbb{T}[b : e] \langle \text{SortIndex}(\mathbb{T}[j + 1 :]) \rangle$ from node $\text{slink}(v)$ in Line 22.

Theorem 4.7. *Given a multi-track \mathbb{T} which $|\mathbb{T}|_{\text{len}} = n$ and $|\mathbb{T}|_{\text{num}} = N$, Algorithm 2 constructs $\text{MTST}(\mathbb{T})$ in $O(nN \log |\Sigma|)$ time and $O(nN)$ space.*

4.3 Indexing structures for multi-track strings

Proof: The correctness follows from the above arguments.

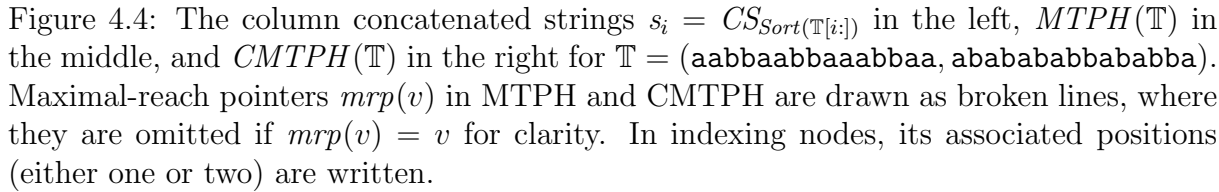
Let us analyze the time complexity. In the first part of Algorithm 2 (in Line 1), the permutations for all sorted mt-suffixes of \mathbb{T} can be computed in $O(nN)$ time due to Lemma 3.4. Next, we consider the time complexity of the second part. The condition of Line 7 can be checked in constant time using the permutations for sorted suffixes, and that of Line 10 can be checked in $O(N \log |\Sigma|)$ time using the same way to Theorem 4.4. At each step the function *canonicalize* is called at most once. A pseudo-code for *canonicalize* is given in Algorithm 3. Using a similar analysis to the Ukkonen algorithm [42], the amortized number of nodes that are visited at each call of *canonicalize* is constant. Since it takes $O(N \log |\Sigma|)$ time to search a branching node (in Lines 2 and 5), the amortized cost for canonicalizing a reference pair is $O(N \log |\Sigma|)$. We consider the number of steps in the double loop. Both i and j are monotonically non-decreasing, and they satisfy $1 \leq i$ and $j \leq n$. Therefore the total number of steps of the algorithm is $O(n)$. Thus, the second part of the Algorithm 2 takes $O(nN \log |\Sigma|)$ time, and hence the entire algorithm works in a total of $O(nN \log |\Sigma|)$ time.

The permutations for each sorted mt-suffix require $O(nN)$ space, and the tries for searching branches of each node require a total of $O(nN)$ space. Hence the overall space complexity is $O(nN)$. ■

4.3.2 Multi-track position heaps

In this section, we propose a new data structure, named *multi-track position heap* (shortly MTPH), based on the position heap [21] for a single string. We will show a construction algorithm of MTPH for any multi-track $\mathbb{T} \in \Sigma_N^n$ in $O(nN \log |\Sigma|)$ time and $O(nN)$ space. The construction algorithm is based on Kucherov's position heap construction algorithm [33]. Using MTPH, the full-permuted pattern matching can be solved in $O(mN^2 \log |\Sigma| + occ)$ time for $\mathbb{P} \in \Sigma_N^m$.

We define a *column concatenated string* $CS_{\mathbb{T}}$ of a multi-track $\mathbb{T} = (t_1, t_2, \dots, t_N)$ by


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

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

Definition 4.8 (MTPH). Let \mathbb{T} be a multi-track string of length n and track count N . Let $s_{i,j} = CS_{Sort(\mathbb{T}[i:j])}$ for $1 \leq i \leq j \leq n$, and $s_{i,n}$ is denoted by s_i briefly. Let $S = \{s_1, s_2, \dots, s_n\}$ be an ordered set of the strings. For $1 \leq i \leq n$, $MTPH_i(\mathbb{T}) = (V_i, E_i)$ is a trie recursively defined by $(V_0, E_0) = (\{\text{root}\}, \emptyset)$, and

$$E_i = E_{i-1} \cup \bigcup_{j=0}^{\Delta_i-1} \{(\overline{s_i[:|q_i|+j-1]}, s_i[|q_i|+j], \overline{s_i[:|q_i|+j]})\},$$

Figure 4.4 shows an example of $MTPH(\mathbb{T})$ and column concatenated strings. Both

4.3 Indexing structures for multi-track strings

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

We will show that $MTPH(\mathbb{T})$ can be constructed in $O(nN)$ time by updating $MTPH(\mathbb{T}[: i - 1])$ to $MTPH(\mathbb{T}[: i])$ iteratively for $i = 1, 2, \dots, n$, similar to the online construction algorithm for position heaps [33]. We remark that it is not trivial because s_i is not necessarily a suffix of s_{i-1} (see Figure 4.4, left). Let us focus on the differences between $MTPH(\mathbb{T}[: i - 1])$ and $MTPH(\mathbb{T}[: i])$. For $1 \leq j \leq i$, if j is a primary position in a node v in $MTPH(\mathbb{T}[: i - 1])$, j must be the primary position stored in the same node v in $MTPH(\mathbb{T}[: i])$. If j is a secondary position in $MTPH(\mathbb{T}[: i - 1])$, there are two cases in $MTPH(\mathbb{T}[: i])$: (1) j becomes a primary position in a newly created node v' , or (2) j remains the secondary position, but in another node v' . In any case, the node v' is in $Des(v)$. Thus, we consider how to update the nodes storing two positions.

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

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

Lemma 4.9. *For two multi-tracks $\mathbb{X} = (x_1, x_2, \dots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \dots, y_N)$, if $Sort(\mathbb{X}) = Sort(\mathbb{Y})$ then $Sort(\mathbb{X}[2 :]) = Sort(\mathbb{Y}[2 :])$.*

4.3 Indexing structures for multi-track strings

Proof: Trivial. ■

Lemma 4.10. *For any multi-track \mathbb{W} of length m , if $CS_{Sort(\mathbb{W})}$ is represented in $MTPH_i(\mathbb{T})$, then $CS_{Sort(\mathbb{W}[2:])}$ is also represented in $MTPH_i(\mathbb{T})$ for any $1 \leq i \leq n$.*

Proof: Because $CS_{Sort(\mathbb{W})}$ is represented in $MTPH_i(\mathbb{T})$, there are $1 \leq j_1 \leq j_2 \leq \dots \leq j_m \leq i$ such that $Sort(\mathbb{T}[j_k : j_k + k - 1]) = Sort(\mathbb{W}[1 : k])$ for $1 \leq k \leq m$, and they have been inserted to MTPH in the order of $k = 1, 2, \dots, m$. At the same time, $Sort(\mathbb{T}[j_k + 1 : j_k + k - 1])$ for $1 \leq k \leq m$ have also been inserted in this order, because MTPH is constructed by inserting suffixes in descending order with respect to the length. Lemma 4.9 leads $Sort(\mathbb{T}[j_k + 1 : j_k + k]) = Sort(\mathbb{W}[2 : k])$. Thus, $CS_{Sort(\mathbb{T}[j_{m+1} : j_m + m])} = CS_{Sort(\mathbb{W})[2:]}$ is represented in $MTPH_i(\mathbb{T})$. ■

Lemma 4.11. *If b is a secondary position of a node v in $MTPH_i(\mathbb{T})$ for $1 \leq b < i$, then $b + 1$ is also a secondary position of another node in it.*

Proof: Let b' be the primary position of v . Then, $b' < b$ and $s_b = s_{b'}[|s_b|]$ hold. From Lemma 4.10, s_{b+1} is represented in $MTPH_i(\mathbb{T})$. In addition, $s_{b'+1}[|s_{b+1}|]$ is also represented. From Lemma 4.9, $s_{b+1} = s_{b'+1}[|s_{b+1}|]$ holds. Since $b' < b$, $b' + 1 < b + 1$. Thus, $b + 1$ is a secondary position of $\overline{s_{b'+1}}$. ■

Let b be the smallest position that is a secondary position in $MTPH(\mathbb{T}[1 : i - 1])$ with $1 \leq b \leq i - 1$. By Lemma 4.11, all the secondary positions are written as $b, b + 1, \dots, i - 1$. In addition, these positions are partitioned into two intervals. Let b' be the smallest position such that $s_{b',i}$ is represented in $MTPH(\mathbb{T}[1 : i - 1])$. Then, $s_{b'+1,i}$ is also represented in it by Lemma 4.10. Similarly, all $s_{b'+2,i}, \dots, s_{i-1,i}$ are represented in it, too. Therefore, all the positions $b, b + 1, \dots, b' - 1$ in the first interval are primary positions in $MTPH(\mathbb{T}[1 : i])$, while all $b', b' + 1, \dots, i - 1$ in the second interval are secondary positions in $MTPH(\mathbb{T}[1 : i])$. Summarizing the above discussion, to obtain $MTPH(\mathbb{T}[1 : i])$, $MTPH(\mathbb{T}[1 : i - 1])$ should be updated as follows: (1) for $b \leq j < b'$, build $path(\overline{s_{j,i-1}}, \overline{s_{j,i}})$ and reset the position j from

4.3 Indexing structures for multi-track strings

$\overline{s_{j,i-1}}$ to the new node $\overline{s_{j,i}}$ as its primary position, and (2) for $b' \leq j \leq i$, reset the position j from $\overline{s_{j,i-1}}$ to the existing node $\overline{s_{j,i}}$ as its secondary position. We refer the position b as the *active position*, and the indexing node $\overline{s_{b,i-1}}$ as the *active node*, similarly to [33]. The nodes storing positions $b, b+1, \dots, i-1$ can be traversed efficiently by using the suffix pointers defined below.

Definition 4.12 (Multi-track suffix pointers). *For any indexing node $\overline{s_{i,j}}$ in $MTPH(\mathbb{T})$, the multi-track suffix pointer of $\overline{s_{i,j}}$ is a pointer from $\overline{s_{i,j}}$ to the node $\overline{s_{i+1,j}}$, and denoted as $mtsp(\overline{s_{i,j}}) = \overline{s_{i+1,j}}$.*

For every indexing node $\overline{s_{i,j}}$ in $MTPH(\mathbb{T}[i])$, the existence of $mtsp(\overline{s_{i,j}})$ is guaranteed by Lemma 4.10. In our algorithm, we will use a chain of N nodes $\perp_1, \perp_2, \dots, \perp_N$, such that each \perp_k ($1 \leq k \leq N$) is connected to \perp_{k+1} by an edge labeled by all $c \in \Sigma$, regarding that $\perp_{N+1} = \text{root}$, and $mtsp(\text{root}) = \perp_1$. They play a role of *sentinel nodes*, similarly to [42] and [33].

We now describe the construction algorithm of $MTPH(\mathbb{T})$. First of all, we compute $\text{SortIndex}(\mathbb{T}[i:n])$ for all $1 \leq i \leq n$ in $O(nN \log |\Sigma|)$ time from Lemma 3.4. It determines every $s_i = CS_{\text{Sort}(\mathbb{T}[i:])}$. In each iteration, we do not need to keep all the secondary nodes to update $MTPH(\mathbb{T}[i-1])$, because these nodes can be visited through the suffix pointers recursively from the active node. Thus, we only maintain the active position b and the active node $\overline{s_{b,i-1}}$. If there is no secondary node in $MTPH(\mathbb{T}[i-1])$, the active node is *root* and the active position is i .

In i -th iteration, the algorithm checks whether there is $\text{path}(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ or not. If it does not exist, the algorithm performs the modifications of Case (1) described above. After the modification, the new indexing node $\overline{s_{b,i}}$ is created as a descendant of $\overline{s_{b,i-1}}$. Then, the active position and the active node are updated to $b+1$ and $mtsp(\overline{s_{b,i-1}}) = \overline{s_{b+1,i-1}}$ respectively, and the algorithm performs the above process iteratively until the path is found. The multi-track suffix pointer $mtsp(\overline{s_{b,i}})$ is built as $mtsp(\overline{s_{b,i}}) = \overline{s_{b+1,i}}$ after the next modification. When $\text{path}(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ is found, the algorithm updates the active node

4.3 Indexing structures for multi-track strings

to $\overline{s_{b,i}}$ and makes the suffix pointer from the last created indexing node to the new active node if such a node exists. Hence, for any indexing node, suffix pointer of it is defined indeed. To update $MTPH(\mathbb{T}[: i - 1])$ into $MTPH(\mathbb{T}[: i])$, it is enough to perform only the modifications of Case (1), because the modifications of Case (2) does not add any node nor edge to $MTPH$. All the secondary positions will be determined after constructing $MTPH(\mathbb{T})$ by traversing nodes through the suffix pointers recursively from the active node.

Algorithm 4 shows a pseudo-code of the construction algorithm, and the function to find $path(\overline{s_{b,i-1}}, \overline{s_{b,i}})$ at Line 27. Let us analyze the running time of Algorithm 4. Each iteration of the **while**-loop from Line 9 takes $O(nN \log |\Sigma|)$ time over the whole run of the algorithm, because at most N nodes and edges are visited or created in each iteration and $1 \leq b \leq n$. Each process in the rest of the **for**-loop from Line 6 takes at most $O(N)$ time, and the loop iterates exactly n times. Thus, the running time of Algorithm 4 is $O(nN)$ time.

Theorem 4.13. *Algorithm 4 constructs $MTPH(\mathbb{T})$ in $O(nN \log |\Sigma|)$ time and $O(nN)$ space.*

We now consider to solve Problem 1 for a pattern \mathbb{P} by using $MTPH(\mathbb{T})$. Let w be the longest prefix of $CS_{Sort}(\mathbb{P})$ represented in $MTPH(\mathbb{T})$. We can compute w in $O(mN \log |\Sigma|)$ time by traversing $path(root, \overline{w})$. If $w = CS_{Sort}(\mathbb{P})$, all the positions stored in the nodes of the subtree rooted by \overline{w} are the occurrences of the pattern \mathbb{P} in \mathbb{T} . We will deal with enumerating all these positions later. Before it, remark that we should also consider another type of occurrences; let I be the set of positions stored in the nodes on $path(root, \overline{w})$. These are also candidates for the occurrences. Because $path(root, \overline{w})$ contains at most m indexing nodes, $|I| = O(m)$. For each $i \in I$, we check whether $\mathbb{P} \cong \mathbb{T}[i : i + m - 1]$ or not by simply comparing $CS_{Sort}(\mathbb{P})$ with $s_{i,i+m-1} = CS_{Sort}(\mathbb{T}[i:i+m-1])$, and report it if it does. Both the computation of $s_{i,i+m-1}$ and the comparison are done in $O(mN)$ time.

We now explain how to enumerate all the positions in the nodes of the subtree rooted

4.3 Indexing structures for multi-track strings

Algorithm 4: MTPH construction algorithm

Input: \mathbb{T}
Output: $MTPH(\mathbb{T})$

```

1  compute  $SortIndex(\mathbb{T}[i : n])$  for all  $1 \leq i \leq n$ ;
2  create  $root$  and  $\perp_N$ ;   create edge  $(\perp_N, c, root)$  for each character  $c$ ;
3  for  $j = N - 1$  downto 1 do
4     $\perp_j$  create node  $\perp_j$ ;   create edge  $(\perp_j, c, \perp_{j+1})$  for each character  $c$ ;
5   $mtsp(root) = \perp_1$ ;    $activeNode = root$ ;    $b = 1$ ;
6  for  $i = 1$  to  $n$  do
7     $lastNode = \text{undefined}$ ;
8     $targetNode = find(activeNode, s_b[\text{depth}(activeNode)+1 :]);$ 
9    while  $targetNode = \text{undefined}$  do
10      $u = activeNode$ ;
11      $w = s_b[\text{depth}(activeNode)+1 :];$ 
12     for  $j = 1$  to  $N$  do
13       if there is not an edge labeled by  $w[j]$  from  $u$  then
14         create a node  $v$  and an edge  $(u, w[j], v)$  where
15          $label(v) = label(u) \cdot w[j]$ ;
16       else
17         Let  $v$  be a child of  $u$  connected by the edge  $(u, w[j], v)$ ;
18        $u = v$ ;
19     if  $lastNode \neq \text{undefined}$  then  $mtsp(lastNode) = u$ ;
20      $lastNode = u$ ;   Let  $lastNode$  store  $b$ ;
21      $activeNode = mtsp(activeNode)$ ;    $b = b + 1$ ;
22      $targetNode = find(activeNode, s_b[\text{depth}(activeNode)+1 :]);$ 
23    $activeNode = targetNode$ ;
24   if  $lastNode \neq \text{undefined}$  then  $mtsp(lastNode) = activeNode$ ;
25 if  $b \neq n + 1$  then
26   for  $b \leq j \leq n$  do
27     Let  $activeNode$  store  $j$ ;    $activeNode = mtsp(activeNode)$ ;
28 Function  $find(node, w)$ 
29   for  $j = 1$  to  $N$  do
30     if exist edge  $(node, w[j], v)$  then  $node = v$ ;
31     else return  $\text{undefined}$ ;
32   return  $node$ ;

```

4.3 Indexing structures for multi-track strings

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

We now show that the matching by using MTPH can be accelerated by adding *maximal-reach pointers* (shortly MRPs). MRPs are the auxiliary structures for standard position heaps proposed by Ehrenfeucht et al. [21]. We will extend it to MTPHs as follows.

Definition 4.14 (MRPs for MTPHs). *For an indexing node α_i storing i in $MTPH(\mathbb{T})$, the maximal-reach pointer of α_i is a pointer from α_i to $\overline{s_i[\ell_i]}$, and denoted by $mrp(\alpha_i) = \overline{s_i[\ell_i]}$, where $s_i[\ell_i]$ is the longest prefix of $s_i = CS_{Sort(\mathbb{T}[i:])}$ represented in $MTPH(\mathbb{T})$.*

We call a MTPH of \mathbb{T} added the maximal-reach pointers an *augmented multi-track position heap* (shortly AMTPH) of \mathbb{T} , and denoted by $AMTPH(\mathbb{T})$. Algorithm 5 is an algorithm for adding MRPs to MTPH, that is based on Kucherov's algorithm for standard position heaps [33]. First of all, the algorithm preprocesses $MTPH(\mathbb{T})$ so that for any node v , it can obtain the depth of v in $O(1)$ time, by assigning unique numbers to the nodes by the depth first traversal. In i -th iteration between Line 2 and Line 8, it adds a pointer $mrp(\alpha_i) = \overline{s_i[\ell_i]}$, where $\overline{s_i[\ell_i]}$ is determined as follows: beginning by $currNode = root$, it goes down to a child $\overline{s_i[\ell]}$ of $depth(currNode) \leq \ell \leq |s_i|$ until either $currNode$ does not have a child $\overline{s_i[\ell]}$ or $\ell = |s_i|$ holds (Line 3 to Line 6). Then, $mrp(\alpha_i)$ is obtained as $\overline{s_i[\ell_i]}$ (Line 7). The next $i+1$ -th iteration begins at $mtsp(pointerNode)$, where $pointerNode$ is the deepest indexing node visited in the i -th iteration and computed in Line 5. Note that, the process of Line 5 can be computed in constant time because whether

4.3 Indexing structures for multi-track strings

Algorithm 5: Adding maximal reach pointers for MTPH

Input: \mathbb{T} , and $MTPH(\mathbb{T})$ with multi-track suffix pointers

```

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

$currNode$ is an indexing node or not is determined by $depth(currNode) \bmod N = 0$ or not.

Let us consider the time complexity of Algorithm 5. Each process of Line 1, Line 7 and Line 8 can be done in $O(1)$ time, so that these processes take $O(n)$ time in total. Let us consider the number of executions of **while**-loop at Line 3. In each loop, $s_i[\ell]$ corresponding to a letter in the text \mathbb{T} is read. We assume $s_i[\ell_i]$ belongs to k -th column of \mathbb{T} for $1 \leq k \leq n$. k does not decrease between i -th and $(i+1)$ -th iterations because $(i+1)$ -th iteration begins at $mtsp(\overline{s_{i,k}}) = \overline{s_{i+1,k}}$. In addition, since $|s_i[\ell_i]| \geq N$, $i \leq k$ holds. Thus, all letters in \mathbb{T} are read at least one time in all iterations. On the other hand, the letters corresponding to the labels on $path(pointerNode, currNode)$ in i -th iteration can be read redundantly in $(i+1)$ -th iteration. However, the number of such labels does not exceed N in each iteration. Therefore, the total number of executions of **while**-loop does not exceed $2nN$. As a result, the running time of Algorithm 5 is $O(nN \log |\Sigma|)$ time.

Now we present the full-permuted pattern matching algorithm by using AMTPH. In the naive matching algorithm with MTPH described above, the cost of the comparison of $CS_{Sort(\mathbb{P})}$ with $s_{i,i+m-1}$ for $i \in I$ can be reduced from $O(mN)$ to $O(1)$ by using the MRPs and the lowest common ancestor queries mentioned in Lemma 2.5, if $CS_{Sort(\mathbb{P})}$ itself is represented in $AMTPH(\mathbb{T})$. Whether $CS_{Sort(\mathbb{P})} = s_{i,i+m-1}$ or not is determined by $mrp(\alpha_i) \in Des(\overline{CS_{Sort(\mathbb{P})}})$. If $LCA(mrp(\alpha_i), \overline{CS_{Sort(\mathbb{P})}}) = \overline{CS_{Sort(\mathbb{P})}}$, then $mrp(\alpha_i) \in$

4.3 Indexing structures for multi-track strings

$Des(\overline{CS_{Sort(\mathbb{P})}})$ holds. By Lemma 2.5, the query $LCA(mrp(\alpha_i), \overline{CS_{Sort(\mathbb{P})}})$ can be answered in $O(1)$ time after an $O(nN)$ time and space preprocessing of $AMTPH(\mathbb{T})$. Thus, the comparison of $CS_{Sort(\mathbb{P})}$ with $s_{i,i+m-1}$ can be done in $O(1)$ time. Hence, the total time is $O(mN \log |\Sigma| + occ)$ in this case, different from $O(m^2N \log |\Sigma| + occ)$ time of the naive algorithm. Remark that, unfortunately, this result does not improve the upper-bound of the time complexity of the matching for the worst case. If $CS_{Sort(\mathbb{P})}$ is not represented in $AMTPH(\mathbb{T})$, we must compute $CS_{Sort(\mathbb{P})} = s_{i,i+m-1}$, that takes $O(mN)$ time. In this case, all comparisons are done in $O(m^2N)$ time because $|I| < m$. By considering the above two cases, the time bound of the matching is $O(mN \log |\Sigma| + occ + m^2N \log |\Sigma|) = O(m^2N \log |\Sigma| + occ)$.

Theorem 4.15. *Problem 1 can be solved in $O(m^2N \log |\Sigma| + occ)$ time by using $MTPH(\mathbb{T})$ or $AMTPH(\mathbb{T})$.*

4.3.3 Contracted multi-track position heaps

We propose a more space-efficient version of MTPH, *contracted multi-track position heap* (shortly CMTPH), by omitting non-indexing nodes of MTPH (see Figure 4.4, right). CMTPH needs only $O(n)$ space. Thus, CMTPH is very memory-efficient compared with the mt-suffix tree and MTPH when the track count N is large. We will show that CMTPH can be constructed in $O(nN \log |\Sigma|)$ time with going through the construction of MTPH.

The definition of CMTPH is as following:

Definition 4.16 (CMPH). *Let \mathbb{T} be a multi-track string of length n and track count N . Let $S = \{s_1, s_2, \dots, s_n\}$ be an ordered set of strings, where $s_i = CS_{Sort(\mathbb{T}[i:])}$ for $1 \leq i \leq n$. A contracted multi-track position heap of \mathbb{T} , denoted by $CMTPH(\mathbb{T})$, is a sequence hash tree of S , i.e., $CMTPH(\mathbb{T}) = SHT(S)$.*

Since $CMTPH(\mathbb{T})$ is a sequence hash tree for an ordered set of cardinality n , it has $n + 1$ nodes and n edges, so that $CMTPH(\mathbb{T})$ consumes only $O(n)$ space. Given \mathbb{T} ,

4.3 Indexing structures for multi-track strings

we can construct easily $CMPH(\mathbb{T})$ in $O((nN + n^2) \log |\Sigma|)$ time as follows: compute $SortIndex(\mathbb{T}[i :])$ for $1 \leq i \leq n$ in $O(nN \log |\Sigma|)$ time, then insert all $s_i = CS_{Sort(\mathbb{T}[i:])}$ to the tree in order; each insertion can be done in $O(n \log |\Sigma|)$, so that $O(n^2 \log |\Sigma|)$ in total.

We now show a more efficient construction algorithm for $CMPH(\mathbb{T})$, that runs in $O(nN)$ time. It re-assign the positions in the nodes in $MTPH(\mathbb{T})$, and eliminates all non-indexing nodes as follows. First let us noticed that in Figure 4.4, $CMPH(\mathbb{T})$ is a subtree of $MTPH(\mathbb{T})$ with the same root node, if we ignore the positions stored in the nodes. It is shown by following lemmas.

Lemma 4.17. *Let $W = \{w_1, w_2, \dots, w_K\}$ and $W' = \{w'_1, w'_2, \dots, w'_k\}$ ($k \leq K$) be ordered sets of strings such that W' is a subset of W . Then $SHT(W')$ is a subtree of $SHT(W)$ rooted by the root of $SHT(W)$.*

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

Lemma 4.18. *For $1 \leq i \leq n$, $CMPH_i(\mathbb{T})$ is a subtree of $MTPH_i(\mathbb{T})$ rooted by the root of $MTPH_i(\mathbb{T})$, if we ignore the positions stored in the nodes.*

Proof: $CMPH_i(\mathbb{T})$ is a sequence hash tree of $S = \{s_1, s_2, \dots, s_i\}$. On the other hand, $MTPH_i(\mathbb{T})$ is equivalent to that of $S' = \bigcup_{k=1}^i \bigcup_{j=1}^{\Delta_k} \{s_k\} = \{\underbrace{s_1, \dots, s_1}_{\Delta_1}, \underbrace{s_2, \dots, s_2}_{\Delta_2}, \dots, \underbrace{s_i, \dots, s_i}_{\Delta_i}\}$. Because S is a subset of S' , the statement holds by Lemma 4.17. ■

4.3 Indexing structures for multi-track strings

Lemma 4.18 implies that all nodes and edges in $CMTPH(\mathbb{T})$ are included in $MTPH(\mathbb{T})$. Therefore, $CMTPH(\mathbb{T})$ can be obtained by the following process. For each $i = 1, 2, \dots, n$, we re-assign the position i stored in an indexing node α_i to its ancestor node $\beta_i \in Anc(\alpha_i)$, that does not store the primary position (i.e., β_i may store the secondary position) and the farthest from α_i (i.e., nearest from the *root*). If there is no such a node, then α_i keeps storing i . After that, we eliminate all nodes that stores no position. Then, the remaining tree is $CMTPH(\mathbb{T})$.

To find β_i efficiently, we use the two types of queries on a rooted tree, that are the nearest marked ancestor query and the level ancestor query referred in Lemma 2.4 and Lemma 2.5, respectively. Each query can be answered in constant time after a $O(nN \log |\Sigma|)$ -time preprocess of the tree.

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

Theorem 4.19. *Given a multi-track \mathbb{T} of length n and track count N over Σ_N , $CMTPH(\mathbb{T})$ can be constructed in $O(nN \log |\Sigma|)$ time and $O(nN)$ space.*

CMTPHs is useful for permuted pattern matching instead of MTPHs. Because any node in CMTPH stores at least one position, the candidate positions for matching is at most $|I| = O(mN)$. Thus, the time complexity is $O(m^2 N^2 \log |\Sigma| + occ)$. It can be improved by using the maximal-reach pointers for CMTPH, denoted by $cmrp(\beta_i)$, in the same way as MTPH. Because $cmrp(\beta_i) = NMA(mrp(\alpha_i))$ holds for any i after marking all β_i 's, we get them in $O(n)$ time. Thus we have the following results.

4.3 Indexing structures for multi-track strings

Theorem 4.20. *Given a multi-track \mathbb{T} of length n and track count N over Σ_N , $CMPH(\mathbb{T})$ with the maximal-reach pointers for $CMPH(\mathbb{T})$ can be constructed in $O(nN \log |\Sigma|)$ time and $O(nN)$ space.*

We call a CMPH of \mathbb{T} added the maximal-reach pointers an *augmented contracted multi-track position heap* (shortly $ACMPH$) of \mathbb{T} , and denoted by $ACMPH(\mathbb{T})$. For the permuted pattern matching, the maximal-reach pointers of CMPH work similarly to that of MTPH. Thus, it is not difficult to see that the following theorem holds.

Theorem 4.21. *Problem 1 can be solved in $O(m^2 N^2 \log |\Sigma| + occ)$ time by using $CMPH(\mathbb{T})$ or $ACMPH(\mathbb{T})$.*

Chapter 5

Approximate permuted pattern matching algorithm

In this chapter, we propose a new data structure *filtering multi-set tree* (*FILM tree*) for solving permuted pattern matching problems in an efficient manner. The matching using the FILM tree is based on the following proposition.

Proposition 5.1. *For multi-tracks $\mathbb{X} = (x_1, x_2, \dots, x_{|\mathbb{X}|_{num}})$ and $\mathbb{Y} = (y_1, y_2, \dots, y_{|\mathbb{Y}|_{num}})$, let X and Y be multi-sets $X = \{x_1, x_2, \dots, x_{|\mathbb{X}|_{num}}\}$ and $Y = \{y_1, y_2, \dots, y_{|\mathbb{Y}|_{num}}\}$. $\mathbb{X} \sqsubseteq \mathbb{Y}$ if and only if $X \subseteq Y$.*

Proposition 5.1 implies that we can determine whether $\mathbb{P} \sqsubseteq \mathbb{T}[i : i + m - 1]$ at position i or not by checking $\{p_1, \dots, p_M\} \subseteq \{t_1[i : i + m - 1], \dots, t_N[i : i + m - 1]\}$. For convenience, we treat a multi-track as a multi-set in the following. In order to verify the multi-set inclusion relation efficiently, we use the spectral Bloom Filter (SBF) [15] described in Section 2.6. Then, we provide a definition of the FILM tree in Section 5.1.

5.1 Filtering multi-set trees

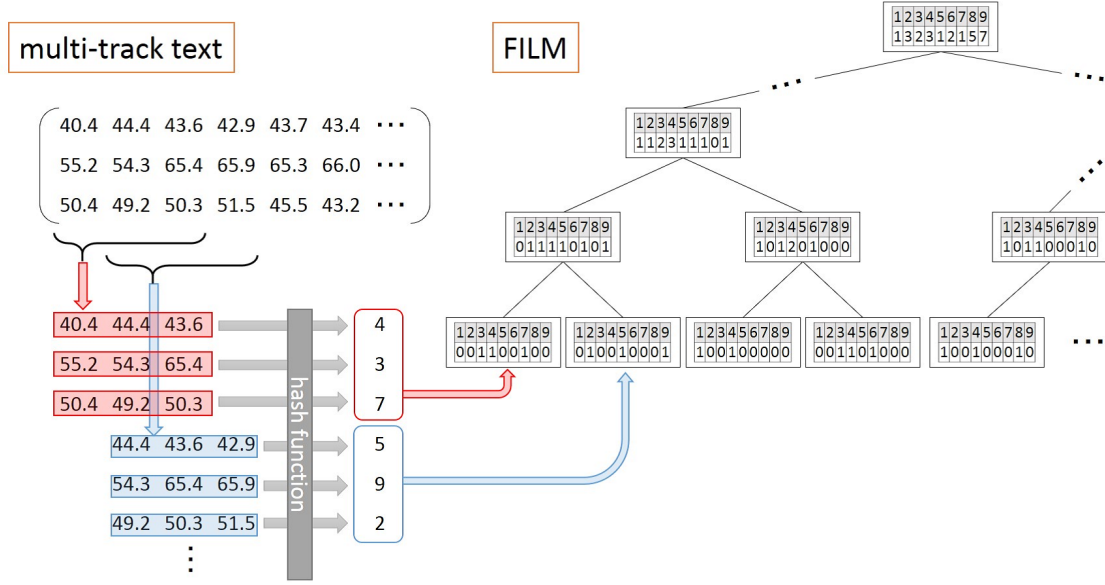


Figure 5.1: Example of a FILM tree, using only one hash function ($k = 1$).

5.1 Filtering multi-set trees

Definition 5.2 (FILM tree). Let \mathbb{T} be a multi-track of length $|\mathbb{T}|_{len} = n$, and m be a positive integer. Let $H = \{h_1, h_2, \dots, h_k\}$ be a set of k hash functions $h_i : U \rightarrow \{0, \dots, u-1\}$, where the domain U is either Σ^m or \mathcal{R}^m . A FILM tree $FILM_m(\mathbb{T})$ of size m for \mathbb{T} is a complete binary tree containing $2^{\lceil \log_2 n \rceil}$ leaves, where each node v represents an SBF of size ω defined as follows. If v is an i -th leaf node with $1 \leq i \leq n - m + 1$, then v represents $SBF_H(\mathbb{T}[i : i + m - 1])$. If $i > n - m + 1$, it represents an empty SBF, that is $(0, 0, \dots, 0)$. If v is an inner node, v represents the SBF $s_L \oplus s_R$, where s_L (resp. s_R) is the SBF represented by the left (resp. right) child of v . In the sequel, we identify a node v with the SBF that v represents.

Fig. 5.1 shows a simple example of a FILM tree. The number of nodes of a $FILM_m(\mathbb{T})$ is $2^{\lceil \log_2 n \rceil + 1} - 1 = O(n)$ because the number of leaves is $2^{\lceil \log_2 n \rceil}$ and each inner node has exactly two children. Each node represents an SBF so that it requires $O(\omega)$ space. Thus, $FILM_m(\mathbb{T})$ needs $O(n\omega)$ space. Algorithm 6 shows a construction algorithm for the FILM tree, in which the tree structure is implemented as an array A in a standard way; the

5.1 Filtering multi-set trees

Algorithm 6: FILM tree construction algorithm

Input: multi-track text \mathbb{T} and an integer m

Output: $FILM_m(\mathbb{T})$

```

1  $n = |\mathbb{T}|_{len}; N = |\mathbb{T}|_{num};$ 
2  $height = \lceil \log_2 n \rceil;$ 
3  $leafNum = 2^{height};$ 
4 for  $i = 1$  to  $n - m + 1$  do
5    $FILM_m(\mathbb{T})[leafNum + i - 1] = SBF_H(\mathbb{T}[i : i + m - 1])$ 
6 end;
7 for  $j = 1$  to  $height$  do
8    $beginNode = 2^{height-j};$ 
9    $endNode = 2 \cdot beginNode - 1;$ 
10  for  $i = beginNode$  to  $endNode$  do
11     $FILM_m(\mathbb{T})[i] = FILM_m(\mathbb{T})[2i] \oplus FILM_m(\mathbb{T})[2i + 1]$ 
12  end
13 end;
14 output  $FILM_m(\mathbb{T})$  as a FILM tree
```

left (resp. right) child of a node $A[i]$ is stored in $A[2i]$ (resp. $A[2i + 1]$). Since H is a set of k functions, it requires $O(knN)$ time to compute $SBF_H(\mathbb{T}[i : i + m - 1])$ for $1 \leq i \leq n - m + 1$ in Line 4 to Line 5. In addition, a calculation in Line 11 needs $O(\omega)$ time and is executed $2^{\lceil \log_2 n \rceil} - 1 = O(n)$ times. Thus, Algorithm 6 runs in $O(knN + n\omega)$ time.

By using $FILM_m(\mathbb{T})$, we can efficiently solve both the full- and sub- permuted pattern matching problems for a multi-track pattern \mathbb{P} of length $|\mathbb{P}|_{len} = m$. Algorithm 7 shows the matching algorithm. At Line 2, it computes a *query filter* $QF = SBF_H(\mathbb{P})$ for \mathbb{P} using the same set H of hash functions. All positions i satisfying $\mathbb{P} \stackrel{\cong}{\subseteq} \mathbb{T}[i : i + m - 1]$ can be found by a depth first search of the FILM tree, defined as the recursive function DFS in Line 4. When the algorithm visits c -th node v in the search, if $\min(FILM_m(\mathbb{T}[c]) \ominus QF) \geq 0$ and v is a leaf (Line 6 and 7), then the algorithm outputs $c - \mathbf{leafNum} + 1$ as a candidate of permuted-matching position. The time complexity of Algorithm 7 is $O(kmN + n\omega)$ time, because QF and \mathbf{sub}_{\min} are computed in $O(kmN)$ and $O(\omega)$ time respectively and DFS is executed at most $2^{\lceil \log_2 n \rceil + 1} - 1$ times.

5.1 Filtering multi-set trees

Algorithm 7: Permuted pattern matching algorithm using $FILM_m(\mathbb{T})$ ($|\mathbb{T}|_{len} = n$)

Input: pattern \mathbb{P} satisfying $|\mathbb{P}|_{len} = m$
Output: matching positions of \mathbb{P} in \mathbb{T} */* it may contains some false positives */*

```

1 leafNum =  $2^{\lceil \log_2 n \rceil}$ ;
2  $QF = SBF_H(\mathbb{P})$ ;
3  $DFS(1)$ ;      /* start depth-first-search from the root node */
4 Function  $DFS(c)$ 
5    $sub_{min} = \min(FILM_m(\mathbb{T})[c] \ominus QF)$ ;
6   if  $sub_{min} \geq 0$  then
7     if  $c \geq \text{leafNum}$  then      /*  $FILM_m(\mathbb{T})[c]$  is a leaf node */
8       output  $c - \text{leafNum} + 1$ 
9     else      /*  $FILM_m(\mathbb{T})[c]$  is an inner node */
10       $DFS(2c)$ ;
11       $DFS(2c + 1)$ 
12    end
13  end
14 end

```

The output of Algorithm 7 contains at least all positions i satisfying $\mathbb{P} \sqsubseteq \mathbb{T}[i : i + m - 1]$ for $1 \leq i \leq n - m + 1$. The correctness is shown as follows. Let $c_l = SBF_H(\mathbb{C}_l)$ and $c_r = SBF_H(\mathbb{C}_r)$ be sibling leaves of the FILM tree computed from substrings \mathbb{C}_l and \mathbb{C}_r of multi-track \mathbb{T} . Assume that \mathbb{P} permuted-matches \mathbb{C}_l , that is $\mathbb{P} \subseteq \mathbb{C}_l$. Because $QF = SBF_H(\mathbb{P})$, we have $\min(c_l \ominus QF) \geq 0$ by Property 2.1 (2). Let v be the parent node of c_l and c_r . Then, $v = SBF_H(\mathbb{C}_l \cup \mathbb{C}_r)$ from Property 2.1 (1) and the definition of the FILM tree. Now, $\mathbb{P} \subseteq (\mathbb{C}_l \cup \mathbb{C}_r)$ holds, thus $\min(v \ominus QF) \geq 0$. Recursively, $\min(u \ominus QF) \geq 0$ holds for all ancestor nodes u of c_l . Therefore, all positions i satisfying $\mathbb{P} \sqsubseteq \mathbb{T}[i : i + m - 1]$ can be found correctly by the depth first search of the FILM tree. Remark that the output of this algorithm may contain some false positives with small probability, as we have already mentioned in Section 2.6. Thus, to obtain the correct matching positions, we have to verify whether $\mathbb{P} \sqsubseteq \mathbb{T}[i : i + m - 1]$ holds or not for each candidate position i , in a naive way.

5.2 Selection of hash function

By selecting suitable hash functions for the SBF, FILM trees can be adopted to solve various permuted pattern matching problems.

In exact permuted pattern matching for multi-track string, we need a hash function for strings in order to consider a FILM tree for strings. We adopt a simple rolling hash that was used in Karp-Rabin algorithm [30] described in Section 2.7.1. For a multi-track string $\mathbb{T} = \{t_1, t_2, \dots, t_N\}$, we can calculate the hash value $h_{rh}(t_i[j : j + m - 1])$ in $O(1)$ time after calculating the value $h_{rh}(t_i[j - 1 : j + m - 2])$, because the following equation holds for any position j in a text t_i ;

$$h_{rh}(t_i[j : j + m - 1]) = a \cdot h_{rh}(t_i[j - 1 : j + m - 2]) + t_i[j + m - 1] \pmod{q},$$

where a is a prime number and q is a positive integer.

In the approximate permuted pattern matching, if each Euclidean distance between tracks of the text and the pattern is small, the multi-track Euclidean distance is small. Based on this idea, we select the locality sensitive hashing (LSH) for the Euclidean distance. By using LSH functions h_{lsh} , we can directly construct a FILM tree for numerical data, without converting them into strings. During the construction, we need to prepare some hash functions. The number of hash functions is determined by the expected collision probability. The construction time of *FILMtree* depends on the number of hash functions.

Chapter 6

Experiments

We conducted computational experiments to evaluate our algorithms. Our experiments are partitioned into four parts. In each part, we evaluate

- exact permuted pattern matching algorithms described in Section 4.1, Section 4.2 and Section 4.3.1,
- indexing structures for the full-permuted pattern matching described in Section 4.3.1, Section 4.3.2 and Section 4.3.3,
- approximate permuted pattern matching algorithms, described in Chapter 5.
- search times of indexing structures and FILM tree for the exact full-permuted pattern matching.

6.1 Exact permuted pattern matching algorithms

We show the experimental results for the exact permuted pattern matching algorithms, by using the generalized suffix array (GSA), the AC-automaton (ACAM) and the multi-track suffix tree (MTST). We implemented these algorithms in C++ and used Linux Debian wheezy with Intel© Xeon CPU E5-2609 2.40GHz and RAM 256GB throughout the experiments. We focus on the preprocessing time (denote “build”), the search time (denote

6.1 Exact permuted pattern matching algorithms

“search”), and the total time (denote “total”) of solving a permuted pattern matching. The total time is a sum of the build time and the search time. Note that, the algorithm by using GSA does not execute any preprocessing actually. However, this algorithm needs to construct some data structures to perform the pattern matching, thus we show the time required to do it as the build time. In each problem, we generated a text and a pattern on a binary alphabet randomly. We got the average time of ten computations.

Table 6.1 presents the results for various text length n and fixed parameters $N = 1000$, $m = 10$ and $M = 1000$, where m is the length of the multi-track pattern, N and M are the number of tracks of the text and the pattern. 0.000 means that the time is less than 0.001 seconds. In Table 6.1, MTST achieves fastest searches for any n . By contrast, GSA is too slowly compared with ACAM and MTST.

Table 6.2 shows the results for various N , with $n = 100000$, $m = 10$ and $M = N$. The total times of ACAM for $N \leq 32$ are less than that of MTST. Thus, ACAM is suitable to be applied for the case that the track count of the multi-track text is small.

We next show the results with respect to the length of the multi-track pattern. Table 6.3 represents the running times for various m and fixed $n = 100000$ and $N = M = 1000$. From Table 6.3, we can confirm that GSA and MTST are not influenced by the length of the pattern m greatly. On the other hand, ACAM strongly depends on m . The larger the size of the pattern is, the larger the size of the AC automaton. Thus, the build time of ACAM increases. By contrast, the reason of the increase of the search time is considered as the increase of calls of failure functions. The failure function are called recursively when a state transition fails during a traversal of an AC automaton. If m is large, the number of failures of state transitions is large and the search time also becomes large. Thus, the results shown in Table 6.3 were obtained.

Table 6.4 show the results for M , where $n = 100000$, $N = 1000$ and $m = 10$. Note that, we do not show the running times of MTST in Table 6.4 because MTST is a data structure only for the full-permuted pattern matching. ACAM can be applied for the

6.1 Exact permuted pattern matching algorithms

Table 6.1: Running times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$

n	GSA			ACAM			MTST		
	build	search	total	build	search	total	build	search	total
10000	3.505	8.237	11.742	0.000	2.370	2.370	1.153	0.000	1.153
20000	8.900	17.853	26.753	0.000	4.802	4.802	2.502	0.000	2.502
30000	14.738	28.037	42.775	0.000	7.196	7.196	3.819	0.002	3.821
40000	21.210	38.805	60.015	0.002	9.700	9.702	5.403	0.000	5.403
50000	27.796	49.296	77.092	0.000	12.119	12.119	6.963	0.000	6.963
60000	34.696	60.482	95.178	0.000	14.538	14.538	9.479	0.001	9.480
70000	42.388	73.184	115.572	0.000	16.944	16.944	10.686	0.001	10.687
80000	49.755	84.058	133.813	0.000	19.302	19.302	13.255	0.000	13.255
90000	57.644	96.377	154.021	0.002	21.754	21.756	13.832	0.001	13.833
100000	65.790	107.321	173.111	0.003	24.390	24.393	15.125	0.002	15.127

Table 6.2: Running times (sec) for N with $n = 100000$, $m = 10$, and $M = N$

N	GSA			ACAM			MTST		
	build	search	total	build	search	total	build	search	total
1	0.021	0.040	0.061	0.000	0.015	0.015	0.429	0.000	0.429
2	0.036	0.091	0.127	0.000	0.029	0.029	0.464	0.001	0.465
4	0.070	0.201	0.271	0.000	0.069	0.069	0.506	0.001	0.507
8	0.155	0.422	0.577	0.005	0.151	0.156	0.535	0.001	0.536
16	0.353	0.982	1.335	0.004	0.317	0.321	0.647	0.001	0.648
32	0.826	2.181	3.007	0.000	0.702	0.702	0.754	0.000	0.754
64	1.949	4.893	6.842	0.000	1.461	1.461	0.944	0.000	0.944
100	3.647	8.085	11.732	0.000	2.371	2.371	1.261	0.000	1.261
200	9.232	17.864	27.096	0.000	4.971	4.971	1.941	0.001	1.942
300	15.269	27.687	42.956	0.001	7.480	7.481	2.723	0.000	2.723
400	21.831	39.125	60.956	0.001	10.145	10.146	3.434	0.001	3.435
500	28.514	49.967	78.481	0.000	12.664	12.664	4.453	0.000	4.453
600	35.495	60.265	95.760	0.003	15.328	15.331	6.670	0.000	6.670
700	44.018	73.296	117.314	0.000	17.786	17.786	9.300	0.000	9.300
800	50.433	84.327	134.760	0.002	19.958	19.960	11.281	0.000	11.281
900	58.031	96.224	154.255	0.000	22.143	22.143	13.293	0.000	13.293
1000	65.784	106.951	172.735	0.002	24.400	24.402	15.177	0.002	15.179

6.1 Exact permuted pattern matching algorithms

Table 6.3: Running times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$

m	GSA			ACAM			MTST		
	build	search	total	build	search	total	build	search	total
1	65.778	110.133	175.911	0.000	4.055	4.055	15.168	0.002	15.170
2	65.774	107.640	173.414	0.001	5.012	5.013	15.105	0.000	15.105
4	65.766	105.987	171.753	0.001	6.262	6.263	15.331	0.002	15.333
8	65.648	106.961	172.609	0.000	15.158	15.158	15.304	0.002	15.306
10	65.811	107.144	172.955	0.001	24.358	24.359	15.207	0.001	15.208
20	65.826	108.693	174.519	0.004	37.848	37.852	15.093	0.000	15.093
30	65.750	108.493	174.243	0.013	38.880	38.893	15.201	0.000	15.201
40	65.883	108.854	174.737	0.015	39.694	39.709	15.179	0.001	15.180
50	65.679	108.238	173.917	0.021	40.153	40.174	15.024	0.003	15.027
60	65.720	107.527	173.247	0.028	40.509	40.537	15.152	0.002	15.154
70	65.615	108.325	173.940	0.031	41.193	41.224	15.208	0.000	15.208
80	65.743	108.023	173.766	0.039	41.391	41.430	15.330	0.007	15.337
90	65.715	108.410	174.125	0.044	41.705	41.749	15.342	0.005	15.347
100	65.658	108.439	174.097	0.050	41.760	41.810	15.359	0.005	15.364

Table 6.4: Running times (sec) for M with $n = 100000$, $N = 1000$, and $m = 10$

M	GSA			ACAM			MTST		
	build	search	total	build	search	total	build	search	total
1	65.744	69.612	135.356	0.000	5.329	5.329	-	-	-
2	65.763	93.372	159.135	0.000	6.717	6.717	-	-	-
4	65.733	101.742	167.475	0.000	8.071	8.071	-	-	-
8	65.741	104.066	169.807	0.000	9.402	9.402	-	-	-
16	65.756	104.793	170.549	0.000	11.474	11.474	-	-	-
32	65.729	105.207	170.936	0.000	13.908	13.908	-	-	-
64	65.753	105.637	171.390	0.000	16.341	16.341	-	-	-
100	65.740	105.580	171.320	0.000	17.907	17.907	-	-	-
200	65.786	106.601	172.387	0.002	20.545	20.547	-	-	-
300	65.716	107.581	173.297	0.000	21.586	21.586	-	-	-
400	65.635	107.732	173.367	0.000	22.836	22.836	-	-	-
500	65.792	105.810	171.602	0.001	23.225	23.226	-	-	-
600	65.802	107.544	173.346	0.001	23.692	23.693	-	-	-
700	65.653	107.260	172.913	0.000	24.027	24.027	-	-	-
800	65.808	107.356	173.164	0.000	24.438	24.438	-	-	-
900	65.797	107.492	173.289	0.001	24.450	24.451	-	-	-
1000	65.815	107.223	173.038	0.002	24.319	24.321	15.177	0.000	15.177

6.2 Indexing structures

sub-permuted pattern matching different from MTST and is faster than GSA for any M . The search time of ACAM increases by M . This is because M accepting states of the AC automaton are accessed to determine whether the pattern permuted-matches at a position in the text or not.

6.2 Indexing structures

Next we show the experimental results of comparing the indexing structures for the multi-track string: the multi-track suffix tree (MTST), the multi-track position heap (MTPH), the contracted multi-track position heap (CMTPH), the augmented multi-track position heap (AMTPH), and the augmented contracted multi-track position heap (ACMTPH). We implemented the matching algorithms by using these data structures in C++. We used Linux Debian wheezy with Intel® Xeon CPU E5-2609 2.40GHz and RAM 256GB throughout the experiments in this section.

In the experiment, we focus on the construction time (denote “build”), the search time (denote “search”), and the required memory of each structure. We got the average time of ten computations. In each computation, we generated a random text and a random pattern on the binary alphabet, and embed the pattern in the text at 50 times with no overlaps randomly. Note that, in this experiment, CMTPH was constructed in a naive way: $NMA(v)$ and $LevA(v, d)$ for any node v need $O(nN)$ time in our implementation, so that the construction of CMTPH requires $O(n^2N \log |\Sigma|)$ time.

First we show the experimental results for various n and fixed $N = 1000$, $m = 10$ and $M = 1000$ in Table 6.5 and Table 6.6, where n and m are the length of the multi-track text and the multi-track pattern, and N and M are the number of tracks of the text and the pattern. As shown in Table 6.5, MTST achieves the fastest construction and search for any n . The searches by using CMTPH and ACMTPH are faster than that by using MTPH and AMTPH experimentally as shown in Table 6.5. Surprisingly, the constructions of CMTPH and ACMTPH are also faster than that of MTPH and AMTPH

6.2 Indexing structures

Table 6.5: Running times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$

n	MTST		MTPH		AMTPH		CMTPH		ACMTPH	
	build	search	build	search	build	search	build	search	build	search
10000	1.134	0.001	4.948	0.811	6.567	0.798	5.104	0.377	6.083	0.366
20000	2.493	0.001	10.795	0.831	13.913	0.800	10.928	0.455	12.943	0.449
30000	3.854	0.001	16.455	0.816	21.387	0.797	16.366	0.577	19.492	0.566
40000	5.344	0.000	24.053	0.813	30.962	0.799	23.599	0.603	27.981	0.598
50000	6.923	0.000	30.192	0.833	38.627	0.800	29.360	0.599	34.677	0.592
60000	8.380	0.001	36.543	0.848	47.758	0.796	35.653	0.639	42.126	0.623
70000	10.100	0.000	46.817	0.870	59.068	0.801	43.879	0.659	51.895	0.641
80000	12.106	0.001	53.449	0.866	68.509	0.801	49.754	0.637	59.032	0.627
90000	13.280	0.000	61.568	0.865	78.881	0.802	57.235	0.708	67.564	0.684
100000	15.096	0.000	68.508	0.865	88.347	0.800	63.498	0.706	75.270	0.681

Table 6.6: Required memory (MBytes) for n with $N = 1000$, $m = 10$, and $M = 1000$

n	MTST	MTPH	AMTPH	CMTPH	ACMTPH
10000	38.715	148.545	309.919	0.353	0.695
20000	77.465	274.185	565.385	0.706	1.408
30000	116.244	411.874	853.641	1.059	2.129
40000	155.046	563.900	1174.621	1.411	2.775
50000	193.857	726.204	1517.302	1.764	3.481
60000	232.678	897.812	1881.761	2.117	4.187
70000	271.503	1078.569	2223.475	2.470	4.897
80000	310.328	1266.892	2617.281	2.823	5.613
90000	349.153	1459.432	3019.903	3.176	6.328
100000	387.975	1659.675	3438.631	3.529	7.044

for $n \geq 30000$. This is caused by constructing a tree that consists only of indexing nodes, which is traversed to enumerate all matching positions in $O(occ)$ time. Focusing on the required space, CMTPH is most memory-efficient as shown in Table 6.6.

We next show the results for $N = 1, 2, 4, \dots, 1000$ with $n = 100000$, $m = 10$ and $M = N$ in Table 6.7 and Table 6.8. For small N , i.e., $N \leq 32$, all methods except for CMTPH achieve very fast search time as seen in Table 6.7. In addition, MTPH can be constructed faster than any other structure for $N = 1, 2, 4, 8, 16$. From Table 6.8, we can confirm that CMTPH and ACMTPH do not depend on the track count of the text N .

Table 6.9 and Table 6.10 show the results for $m = 1, 2, 4, \dots, 100$ with $n = 100000$,

6.2 Indexing structures

Table 6.7: Running times (sec) for N with $n = 100000$, $m = 10$, and $M = N$

N	MTST		MTPH		AMTPH		CMTPH		ACMTPH	
	build	search	build	search	build	search	build	search	build	search
1	0.427	0.001	0.117	0.000	0.167	0.001	0.171	0.000	0.236	0.000
2	0.451	0.000	0.165	0.000	0.260	0.000	0.234	0.004	0.342	0.000
4	0.505	0.001	0.262	0.000	0.426	0.000	0.351	0.004	0.495	0.000
8	0.582	0.000	0.376	0.000	0.638	0.000	0.492	0.004	0.704	0.000
16	0.685	0.000	0.641	0.000	1.083	0.001	0.773	0.005	1.076	0.001
32	0.816	0.000	1.107	0.001	1.778	0.001	1.264	0.004	1.731	0.001
64	1.038	0.000	2.271	0.005	3.522	0.003	2.321	0.011	3.088	0.008
100	1.382	0.001	3.951	0.011	5.934	0.011	3.734	0.017	4.795	0.011
200	2.068	0.000	9.454	0.037	13.487	0.036	8.262	0.045	10.184	0.050
300	2.792	0.000	16.560	0.082	22.675	0.073	14.745	0.102	17.616	0.074
400	3.523	0.000	22.834	0.141	31.279	0.127	20.797	0.180	24.776	0.123
500	4.565	0.000	29.896	0.219	40.457	0.200	26.898	0.270	31.933	0.196
600	6.968	0.000	39.723	0.324	52.022	0.290	36.212	0.326	42.622	0.271
700	9.504	0.000	47.482	0.423	62.622	0.400	43.019	0.404	51.111	0.355
800	11.235	0.001	53.952	0.554	70.167	0.509	49.496	0.501	58.948	0.459
900	13.456	0.003	61.448	0.701	79.211	0.647	56.540	0.603	67.293	0.560
1000	15.263	0.000	68.440	0.864	87.440	0.801	63.176	0.727	74.516	0.697

Table 6.8: Required memory (MBytes) for N with $n = 100000$, $m = 10$, and $M = N$

N	MTST	MTPH	AMTPH	CMTPH	ACMTPH
1	11.062	3.147	7.051	3.528	7.051
2	9.439	5.368	11.543	3.528	7.051
4	8.914	9.107	19.454	3.528	7.052
8	9.414	14.773	31.602	3.529	7.052
16	12.555	31.551	66.150	3.529	7.052
32	18.173	50.389	106.206	3.529	7.048
64	30.042	117.482	244.640	3.529	7.048
100	43.848	206.041	432.287	3.529	7.048
200	82.282	447.861	929.875	3.529	7.047
300	120.654	664.794	1387.846	3.529	7.046
400	158.959	854.735	1790.100	3.529	7.045
500	197.212	1022.678	2148.040	3.529	7.044
600	235.419	1172.711	2420.431	3.529	7.044
700	273.588	1308.124	2703.568	3.529	7.044
800	311.736	1433.092	2964.866	3.529	7.044
900	349.862	1549.510	3208.285	3.529	7.043
1000	387.975	1659.675	3438.631	3.529	7.044

6.2 Indexing structures

Table 6.9: Running times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$

m	MTST		MTPH		AMTPH		CMTPH		ACMTPH	
	build	search	build	search	build	search	build	search	build	search
1	15.042	0.002	68.424	0.084	87.664	0.079	63.171	0.345	74.632	0.125
2	15.115	0.000	68.408	0.167	88.300	0.159	63.128	0.455	75.384	0.428
4	15.045	0.000	67.988	0.336	87.889	0.319	62.903	0.550	74.815	0.515
8	15.306	0.000	67.968	0.686	87.115	0.643	62.777	0.645	74.403	0.630
10	14.918	0.000	69.972	0.863	88.927	0.798	63.371	0.707	74.265	0.676
20	15.430	0.001	68.371	2.014	87.712	1.707	63.730	1.089	75.400	1.090
30	15.199	0.001	68.343	3.058	87.697	2.611	64.172	1.500	75.525	1.458
40	15.379	0.000	68.032	3.939	87.446	3.567	63.517	1.897	75.913	1.910
50	15.135	0.002	68.050	5.126	87.595	4.599	64.628	2.195	76.464	2.092
60	15.057	0.002	68.318	5.136	87.403	4.808	64.874	2.839	77.306	2.917
70	15.071	0.003	69.378	5.307	89.270	5.087	64.852	3.170	78.134	3.033
80	15.157	0.002	68.155	5.521	87.784	5.386	65.556	3.539	79.019	3.575
90	15.124	0.004	67.967	5.637	87.608	5.489	65.442	3.733	78.884	3.754
100	15.328	0.006	68.052	5.664	87.606	5.573	65.870	4.174	79.389	4.255

Table 6.10: Required memory (MBytes) for m with $n = 100000$, $N = 1000$, and $M = 1000$

m	MTST	MTPH	AMTPH	CMTPH	ACMTPH
1	387.980	1655.200	3429.273	3.529	7.043
2	387.980	1655.327	3429.539	3.529	7.044
4	387.977	1655.738	3430.398	3.529	7.044
8	387.976	1658.308	3435.773	3.529	7.044
10	387.975	1659.675	3438.631	3.529	7.044
20	387.970	1664.982	3449.727	3.529	7.044
30	387.961	1669.730	3459.654	3.529	7.044
40	387.956	1675.368	3471.443	3.529	7.044
50	387.950	1680.374	3481.911	3.529	7.044
60	387.943	1685.683	3493.011	3.529	7.044
70	387.936	1691.718	3505.629	3.529	7.044
80	387.928	1696.643	3515.928	3.529	7.044
90	387.925	1702.988	3529.195	3.529	7.044
100	387.920	1708.673	3541.082	3.529	7.043

6.3 Approximate permuted pattern matching algorithms

$N = 1000$ and $M = 1000$. As expected, the construction times of all methods do not change greatly and the search times increase by the length of the pattern. (see Table 6.9). As shown in Table 6.10, the required spaces do not increase greatly.

6.3 Approximate permuted pattern matching algorithms

We performed three sets of experiments. In the first two of them, we assessed the construction time and search time of our algorithm on random data. In the third one, we performed experiments on real-world data. Throughout the experiments, we used a Linux machine with a 2.4GHz Intel© Xeon CPU EE5-2609 and 256GB RAM, running Debian 7.0. In the experiments on random data, we used the following basic parameter values: the length of a text $|\mathbb{T}|_{len} = 100000$, the number of tracks of a text $|\mathbb{T}|_{num} = 1000$, the length of a pattern $|\mathbb{P}|_{len} = 300$, the alphabet size of a multi-track string text $|\Sigma| = 26$, the number of hash functions $k = 1$ and the size of a SBF used in a FILM tree $\omega = 10000$.

In the experiments, we compared our algorithms using the FILM tree with a multi-track suffix tree (MTST). However, MTST can be applied only to multi-track strings. Thus, we prepared numerical data for the FILM tree and string data for MTST converted from the numerical data in the following manner; the value range of numerical data is divided into equal $|\Sigma|$ parts, and each divided part is assigned to a distinct character. Note that the construction time of MTST excludes the time of this conversion.

A naive implementation of SBF in Algorithm 6 and Algorithm 7 would be an integer array. Instead of it, we used an *associative array* in the experiments, because SBFs are very sparse, that means most elements are 0's, especially if they are near to the leaves. By storing only non-zero elements in the associative array, we can greatly reduce the memory requirement of SBFs. Moreover, the operations $u \oplus v$ and $u \ominus v$ can be computed efficiently, that depends only on the number of non-zero elements, but not the size ω .

6.3 Approximate permuted pattern matching algorithms

6.3.1 Construction time on random data

The first set of our experiments assesses the construction time of FILM trees on random data. We varied the values of $|\mathbb{T}|_{len}$, $|\mathbb{T}|_{num}$ and ω , and compared the construction time of FILM trees using rolling hash (RH) and LSH with that of multi-track suffix trees (MTSTs). In this experiment, we used random numerical data for FILM_{LSH} and the string data converted from the numerical data for FILM_{RH} and MTST.

Fig. 6.1 shows the results. In this figure, the y -axes represent the construction times (seconds), and they are on logarithmic scales in Fig. 6.1(b) and Fig. 6.1(c). The x -axes represent the length $|\mathbb{T}|_{len}$ of the multi-track text, the number $|\mathbb{T}|_{num}$ of tracks, and the size ω of SBF in FILM tree in Fig. 6.1(a), Fig. 6.1(b) and Fig. 6.1(c), respectively. In all experiments, MTST was the fastest among them, and FILM trees were much slower than MTST. The construction time of FILM tree depended on $|\mathbb{T}|_{len}$ and ω .

6.3.2 Search time on random data

The experiments in the second set concern with the search time on random data. The search using the FILM tree depends on the size ω of SBF, and the height of the FILM tree that reflects the length $|\mathbb{T}|_{len}$ of text. On the other hand, the search using MTST depends on the length $|\mathbb{P}|_{len}$ of pattern. Thus, we compared the search time of FILM_{LSH} , FILM_{RH} and MTST for various values of $|\mathbb{T}|_{len}$, $|\mathbb{P}|_{len}$ and ω .

Fig. 6.2 shows the results. The y -axes represent the search times (seconds) on logarithmic scales. The x -axis in Fig. 6.2(a) represents the length $|\mathbb{T}|_{len}$ of the multi-track text. The matching algorithm using the FILM tree needs to search from the root node to leaf nodes in order to identify the matching positions. The search time depends on the height of the FILM tree, which is $O(\log_2 n)$ with respect to the text length $n = |\mathbb{T}|_{len}$. The result shows that we can ignore this influence from a practical viewpoint. The x -axis in Fig. 6.2(b) represents the length $|\mathbb{P}|_{len}$ of the pattern. Concerning with MTST, as we expected, the search time increases as the pattern \mathbb{P} becomes longer, because we

6.3 Approximate permuted pattern matching algorithms

have to traverse a path of length $|\mathbb{P}|_{len}$ in MTST. If we implement SBF in the FILM tree with a normal array, the search time does not depend on the length of the pattern in principle. However, we had confirmed in a preliminary experiment that the search time of this implementation was very slow and required large memory, compared to MTST. Thus, we decided to use associative arrays for SBFs. In consequence, the search time of FILM tree is faster than that of MTST, although it mildly depends on the pattern length $|\mathbb{P}|_{len}$. The x -axis in Fig. 6.2(c) represents the size ω of SBF. If we use normal arrays for SBFs, the search using FILM tree would slow down as ω increases. However, thanks to the associative array implementation, we obtained much faster search, that is practically independent of ω .

All results in these experiments show that our proposed method using FILM tree outperforms MTST on search time in any cases.

6.3.3 Construction time and search time on traffic data

The third set is the experiments for real-world data. We used some traffic data comprised of car speed measurements at 702 monitoring points on Tokyo Metropolitan Expressway in Japan. We regarded a time series of them as a multi-track numerical text \mathbb{T} with $|\mathbb{T}|_{num} = 702$. At each monitoring point, the average speed of the cars were recorded at every 5 minutes for one year, so that the length $|\mathbb{T}|_{len}$ was 105120.

As typical applications on the traffic data, we are interested in various subjects; for instance, detecting traffic jams, finding some common patterns in them, and extracting some relations among them, with respect to the time, week, and month, and so on. In the most of all these processing, pattern matching is indispensable as a fundamental operation. Therefore, we evaluated the real performance of our proposed method on these data for pattern matching. We examined four lengths of patterns, $|\mathbb{P}|_{len} = m = 288$ (one day record), 144 (12 hours), 72 (6 hours), and 36 (3 hours), and each pattern was randomly cut out from the text $|\mathbb{T}|_{len}$. As is the previous subsection, we fixed $|\Sigma| = 26$ and $k = 1$,

6.4 Comparison of search times of indexing structures and FILM tree

and varied the size ω of SBF.

Fig. 6.3(a) shows the construction time, and Fig. 6.3(b) shows the search time, both in logarithmic scales. We observe that the tendency of the performance is similar to the one for the random data; although construction of FILM tree is slower than that of MTST, searching using FILM tree is much faster in most situations, and the choice of the size ω does not affect the running time very much. We conclude that our proposed method provides an efficient way to support pattern matching on multi-tracks of this amount of numerical data.

6.4 Comparison of search times of indexing structures and FILM tree

We show the experimental results for comparison indexing structures, the multi-track suffix tree (MTST), the multi-track position heap (MTPH), and the contracted multi-track position heap (CMTPH), with FILM tree (FILM) for the full-permuted pattern matching on multi-track strings in this section. To evaluate the performance of FILM tree for the non-probabilistic search, we also implemented an algorithm by using FILM tree that includes the verification of output positions of the search (FILMexact). We implemented each algorithm in C++ and used Linux Debian wheezy with Intel© Xeon CPU E5-2609 2.40GHz and RAM 256GB throughout the experiments in this section.

We only focus on the search time of each data structure in this section. We computed the average time of ten runs that used a random text and a random pattern on the binary alphabet. The pattern was embed in the text at 50 times with no overlaps randomly. Note that, a implementation of SBF in FILM tree was an integer array different from an associative array in Section 6.3.

Table 6.11, Table 6.12, and Table 6.13 shows the search times for various n and fixed $N = 1000$, $m = 10$ and $M = 1000$, for various N with $n = 100000$, $m = 10$ and $M = N$,

6.4 Comparison of search times of indexing structures and FILM tree

Table 6.11: Search times (sec) for n with $N = 1000$, $m = 10$, and $M = 1000$

n	MTST	MTPH	CMTPH	FILM	FILMexact
10000	0.000	0.817	0.374	0.010	0.038
20000	0.000	0.834	0.454	0.019	0.046
30000	0.001	0.827	0.574	0.030	0.056
40000	0.000	0.819	0.604	0.039	0.066
50000	0.000	0.857	0.601	0.047	0.073
60000	0.001	0.864	0.640	0.059	0.085
70000	0.000	0.879	0.658	0.067	0.094
80000	0.000	0.890	0.640	0.074	0.102
90000	0.001	0.871	0.716	0.078	0.109
100000	0.001	0.879	0.715	0.098	0.122

and for various m with $n = 100000$, $N = 1000$ and $M = 1000$, respectively. MTST achieves the fastest search almost for any parameter. On the other hand, FILM gives the second fastest search. The search speed of FILM can easily be accelerated by adopting an implementation of SBF as an associative array as shown in Section 6.3. In addition, FILM also has the merit that it can be applied for the sub-permuted pattern matching. As shown in the results of FILMexact, the performance of the search by using FILM tree is good compared with that of MTPH and CMTPH even if the search includes the verification of output positions. However, it is expected that if occurrences of the pattern in the text increase, the search speed of FILMexact may be worse because the verification cost becomes larger.

6.4 Comparison of search times of indexing structures and FILM tree

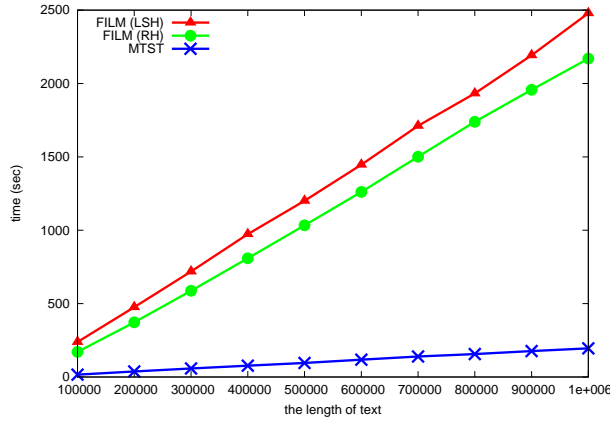
Table 6.12: Search times (sec) for N with $n = 100000$, $m = 10$, and $M = N$

N	MTST	MTPH	CMTPH	FILM	FILMexact
1	0.001	0.000	0.001	0.007	0.009
2	0.001	0.000	0.002	0.003	0.003
4	0.000	0.000	0.004	0.004	0.001
8	0.000	0.000	0.005	0.004	0.006
16	0.000	0.000	0.004	0.004	0.004
32	0.000	0.001	0.006	0.010	0.009
64	0.000	0.006	0.009	0.012	0.016
100	0.000	0.010	0.014	0.017	0.016
200	0.000	0.033	0.043	0.027	0.031
300	0.000	0.087	0.100	0.034	0.043
400	0.000	0.144	0.186	0.046	0.057
500	0.000	0.220	0.270	0.057	0.067
600	0.001	0.321	0.326	0.061	0.079
700	0.000	0.437	0.395	0.071	0.088
800	0.000	0.563	0.508	0.077	0.098
900	0.000	0.724	0.607	0.084	0.109
1000	0.003	0.890	0.715	0.094	0.123

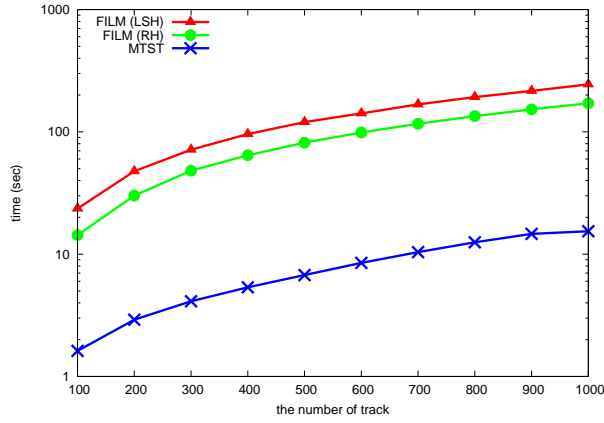
Table 6.13: Search times (sec) for m with $n = 100000$, $N = 1000$, and $M = 1000$

m	MTST	MTPH	CMTPH	FILM	FILMexact
1	0.000	0.087	0.343	0.218	0.719
2	0.001	0.168	0.460	0.305	0.317
4	0.000	0.339	0.548	0.319	0.334
8	0.001	0.695	0.649	0.205	0.231
10	0.001	0.891	0.720	0.095	0.122
20	0.001	2.062	1.094	0.060	0.104
30	0.000	3.068	1.497	0.058	0.115
40	0.002	3.960	1.917	0.061	0.140
50	0.002	5.561	2.310	0.061	0.160
60	0.005	5.131	2.837	0.058	0.197
70	0.002	5.319	3.149	0.060	0.184
80	0.002	5.517	3.505	0.061	0.202
90	0.007	5.650	3.728	0.059	0.247
100	0.006	5.698	4.210	0.059	0.233

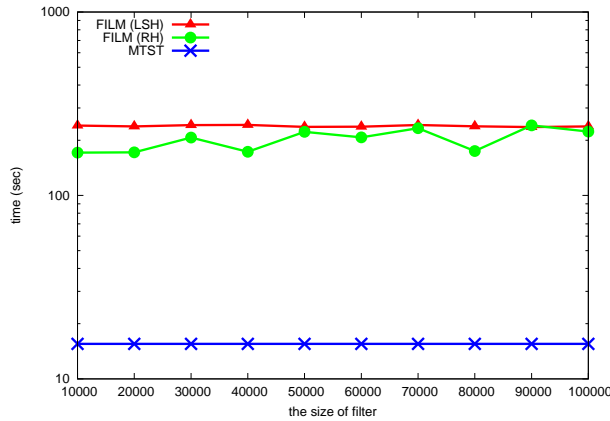
6.4 Comparison of search times of indexing structures and FILM tree



(a) The length $|\mathbb{T}|_{len}$ of text varied from 100000 to 1000000, while the other parameters were fixed as $|\mathbb{T}|_{num} = 1000$, $m = |\mathbb{P}|_{len} = 300$, and $\omega = 10000$.



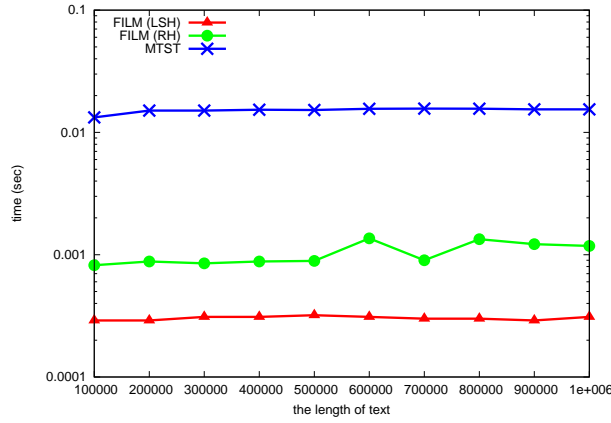
(b) The number $|\mathbb{T}|_{num}$ of tracks varied from 100 to 1000, while the other parameters were fixed as $|\mathbb{T}|_{len} = 100000$, $m = |\mathbb{P}|_{len} = 300$, and $\omega = 10000$.



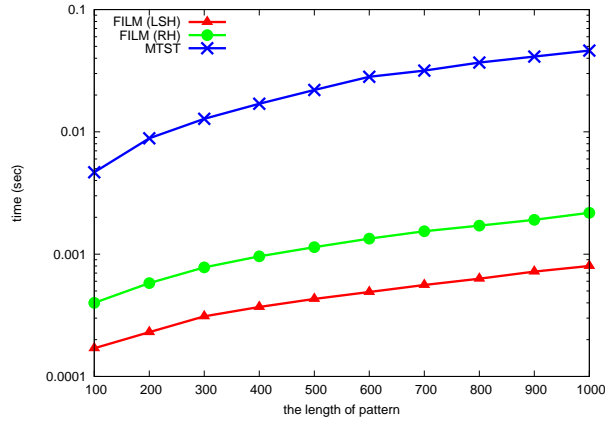
(c) The size ω of SBF varied from 10000 to 100000, while the other parameters were fixed as $|\mathbb{T}|_{len} = 100000$, $|\mathbb{T}|_{num} = 1000$ and $m = |\mathbb{P}|_{len} = 300$. Because MTST does not use SBF, the construction time of it is constant.

Figure 6.1: Comparison of the construction time on random data.

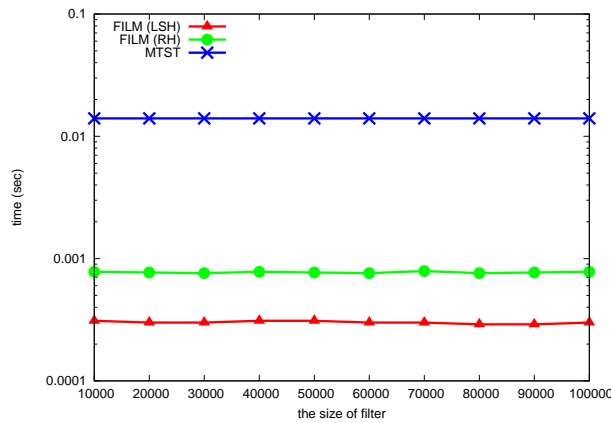
6.4 Comparison of search times of indexing structures and FILM tree



(a) The length $|\mathbb{T}|_{len}$ of the text varied from 100000 to 1000000, while the other parameters were fixed as $|\mathbb{T}|_{num} = |\mathbb{P}|_{num} = 1000$, $|\mathbb{P}|_{len} = 300$ and $\omega = 10000$.



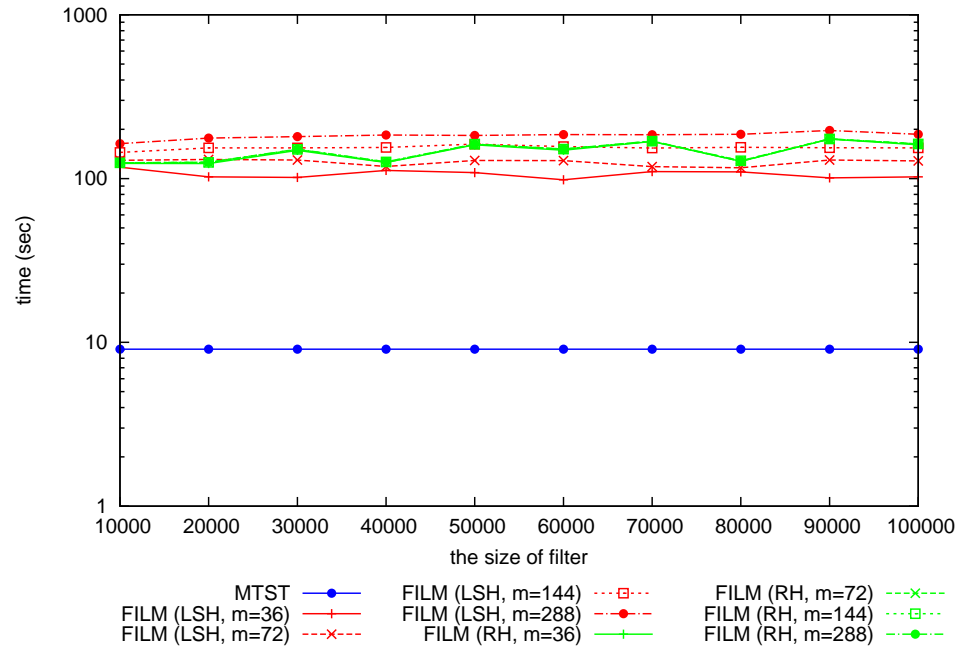
(b) The number $|\mathbb{T}|_{num}$ of tracks varied from 100 to 1000, while the other parameters were fixed as $|\mathbb{T}|_{len} = 100000$, $|\mathbb{T}|_{num} = |\mathbb{P}|_{num} = 1000$ and $\omega = 10000$.



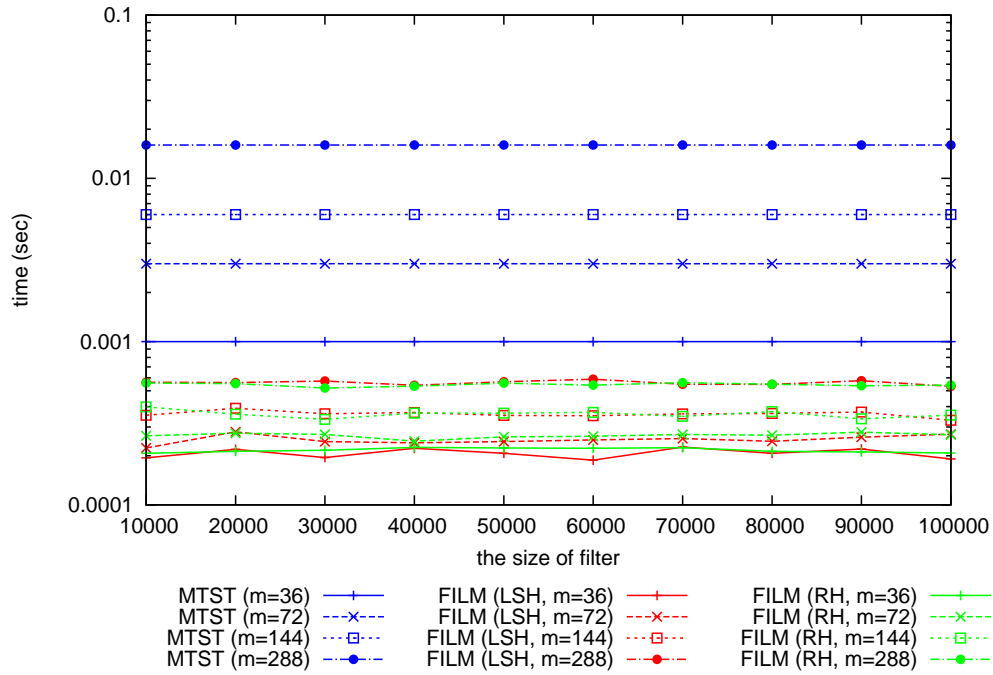
(c) The size ω of SBF varied from 10000 to 100000, while the other parameters were fixed as $|\mathbb{T}|_{len} = 100000$, $|\mathbb{T}|_{num} = |\mathbb{P}|_{num} = 1000$ and $|\mathbb{P}|_{len} = 300$. Because MTST does not use SBF, the search time of it is constant.

Figure 6.2: Comparison of the search time on random data.

6.4 Comparison of search times of indexing structures and FILM tree



(a) Construction time.



(b) Search time.

Figure 6.3: Comparisons of construction time and search time on the real traffic data. The size ω of SBF varied from 10000 to 100000, and the pattern length $m = |\mathbb{P}|_{len}$ was changed to 36, 72, 144, and 288.

Chapter 7

Conclusion

We introduced a new form of string pattern matching called the permuted pattern matching on multi-track strings. We showed that the permuted pattern matching problem on multi-track strings can be solved in $O(nN \log |\Sigma|)$ time and $O(mM + N)$ space using the AC-automaton. Furthermore, by using constant time longest common extension queries after linear time pre-processing, we can solve the problem in $O(nN)$ time and space for integer alphabets. However, these solutions do not allow a linear pre-processing of the text multi-tracks so that pattern matching cannot be performed in worst case linear time with respect to the pattern length plus output size, as do various string indices (e.g., suffix trees, suffix arrays) for normal string pattern matching.

For this problem, we proposed a new indexing structure called multi-track suffix trees (mt-suffix tree). Given the mt-suffix tree for a text multi-track, we can solve the full-permuted-matching (i.e., $M = N$) problem for any pattern multi-track in $O(mN \log |\Sigma| + occ)$ time, where *occ* is the number of positions of the text where the pattern permuted-matches. We also developed an algorithm for constructing the mt-suffix tree, based on the Ukkonen algorithm [42], running in $O(nN \log |\Sigma|)$ time and $O(nN)$ space. For constant size alphabets, the proposed algorithm performs in optimal linear time in the total size of the input texts.

We also proposed two new indexing structures, MTPH and CMTPH, for multi-track

strings, that are memory-efficient compared with the mt-suffix tree; MTPH and CMTPH need $O(nN)$ and $O(n)$ space, respectively. We showed an $O(nN \log |\Sigma|)$ -time construction algorithms of MTPH and CMTPH, and proposed MRPs for both of them. By using these data structures, the permuted pattern matching problem can be solved efficiently: $O(m^2N \log |\Sigma| + occ)$ time by MTPH, and $O(m^2N^2 \log |\Sigma| + occ)$ time by CMTPH.

The problem of developing a text index that can be used for solving sub-permuted pattern matching (i.e., $M < N$) in $O(mM \log |\Sigma| + occ)$ time is an open problem. Boyer-Moore type algorithms for permuted-matching may also be of interest for further research. To construct CMPTH directly in $O(nN)$ time without constructing MTPH is also our future work.

In addition to above results for the exact permuted pattern matching, we proposed a data structure FILM tree that also can be applied to the approximate permuted pattern matching problem. We considered some examples to demonstrate the effectiveness of this approach, such as full/sub-permuted pattern matching problems on string multi-tracks and full/sub-permuted approximate pattern matching problems on numerical multi-tracks, as well as providing their algorithms. FILM tree requires $O(n\omega)$ space, where n and ω are the lengths of the multi-track text and the size of SBF, respectively. We performed a comparison with the mt-suffix tree for the full-permuted pattern matching problem and demonstrated that FILM tree can search patterns faster than the mt-suffix tree.

FILM tree is a simple and powerful data structure for permuted pattern matching problems, but it is only suitable for fixed length patterns. Thus, we need to consider the development of a version for variable length patterns in our future research.

References

- [1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [2] Amihood Amir and Martin Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *SODA*, pages 212–223, 1991.
- [3] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutre, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.
- [4] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.
- [5] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.
- [6] Michael A. Bender and Martín Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.
- [7] Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- [8] Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

REFERENCES

- [9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
- [10] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *STOC*, pages 327–336, 1998.
- [11] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *CPM*, pages 55–69. Springer, 2003.
- [12] Robert G. Busacker and Paul J. Gowen. A procedure for determining a family of minimum cost flow networks. *Operations Research Office Technical Report*, 15, 1961.
- [13] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.
- [14] Edward G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, 1970.
- [15] Saar Cohen and Yossi Matias. Spectral bloom filters. In *SIGMOD*, pages 241–252, 2003.
- [16] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002.
- [17] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of 20th annual symposium on Computational geometry*, pages 253–262, 2004.
- [18] C Stuart Daw, Charles Edward Andrew Finney, and Eugene R Tracy. A review of symbolic analysis of experimental data. *Review of Scientific Instruments*, 74(2):915–930, 2003.
- [19] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

REFERENCES

- [20] Shiri Dori and Gad M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Information Processing Letters*, 98(2):66–72, 2006.
- [21] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 9(1):100–121, 2011.
- [22] Martin Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997.
- [23] Raffaele Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM Journal on Computing*, 24(3):520–562, 1995.
- [24] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.
- [25] Yun-Wu Huang and Philip S Yu. Adaptive query processing for time-series data. In *KDD*, pages 282–286. ACM, 1999.
- [26] Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, 2010.
- [27] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
- [28] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *ICALP*, pages 943–955, 2003.
- [29] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.
- [30] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

REFERENCES

- [31] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.
- [32] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *CPM*, pages 200–210, 2003.
- [33] Gregory Kucherov. On-line construction of position heaps. In *SPIRE*, volume 7024, pages 326–337, 2011.
- [34] N. Jesper Larsson and Kunihiro Sadakane. Faster suffix sorting. Technical Report in Department of Computer Science LU-CS-TR:99-214, Lund University, Sweden, 1999.
- [35] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, 2007.
- [36] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [37] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- [38] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *DCC*, pages 193–202, 2009.
- [39] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43–49, 1978.
- [40] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
- [41] Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, 39(1):1–19, 2004.

REFERENCES

- [42] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [43] Peter Weiner. Linear pattern matching algorithms. In *SWAT*, pages 1–11, 1973.
- [44] Jeffery Westbrook. Fast incremental planarity testing. In *ICALP*, pages 342–353, 1992.

List of Publications

Refereed Papers in Journals

1. Kazuyuki Narisawa, Takashi Katsura, Hiroyuki Ota, Ayumi Shinohara, Filtering Multi-set Tree : Data Structure for Flexible Matching Using Multi-track Data, Interdisciplinary Information Sciences.

Refereed Papers in Conferences

1. Takashi Katsura, Kazuyuki Narisawa, Ayumi Shinohara, Hideo Bannai and Shunsuke Inenaga, Permuted Pattern Matching on Multi-Track Strings, In SOFSEM, pp 280-291, January 2013.
2. Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, Ayumi Shinohara, Position Heaps for Permuted Pattern Matching on Multi-Track Strings, In the local proceeding of SOFSEM 2015, pp XXX-XXX, January 2015.(Poster)

Papers in Domestic Conferences

1. 桂 敬史, 成澤 和志, 篠原 歩. マルチトラック文字列とその索引構造に関する研究, 夏のLA シンポジウム, July 2011.
2. 斎藤 淳哉, 桂 敬史, 一井 宏次, 伊東 裕二, 可児 輝之, 棚橋 広亮, 成澤 和志, 篠原 歩.

実ロボットの自律学習を支援する統合分析環境 : SATORI, 人工知能学会 第 82 回 人工知能基本問題研究会, August 2011.

3. 桂 敬史, 成澤 和志, 篠原 歩, 坂内 英夫, 稲永 俊介. マルチトラック文字列の順列パターン照合と索引構造, コンピューテーション研究会, vol.112, no. 119, pp. 1-8, September 2012.
4. 大田 裕之, 桂 敬史, 成澤 和志, 篠原 歩. マルチトラックデータ上の近似順列パターン照合と索引構造, コンピューテーション研究会, vol. 113, no. 14, pp. 9-16, April 2013.
5. 桂 敬史, 大友 雄平, 成澤 和志, 篠原 歩, マルチトラック文字列上の順列パターン照合のための省メモリな索引構造, コンピューテーション研究会, vol. 114, no. 199, pp. 1-8, September 2014.