Fast Algorithms on Pattern Matching Problems

TOHOKU
UNIVERSITY

Diptarama Hendrian

Graduate School of Information Sciences
Tohoku University, Japan

This dissertation is submitted in partial fulfillment of the requirements for the degree of *Doctor of Philosophy*

23 January 2018

Acknowledgment

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Ayumi Shinohara for his lot of advice and guidance during the 6 years of my study and research at Tohoku University. His continuous guidance helped me all the time of research and writing of this dissertation.

I would also like to express my appreciation to my dissertation committee: Prof. Takeshi Tokuyama and Prof. Xiao Zhou, for their insightful suggestions, feedbacks, and comments which help me to get a better dissertation.

Beside my supervisor, I would also to thank Prof. Ryo Yoshinaka for his advice and guidance during the last 2 years of my Ph.D. study, Dr. Kazuyuki Narisawa for his advice and guidance during the first 1 year of my Ph.D. study, and Prof. Shunsuke Inenaga for his valuable suggestions and discussion on my research.

My deepest appreciation also goes to staffs and colleagues of Shinohara-Yoshinaka Laboratory. Naomi Ogasawara who provided a lot of helpful supports on organizing documents and research fund. All the colleagues in the lab made my college life more enjoyable. In particular, Yohei Ueki, Ichinari Sato, Shintaro Narisada, Kaizaburo Chubachi, Yuki Igarashi, and Davaajav Jargalsaikhan for great research discussion so we can write papers together.

I would also like to express my gratitude to Tohoku University Division for Interdisciplinary Advanced Research and Education, for their support in scholarship and research grant during my Ph.D. study.

Finally, I would like to express special thanks to my wife Radiztia for her love, support, and encouragement that make my life more enjoyable and colorful. I also would like to thank my family and friends for their support.

Diptarama Hendrian

23 January 2018

Abstract

In this dissertation, we propose fast algorithms for various pattern matching problems. Among these pattern matching problems, the first problem is called the dynamic dictionary matching problem, where we find occurrences of patterns in a set that can change dynamically by inserting a pattern into or deleting a pattern from the dictionary. We propose two algorithms for this problem, one based on the directed acyclic word graph (DAWG) and the other based on the Aho-Corasick automaton (AC-automaton) algorithm. The DAWG-based algorithm can solve the problem in $O(m \log \sigma)$ update time and $O(n \log \sigma + occ)$ matching time if only insertion is allowed, and $O(\sigma m + \frac{\log d}{\log \log d})$ update time and $O(n(\frac{\log d}{\log \log d} + \log \sigma) + occ \frac{\log d}{\log \log d})$ matching time if both insertion and deletion are allowed; here, n denotes the length of the text, m denotes the length of pattern that is inserted or deleted, d is the total length of the patterns in the dictionary, σ denotes the alphabet size, and occ denotes the number of occurrence positions of all patterns. In contrast, the AC-automaton-based algorithm has an update time in $O(m \log \sigma + u_f + u_o)$ if only insertion is allowed, and $O(\sigma m + u_f + u_o)$ if both insertion and deletion are allowed, where u_f is the number of states whose failure link needs to be updated, and u_o is the number of states on which the value of the output function needs to be updated. The AC-automaton-based algorithm takes $O(n \log \sigma + occ)$ time for matching the case of both these settings.

The second problem is called the parameterized pattern matching problem, where we find substrings of a text that match the pattern by replacing some symbols by other appropriate symbols. We propose a data structure called the parameterized position heap to solve this problem. We propose an algorithm to construct parameterized position heaps, which runs in $O(n \log(|\Sigma| + |\Pi|))$ time, where $|\Sigma|$ is the number of constant symbols, $|\Pi|$ is the number of parameter symbols. Using the parameterized position heap, we can perform parameterized pattern matching in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.

The third problem is called order-preserving pattern matching, in which we consider the relative order of the elements of the strings instead of their value. We propose a duel-and-sweep algorithm for order-preserving pattern matching. Our algorithm runs in $O(n + m \log m)$ time which is theoretically the same as the Knuth-Morris-Pratt (KMP) algorithm for order-preserving pattern matching, where n denotes the length of the text and m denotes the length of the pattern. Moreover, we show that our algorithm is faster than the KMP-based algorithm through experiments.

The last problem is called permuted pattern matching, where we use multi-track strings which is a group of strings, and find the occurrence positions of all permutations of the pattern in the text. We propose some algorithms for permuted pattern matching on multi-track strings. The permuted matching automaton algorithm has a fast theoretical computing time with $O(mM \log \sigma)$ as the preprocessing time and $O(nN \log \sigma + occ)$ as the matching time, where n denotes the length of the text, m denoted the length of the pattern, N denotes the number of sequences in the text, M denotes the number of sequences in the text, σ denotes the alphabet size, and occ denotes the number of occurrence positions of the pattern. The multi-track Boyer-Moore algorithm and the Horspool algorithm with track-trie are the fastest algorithms experimentally. Furthermore, we also propose the multi-track AC-automaton algorithm that can solve dictionary matching on multi-tracks, finding multiple multi-track patterns in a multi-track text. Finally, we propose filtration algorithms that can, in practice, perform permuted pattern matching fast.

Contents

Chapter 1

Introduction

1.1 Background

In recent years, owing to the decreasing costs of computers and faster internet speeds, the use of computers to store and process data is increasing. Among of these data, many of the variants can be represented as strings. For example, in biology, DNA sequences can be represented as strings consisting of ‘A’, ‘C’, ‘T, and ‘G’; in addition, proteins can be represented as strings of amino acid sequences. Furthermore, in finance, stock prices and currency rates can be represented as integer strings, as can sensor data, such as vehicle velocity, air temperature, and seismic intensity. Moreover, digital data and computer programs can be represented as binary strings. Thus, considering these examples, string processing is considerably important for data analysis in various applications, including pattern matching, data compression, and text similarity, among others. In particular, considering the proliferation of data in modern times, commonly referred to as Big Data, we can gain considerable useful information by analyzing these large datasets. For example, we can find similarities between organisms by identifying similar structures on their DNA; in addition, we can predict future events from sensor data such those related to natural phenomenon like earthquakes or man-made incidents like traffic accidents, which can help us prevent them by applying appropriate countermeasures sooner. In order to process such large data time- and cost-effectively, we require fast algorithms and space-efficient data structures.

One of the fundamental tasks in string processing is the *pattern matching* problem. Given two strings, a text T and a pattern P, the pattern matching problem involves finding all occurrences of P in T. The pattern matching problem has wide applications, such as in keyword search, event recognition, and search engines. A simple approach to this problem involves checking the entire text for the pattern to find its occurrences or its absence at all positions; however, this approach is not efficient and takes $O(|T||P|)$ time in the worst case, where $|T|$ and $|P|$ denote the length of T and P, respectively. Therefore, to overcome the limitations of this simple approach, many pattern matching algorithms have been developed since the 1970s. Pattern matching can be performed efficiently by preprocessing the pattern or by constructing indexing structures based on the text. The first approach is effective in finding a fixed pattern in multiple texts or in the cases where the text is available only in real-time and the pattern needs to be found immediately.

Among the previously proposed pattern matching algorithms in this approach, the *Knuth-Morris-Pratt (KMP) algorithm* [51] is a fundamental algorithm; it utilizes the periodicity of prefixes of a pattern to achieve pattern matching in $O(|T|)$ time with $O(|P|)$ preprocessing time and space complexities. Another basic algorithm for pattern matching is the *Boyer-Moore algorithm* [14], which uses the symbol occurrence positions and periodicity of suffixes of a pattern instead its prefixes as in the KMP algorithm. Later, Horspool [40] proposed an algorithm that used only symbol occurrence positions; this algorithm was shown to be faster than the Boyer-Moore algorithm on average, but was slower for the worst case. Further, Vishkin [70] proposed an algorithm called the *duel-and-sweep algorithm* that utilizes non-periodic properties of a pattern; this method was designed using parallel computing. The duel-and-sweep algorithm first considers all positions in T to be candidates of occurrence positions of P, then eliminates the candidates until only real occurrence positions remain. This algorithm is divided into two stages, namely the *duel* and *sweep* stages. In the duel stage, the algorithm duels, i.e., compares, the candidates with each other using a *witness table*, eliminating the candidates that are proved to be incorrect. This witness table contains a witness for two overlapped candidates, to prove that those candidates cannot occur simultaneously, thus at least one of the candidates is incorrect. After the duel stage, all the remaining candidates are checked in the sweep stage. This algorithm takes $O(|P|)$ time to construct the witness table for P and $O(|T|)$ time for the duel and sweep stages. Aside from these algorithms, the other noteworthy fast or space-efficient algorithms have also been proposed, including the constant space algorithm [36], Karp-Rabin algorithm [47] that uses a hash function, quick searching algorithm [65], and sampling algorithm [71].

In contrast, the other approach for pattern matching algorithms that involve constructing indexing structures from the text, are effective for finding many patterns in a fixed text, for example, finding keywords in a book or webpage. The most basic indexing structure for pattern matching is the *suffix trie*. The suffix trie of a text T is a trie of all suffixes of T. We can find a pattern P on the trie of T from the root iff P is a substring of T. Thus, pattern matching using a suffix trie takes $O(|P|)$ time. However, the suffix trie of T has a space complexity of $O(|T|^2)$. Therefore, to improve the space complexity of the suffix trie, Weiner proposed the *suffix tree* [72], the compacted suffix trie that only require $O(|T|)$ space, by removing nodes that have only one outgoing edge, and concatenating the labels into a string. In addition, Weiner proposed a linear time suffix tree construction algorithm, which is later further improved by McCreight [58]. Ukkonen [69] introduced an online construction algorithm for suffix trees that runs in $O(|T|)$ time; in this algorithm the suffix tree of T is constructed by updating the suffix tree each time a new symbol is read. Another data structure that can be constructed from suffix tries is the *directed acyclic word graph (DAWG)*, which was introduced by Blumer et al. [12]. The DAWG of T is the smallest deterministic finite automaton (DFA) that accepts all suffixes of T, and can be constructed by minimizing the suffix trie of T. In addition, Blumer et al. [12]

proposed a direct online construction algorithm for DAWGs that runs in $O(|T|)$ time. Later, Blumer et al. [13] improved the algorithm so that the DAWG for a set of strings can be constructed in linear time. Ehrenfeucht et al. [32] proposed another linear indexing structure called the *position heap*. The position heap of T is a sequence hash tree [20] of all suffixes of T; in addition, a position heap of T can be considered a subtree of the suffix trie of T that shares the same root. Ehrenfeucht et al. proposed a linear time position heap construction algorithm by inserting all suffixes of T from the shortest to the longest in the length. Furthermore, Kucherov [54] proposed an online position heap construction algorithm that involves inserting all suffixes of T from the longest to the shortest in the length that runs in $O(|T|)$. Aside from these indexing structures, many other indexing structures for pattern matching have been proposed such as suffix arrays [57], suffix cactus [46], suffix trays and trists [23], linear-size suffix tries [27], and linear-sized compact DAWGs [66].

It is common to find more than one pattern in a text. Therefore, given a set of patterns D called a *dictionary* and a text T, we want to find all occurrences of all patterns in D in T. This problem is called the *dictionary matching* problem. In this case, we can find the occurrence positions of all the patterns in the text individually using a pattern matching algorithm. However, this approach is inefficient for finding a fixed set of patterns in the case when the text is flexible or given in an online manner. In order to address this problem efficiently, Aho and Corasick [1] proposed automata with a failure function that can solve this problem in $O(|T|)$ time with an $O(\|D\|)$ preprocessing time, where $\|D\|$ denotes the total length of the patterns in D. It is noteworthy that this algorithm is an extension of the KMP-algorithm for multiple patterns. Commentz-Walter [24] proposed another automaton-based algorithm that is based on the Boyer-Moore algorithm. While theoretically slower than the Aho-Corasick algorithm, in practice, this algorithm is faster than the Aho-Corasick algorithm on average. Meyer [59] introduced a dictionary matching problem where insertion of a pattern into the dictionary is allowed, which is known as the *semi-dynamic dictionary matching problem.* He proposed an algorithm for this semi-dynamic dictionary matching, which updates the Aho-Corrasick automata (AC-automata) when a new pattern is inserted into the dictionary. Later, Amir et al. [5] introduced the *dynamic dictionary matching problem* which allows for both insertion and deletion of patterns in a dictionary. Further, several linear $O(\|D\|)$ space data structures are used in solving the dynamic dictionary matching problem have been proposed [2, 3, 6, 5, 16, 42].

Motivated by finding duplications in program codes, Baker introduced an extension of pattern matching, called *parameterized pattern matching* [9, 11]. The parameterized pattern matching focuses on matching a string with another string by replacing the symbols in the first string with symbols of the other string using some functions. She introduced *parameterized strings*, which are strings consisting of *constant symbols*, i.e., symbols that cannot be replaced, and *variable symbols*, i.e., symbols that can be replaced by other variable symbols. Given two *parameterized strings*, we can say that the strings are *parameter matched*, if there exists a bijection from the symbols of one string to the symbols of the other string such that the bijection is an identity on the constant symbols and the strings match after the symbols are replaced. Given a text T and a pattern P, the *parameterized pattern matching* problem involves finding positions of the substrings of T that parameterized match with P. To solve this problem, Baker proposed an encoding called *prev-encoding* for parameterized strings and constructed *parameterized suffix trees* from the encoded texts; her algorithm constructs parameterized suffix trees in $O(n|\Pi| + \log(|\Sigma| + |\Pi|))$ time and can perform parameterized pattern matching in $O(m\log(|\Sigma| + |\Pi|) + occ)$ time using the constructed parameterized suffix trees. Later, Kosaraju [62] proposed a faster parameterized suffix tree construction algorithm, that can construct the trees in $O(n\log(|\Sigma| + |\Pi|))$ time. Another data structure for parameterized strings is the *parameterized suffix array*, proposed by Deguchi et al. [31] for binary strings and later generalized by I et al. [41]. Other algorithms for parameterized strings have been proposed that can be found in [7, 10, 21, 22].

Recently, Kubica et al. [53] and Kim et al. [50] independently introduced another variant of pattern matching, called *order-preserving pattern matching*. The order preserving pattern matching considers *order-isomorphism* of *integer strings* instead of their value. Given two integer strings of equal length, they are *order-isomorphic* iff the value relation of any two positions in the strings is the same. Then, the order-preserving pattern matching algorithm involves finding positions of substrings of a text T that is order-isomorphic with a pattern P. On considering this problem, Kubica et al. motivated by the combinatorial properties of the strings, while Kim et al. motivated by some applications of this problem such as stock market analysis and melody similarities. Both of aforementioned studies proposed algorithms based on the KMP algorithm and can solve the order-preserving pattern matching problem in $O(|T| + |P| \log |P|)$ time. In addition, Cho et al. [19] proposed another algorithm for this problem based on the the Horspool algorithm that uses q-grams and showed that their algorithm is experimentally faster than the KMP-based algorithm, although their algorithm is theoretically slower in the worst case. As another approach, Crochemore et al. [28] proposed indexing structures for order-preserving pattern matching called incomplete order-preserving suffix trees and complete order-preserving suffix trees. The incomplete order-preserving suffix tree for T can be constructed in $O(|T| \log\log |T|)$ time, whereas the complete order-preserving suffix tree of T can be constructed in $O(\frac{|T| \log |T|}{\log\log |T|})$ time. Order-preserving pattern matching can be performed using these data structures in $O(|P| + occ)$ time. Furthermore, Chhabra and Tarhio [18] also Faro and Külekci [33] proposed filtration methods for order-preserving pattern matching which are practically fast. Moreover, the filtration algorithms can perform order-preserving pattern matching faster using Single Instruction Multiple Data (SIMD) instructions [15, 17, 68].

In 2013, Katsura et al. [48] proposed a pattern matching problem on strings, which consisted of multiple sequences of the same length, called *multi-track strings* (*multi-track*, in short). Two multi-tracks of the same length that have the same number of sequences are *permuted match* if one of the multi-tracks is a permutation of another multi-track. Given a multi-track text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$ that have the same number of sequences, the permuted pattern matching problem involves finding positions of the substrings of $\mathbb{T}$ that permuted match with $\mathbb{P}$. This problem is motivated by the need to find patterns of multi-sequence data in various types of texts, such as multiple-sensor data, polyphonic music data, and multiple genomes. In addition, Katsura et al. [48] proposed some algorithms to solve the permuted pattern matching problem. The first algorithm uses the AC-automata of all sequences in $\mathbb{P}$, which can solve the permuted pattern matching problem in linear time with respect to the size of $\mathbb{T}$ and $\mathbb{P}$. The second algorithm uses an indexing structure called *multi-track suffix trees*; using their algorithm, the multi-track suffix tree of $\mathbb{T}$ can be constructed in linear time with respect to the size of $\mathbb{T}$. Furthermore, later, Katsura et al. [49] proposed two data structures for permuted pattern matching: the time-efficient *multi-track position heap* and the memory-efficient *contracted multi-track position heap*. A variant of permuted pattern matching problem, called *sub-permuted pattern matching* problem, is considered when the number of sequences in the pattern is the same or less than the number of sequences in the text; this problem is considered more difficult than the permuted pattern matching problem. However, this problem can be solved by using the AC-automaton based algorithm [48].

In this dissertation, we propose algorithms for the dynamic dictionary matching, parameterized pattern matching, order-preserving pattern matching, and permuted pattern matching problems. Furthermore, we show the worst case computation time of our algorithms theoretically; in addition, we experimentally obtain the running time of some of the algorithms by implementing them in practice.

1.2 Contributions

In this dissertation, we propose fast algorithms for various pattern matching problems, namely the dynamic dictionary matching, parameterized pattern matching, order-preserving

Table 1.1: Summary of proposed algorithms on each problems. Each algorithm is published on the paper in List of Publication indicated by superscripted number.

Problem	Algorithm
Semi-dynamic dictionary matching	DAWG-based[9]
	AC-automaton-based[9]
Dynamic dictionary matching	DAWG-based
	AC-automaton-based
Parameterized pattern matching	Parameterized position heaps[6]
Oerder-preserving pattern matching	Duel-and-sweep[4]
Permuted pattern matching	MTKMP[14]
	MT AC-automaton[11]
	MT permuted matching automaton[11]
	MT Boyer-Moore[11]
	MT Horspool[11]
	Filtration[14]
	MT duel-and-sweep

pattern matching, and permuted pattern matching problems. The details of our algorithms are as follows:

1. For the semi-dynamic and dynamic dictionary matching problems, we propose efficient algorithms by using DAWGs and AC-automata. The first algorithm is a DAWG-based algorithm that has an update time in $O(m \log \sigma)$ and matching time in $O(n \log \sigma + occ)$ for the semi-dynamic dictionary matching problem; in contrast, it has an update time in $O(\sigma m + \frac{\log d}{\log \log d})$ and a matching time in $O(n(\frac{\log d}{\log \log d} + \log \sigma) + occ\frac{\log d}{\log \log d})$ for the dynamic dictionary matching problem. Here n denotes the length of the text, m denotes the length of the pattern, d denotes the total length of the patterns in the dictionary, and σ denotes the alphabet size. We also propose an algorithm that can update the AC-automata by using DAWGs in $O(m \log \sigma + u_f + u_o)$ time for the semi-dynamic setting and $O(\sigma m + u_f + u_o)$ time for the dynamic settings, where u_f is the number of states whose failure link needs to be updated, and u_o is the number of states for which the value of the output function needs to be updated. We show that our AC-automaton update algorithm is faster than any

other AC-automata update algorithms. The details are described in Chapter 4.

2. For parameterized pattern matching, we propose a data structure called *parameterized position heap*. The parameterized position heap for T is constructed from all prev-encoded suffixes of T from the longest to the shortest. Further, we propose an online construction algorithm based on Kucherov's algorithm [54]. The algorithm can construct the parameterized position heap of T in $O(n \log(|\Sigma| + |\Pi|))$ time, where $|\Sigma|$ is the number of constant symbols and $|\Pi|$ is the number of parameter symbols. We can find the occurrence positions of P in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time by using the parameterized position heaps. The details are described in Chapter 5.

3. For order-preserving pattern matching, we propose a duel-and-sweep algorithm that can perform order-preserving pattern matching in $O(n)$ time with $O(m \log m)$ preprocessing time. We use a pair of positions instead of one position as a witness for the duel stage. We show that our algorithm is theoretically as fast as the KMP algorithm for order-preserving pattern matching and faster than it in practice. The details are described in Chapter 6.

4. For permuted pattern matching, we propose some algorithms and data structures that preprocess the pattern. We extend the KMP algorithm, Boyer-Moore algorithm, Horspool algorithm, and duel-and-sweep algorithm for use with multi-tracks. In addition. we propose an AC-automaton-based algorithm that can perform dictionary matching on multi-tracks in linear time with respect to the size of the input. Through experiments, we show that our algorithms are faster than existing algorithms. The details are described in Chapter 7.

The rest of this dissertation is organized as follows. In Chapter 2, we describe the notations and basic algorithms that we use. Next, in Chapter 2, we describe the data structures that we extend to solve the problems that are specified above. In Chapters 4, 5, 6, and 7, we propose algorithms for dynamic-dictionary matching, parameterized pattern matching, order-preserving pattern matching, and permuted pattern matching, respectively. Furthermore, we describe the details of the problems and some previous results in these chapters. Lastly, we conclude our work in Chapter 8 and discuss future work that we might undertake.

Chapter 2

Preliminaries

2.1 Notation

Let Σ be an *alphabet*, a set of symbols, and $\sigma = |\Sigma|$ be the alphabet size. An element of Σ^* is called a *string*. For a string $W \in \Sigma^*$, the length of W is denoted by $|W|$. The *empty string*, denoted by ε, is a string of length 0. For a string $W \in \Sigma^*$ of length n, $W[i]$ denotes the i-th symbol of W, $W[i:j] = W[i]W[i+1]\dots W[j]$ denotes a substring of W that begins at position i and ends at position j for $1 \leq i \leq j \leq n$. For convenience, we abbreviate $W[1:i]$ to $W[:i]$ and $W[i:n]$ to $W[i:]$, which are called *prefix* and *suffix* of W, respectively. Moreover, let $W[i:j] = \varepsilon$ if $i > j$. The reverse string of W is denoted by $W^R = W[n]W[n-1]\dots W[2]W[1]$. For two strings X and Y, $X \prec Y$ denotes that X is lexicographically smaller than Y and $X \preceq Y$ denotes that either $X = Y$ or $X \prec Y$. For a string W, let $\mathit{Substr}(W)$ denote the set of all substrings of W. Similarly, let $\mathit{Pref}(W)$ be the set of all prefixes of W and $\mathit{Suff}(W)$ be the set of all suffixes of W.

Let $D = \{W_1, W_2, \dots, W_r\}$ be a set of strings over Σ. D is often called a *dictionary*. Let $|D| = r$ be the size of dictionary and $\|D\| = \sum_{i=1}^{r} |W_i|$ be the total length of the patterns in D. For a dictionary D, let $\mathit{Substr}(D) = \bigcup_{i=1}^{r} \mathit{Substr}(W_i)$, $\mathit{Pref}(D) = \bigcup_{i=1}^{r} \mathit{Pref}(W_i)$, and $\mathit{Suff}(D) = \bigcup_{i=1}^{r} \mathit{Suff}(W_i)$.

The pattern matching problem is a task to find occurence positions of a pattern P in a text T. Formally, the pattern matching problem is defined as follows.

Definition 2.1. *Given a text T of length n and a pattern P of length m, output all position i such that $T[i : i + m - 1] = P$.*

We will describe a naive algorithm and some efficient algorithms for pattern matching problem in this chapter.

2.2 Naive algorithm for pattern matching

In this section, we will describe a naive algorithm for pattern matching problem. Let T denote a text of length n and P denote a pattern of length m.

Let $i = 1$ and $j = 1$. The naive algorithm compares P with T from the left to the right. First, the algorithm compares $P[j]$ with $T[i+j-1]$ which is initially $P[1]$ with $T[1]$. If $P[j] = T[i+j-1]$, the algorithm will compare the next symbol by increasing the value of j by one and repeat the same procedure. On the other hand, if $P[j] \neq T[i+j-1]$, we found a mismatch at position j of pattern and $T[i : i+m-1]$ will not match P. In this case we *shift* P by one to the right by increasing the value of i by one and start comparing $P[1]$ with $T[i]$ by setting $j = 1$. When the value of j is increased until $m + 1$, we know that $P[j] = T[i + j - 1]$ holds for $1 \leq j \leq m$. We should output i as an occurrence position since $P = T[i : i + m - 1]$, then shift P by one to the right and set $j = 1$.

By using above procedure, the naive algorithm can perform pattern matching in $O(nm)$ time. The pattern matching problem can be solved faster by increasing the shift amount when a mismatch occurs, and by setting comparison position j to other than 1.

2.3 Knuth-Morris-Pratt algorithm

In this section, we will describe two linear time algorithms for the exact pattern matching problem, called the Morris-Pratt algorithm (MP algorithm) and the Knuth-Morris-Pratt algorithm (KMP algorithm) [51]. Let T denote a text of length n and P denote a pattern of length m.

Similarly to the naive algorithm, the MP algorithm compares P to T from the left to the right. The MP algorithm is an algorithm that uses *border array* $Border_P$ of P to compute the shift amount *MPShift*, which is also called *failure function*, when there is a mismatch on position $i + 1$ of P. A *border* of P is any string that is simultaneously a prefix and a suffix of P. A *proper border* of P is any border of P except P itself. A border array $Border_P$ of P is an array where $Border_P[i]$ is the length of the longest proper border of $P[: i]$ for $1 \leq i \leq m$. In order to simplify the algorithm let $Border[0] = -1$. The amount of shift that MP algorithm do when a mismatch occurs at position $i + 1$ is $MPShift[i] = i - Border[i]$ for $0 \leq i \leq m$. In other words, when $P[i]$ mismatches $T[j]$, MP algorithm continue the matching from $P[i']$ with $T[j]$ where $i' = i - Border[i]$. Note that $MPShift[m]$ is used if P matches a substring of T.

The KMP algorithm is an improvement of the MP algorithm. This algorithm uses a stronger condition $StrongBorder_P$ to compute the shift amount *KMPShift*. The condition is defined as $StrongBorder[i]_P = k$, where k is the length of the longest border that satisfies $P[k + 1] \neq P[i + 1]$. In order to simplify the algorithm let $StrongBorder_P[0] = -1$. By using $StrongBorder_P[i]$, the shift amount is defined as $KMPShift[i] = i - StrongBorder_P[i]$ for $0 \leq i \leq m$. Therefore, the KMP algorithm shifts the pattern the same or more than the MP algorithm when a mismatch occurs.

We will extend the KMP algorithm for permuted pattern matching in Chapter 7.

2.4 Boyer-Moore algorithm and Horspool algorithm

In this section, we will describe two exact pattern matching algorithms that utilize occurrence positions of symbols in the pattern. The first algorithm is called Boyer-Moore algorithm [14] and the second algorithm is called Horspool algorithm [40].

Boyer-Moore algorithm uses two functions $BadSym_P$ and $GoodSuf_P$ while Horspool algorithm uses only BadSym_P to determine the shift amount of the pattern where there is a mismatch. For a symbol c, we define $\mathit{BadSym}_P[c]$ as the rightmost occurrence position of c on P excluding $P[m]$ and $\mathit{BadSym}_P[c] = 0$ where there is no occurrence of c in P or when c only occurs in position m on P. For a position i on P, $\mathit{GoodSuf}_P[i]$ is defined as follows. If $P[i+1:]$ occurs in $P[:i]$, $\mathit{GoodSuf}_P[i] = j$ where j is the the rightmost occurrence position of $P[i+1:]$ in $P[:i]$. If $P[i+1:]$ do not occur in $P[:i]$, $\mathit{GoodSuf}_P[i] = j$ where j is the longest suffix of $P[i+1:]$ that also a prefix of P.

Boyer-Moore algorithm matches the pattern to the text from the rightmost symbol of the pattern to the left. When a mismatch occurs at position i on the pattern and j on the text, Boyer-Moore algorithms shift the pattern by $\mathit{BMShift}[i] = \max(\mathit{GoodSuf}_P[i], \mathit{BadSym}_P[T[j]])$ to the right. The algorithm uses $\mathit{BMShift}[0] = \mathit{GoodSuf}_P[0]$ if the pattern matches a substring of the text. On the other hand, Horspool algorithm can perform pattern matching from the left side or the right side of the pattern. When a mismatch occurs at position i on the pattern and j on the text, Boyer-Moore algorithms shift the pattern by $\mathit{HorsShift}[i] = \mathit{BadSym}_P[T[j+m-i]]$ to the right.

We will extend Boyer-Moore algorithm and Horspool algorithm for permuted pattern matching in Chapter 7.

2.5 Duel-and-sweep algorithm

In this section, we will describe the duel-and-sweep algorithm for pattern matching problem [4, 70].

First, the duel-and-sweep algorithm considers all positions i in T to be *candidates* for occurrence positions and eliminates the candidates until only the correct occurrence positions of P remain. The algorithm eliminates the candidates in two stages, *dueling* and *sweeping* stages. In the duel stage, the algorithm “duels” two superimposed and contradicted candidates by using a *witness table* and deletes one of the candidates. In the sweeping stage, the algorithm checks all candidates whether they are matched or not.

First, we will describe the witness tables that will be used in the duel stage. The algorithm first preprocesses P to construct the witness table WIT_P of P. The witness table WIT_P is an array of size $m-1$ that saves a witness position i such that $P[i] \neq P[i+a]$ in $WIT_P[a]$ for each offset a. When the overlap regions are matched for offset a, we save 0 in $WIT_P[a]$.

After constructing the witness table WIT_P, the algorithm performs dueling on the text. For superimposed pair of candidate positions x, $x+a$, the algorithm duels the candidates by using $WIT_P[a]$ and eliminates one of the candidates if $WIT_P[a] \neq 0$, otherwise we call x is *consistent* with $x+a$ and keep both candidates. Formally, let $i = WIT_P[a]$, the algorithm eliminates $x+a$ if $T[x+a+i-1] \neq P[i]$, otherwise x is eliminated in dueling stage. After the duel stage, all remaining candidates are consistent with each other, The algorithm then checks whether the candidates are matched or not in the sweep stage.

We extend this algorithm for order-preserved pattern matching and describe it in Chapter 6.

Chapter 3

Data Structures

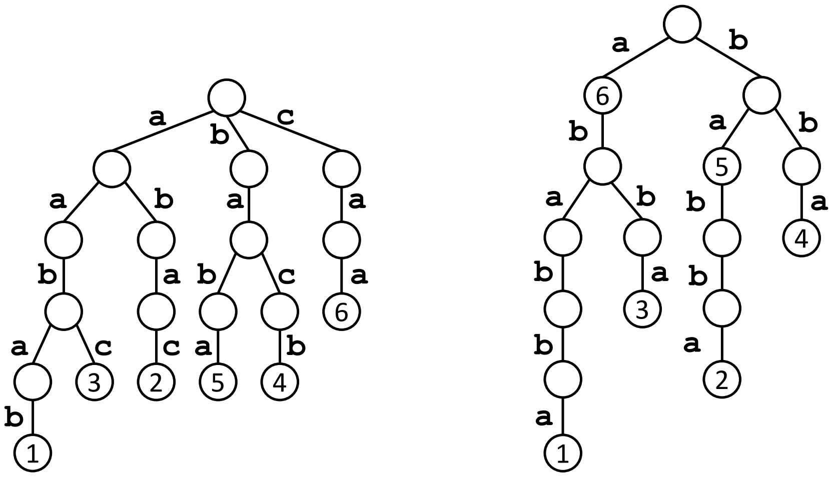
3.1 Suffix trie

Tries are tree data structures that contain information of strings and their prefixes. Let D be a set of strings. The suffix trie of D, denoted by $\mathit{Trie}(D)$, is a tree where each edge of the trie labeled by a symbol, each leaf represents a string that is stored in the trie, and each node represents a prefix of a string that is stored in the trie. Fig. 3.1 (a) shows an example of a trie. We can find any prefix $W \in \mathit{Pref}(D)$ by traversing $\mathit{Trie}(D)$ by following edges $W[1]$ to $W[|W|]$ from the root. We identify each node in the trie by the string that is obtained by concatenating the path labels from the root to the node.

The suffix trie of a string T is a trie that contains all suffixes of T.

Definition 3.1. *For a string W, the suffix trie of W is $\mathit{STrie}(W) = \mathit{Trie}(\mathit{Suff}(W))$.*

Fig. 3.1 (b) shows an example of a suffix trie. Since we can find any prefix of any string that is contained in a trie and a suffix trie contains all suffixes of a string, we can find any substrings of the string by using the suffix trie. Suffix trie is useful for solving pattern matching problem. Given a pattern P of length m and a text T of length n over an alphabet Σ, we can find occurrences of P in $O(m \log |\Sigma|)$ by using the $\mathit{STrie}(T)$. However, we need $O(n^2 \log |\Sigma|)$ time and space to construct $\mathit{STrie}(T)$. Many of suffix



(a) (b)

Figure 3.1: (a) The trie of $D = \{\mathtt{aabab}, \mathtt{abac}, \mathtt{aabc}, \mathtt{bacb}, \mathtt{baba}, \mathtt{caa}\}$ and (b) the suffix trie of $W = \mathtt{ababba}$.

trie based data structures that can be constructed in linear time with respect to the text length are proposed, such as suffix trees [72, 58, 69], directed acyclic word graphs [12, 13], and position heaps [32, 54]. We will describe some of these data structures later in this chapter.

3.2 Aho-Corasick automaton

Let $D = \{p_1, p_2, \ldots, p_r\}$ be a set of *patterns* over Σ called a *dictionary.* Let $d = \|D\| = \sum_{i=1}^{r} |p_i|$, the total length of the patterns in D. The *Aho-Corasick Automaton* of D, denoted by $\mathsf{AC}(D)$, is a trie of all patterns in D, associated to *goto*, *failure* and *output* functions [1]. We call a node in Aho-Corasick automaton as a *state*. We often identify a state s of $\mathsf{AC}(D)$ with the string obtained by concatenating all the labels found on the path from the root to the state s. The state transition function *goto* δ is defined so that for any two states $s, s' \in \mathit{Substr}(D)$ and any symbol $c \in \Sigma$, if $s' = sc$ then $s' = \delta(s, c)$. We can naturally extend the domain of the second argument of δ to Σ^*, that $\delta(s, cw) = \delta(\delta(s, c), w)$ and $\delta(s, \varepsilon) = s$ for any string s and symbol c. The failure function is defined by $\mathsf{flink}(s) = s'$ where s' is the longest proper suffix of s such that

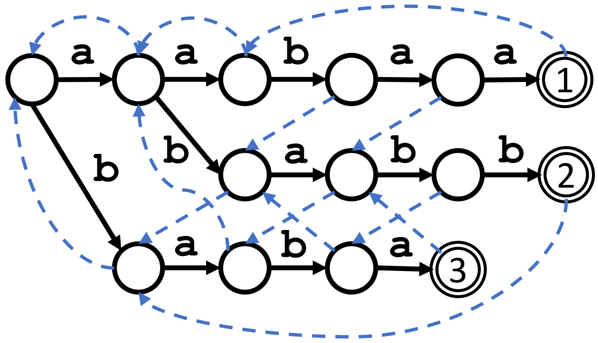


Figure 3.2: The Aho-Corasick automaton of $D = \{\mathtt{aabaa}, \mathtt{ababb}, \mathtt{baba}\}$. Black solid lines represent the goto function and blue dashed lines represent the failure function.

$s' \in \mathit{Pref}(D)$. We can also extend the failure function that $\mathsf{flink}^i(s) = \mathsf{flink}^{i-1}(\mathsf{flink}(s))$ and $\mathsf{flink}^1(s) = \mathsf{flink}(s)$. Finally, $\mathsf{output}(s)$ returns the set of all patterns that are suffixes of s. Fig. 3.2 shows an example of Aho-Corasick automaton.

For a dictionary D and a text T, $\mathsf{AC}(D)$ can be constructed in $O(d \log \sigma)$ time. We find occurrence positions of any pattern on given text T of length n in $O(n \log \sigma)$ time by using the AC-automaton. This problem is called *dictionary matching*. This problem will be discussed in Chapter 4.

3.3 Directed acyclic word graph

The *Directed acyclic word graph*, shortly *DAWG*, graph data structure obtained by minimizing suffix tries [12, 13]. Let $D = \{W_1, W_2, \ldots, W_r\}$ be a set of strings over Σ. Let $d = \|D\| = \sum_{i=1}^{r} |W_i|$ be the total length of the patterns in D. For any string x, let

$$\mathsf{endPos}_D(x) = \{(i,j) \mid x = p_i[j - |x| + 1 : j], |x| \leq j \leq |p_i|, p_i \in D\}$$

namely, $\mathsf{endPos}_D(x)$ represents the set of ending positions of x in patterns of D. For any $x, y \in \mathit{Substr}(D)$, we define the equivalence relation $\equiv_D$ such that $x \equiv_D y$ iff $\mathsf{endPos}_D(x) = \mathsf{endPos}_D(y)$. We denote by $[x]_D$ the equivalence class of x with respect to $\equiv_D$. The *directed acyclic word graph* (*DAWG*) [13, 12] of D, denoted by $\mathsf{DAWG}(D)$, is an edge-labeled

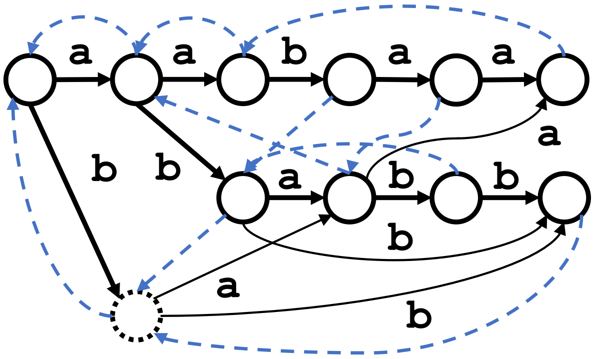


Figure 3.3: Directed acyclic word graph of $D = \{\mathsf{aabaa}, \mathsf{ababb}\}$. Solid-line circles show trunk nodes and the dashed-line circle shows a non-trunk node. Thick solid lines, thin solid lines, and dashed lines show primary edges, secondary edges and suffix links, respectively.

directed acyclic graph (V, E) such that

$$\begin{aligned} V &= \{[x]_D \mid x \in \mathit{Substr}(D)\}, \\ E &= \{([x]_D, c, [xc]_D) \mid x, xc \in \mathit{Substr}(D), c \in \Sigma, x \not\equiv_D xc\}. \end{aligned}$$

Namely, each node of $\mathsf{DAWG}(D)$ represents each equivalence class of substrings of D, and henceforth we will identify a DAWG node with an equivalence class of substrings. The node $[\varepsilon]_D$ is called the *source* of $\mathsf{DAWG}(D)$. For each node $[x]_D$ except the source, the *suffix link* is defined by $\mathsf{slink}([x]_D) = [y]_D$, where y is the longest suffix of x satisfying $x \not\equiv_D y$. For convenience, we define $\mathsf{slink}^1([x]_D) = \mathsf{slink}([x]_D)$ and $\mathsf{slink}^i([x]_D) = \mathsf{slink}(\mathsf{slink}^{i-1}([x]_D))$ for $i > 1$. A node v of $\mathsf{DAWG}(D)$ is called a *trunk node* if there is a path from the source to v which spells out some prefix of a pattern in $\mathit{Pref}(D)$, and it is called a *non-trunk node* otherwise. An edge e from $[x]_D$ to $[xc]_D$ labeled by c is a *primary edge* if both x and xc are the longest strings in their equivalence classes, otherwise it is a *secondary edge*. It is known (c.f. [13]) that the numbers of nodes, edges, and suffix links of $\mathsf{DAWG}(D)$ are all linear in d. Fig. 3.3 shows an example of a DAWG.

DAWGs can be used for pattern matching [26]. Blumer et al. proposed a DAWG construction algorithm for a single string [12], then they improved their algorithm for multiple strings [13]. The algorithm can construct a DAWG online in $O(d \log \sigma)$ time, where the algorithm updates the DAWG each time a new symbol is read. Later, Kucherov et al. [55] proposed a deletion algorithm for DAWG, which can update the DAWG when a string is deleted from the dictionary. We use DAWGs to solve the dynamic dictionary problem that will be described in Chapter 4.

3.4 Position heap

In this section, we briefly review the position heap for strings. First, we introduce the *sequence hash tree* that is a trie for hashing proposed by Coffman and Eve [20]. Each edge of the trie is labeled by a symbol and each node can be identified with the string obtained by concatenating all labels found on the path from the root to the node.

Definition 3.2 (Sequence Hash Tree). *Let $D = (W_1, \ldots, W_n)$ be an ordered set of strings over Σ and $D_i = (W_1, \ldots, W_i)$ for $1 \le i \le n$. A* sequence hash tree $\mathsf{SHT}(D) = (V_n, E_n)$ *for D is a trie over Σ defined recursively as follows. Let $\mathsf{SHT}(D_i) = (V_i, E_i)$. Then,*

$$\mathsf{SHT}(D_i) = \begin{cases} (\{\varepsilon\}, \emptyset) & \text{if } i = 0, \\ (V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}) & \text{if } 1 \le i \le n. \end{cases}$$

where p_i is the shortest prefix of W_i such that $p_i \notin V_{i-1}$, , $q_i = W_i[1 : |p_i| - 1]$, and $c = W_i[|p_i|]$, also we store index i in p_i. If no such p_i exists, then $V_i = V_{i-1}$ and $E_i = E_{i-1}$. We store index i in W_i in this case. Therefore, each node in a sequence hash tree except the root stores one or several indices of strings in the input set.

An example of a sequence hash tree is shown in Fig. 3.4 (a).

Given a text T of length n, the *position heap*, proposed by Ehrenfeucht et al. [32], is a sequence hash tree of all suffixes of T that is sorted in ascending order of length. Later, Kucherov [54] proposed another type of position heap, which is a sequence hash tree of sorted *Suff*(T) in descending order of length. The position heap definiton by Kucherov will be used in Chapter 5.

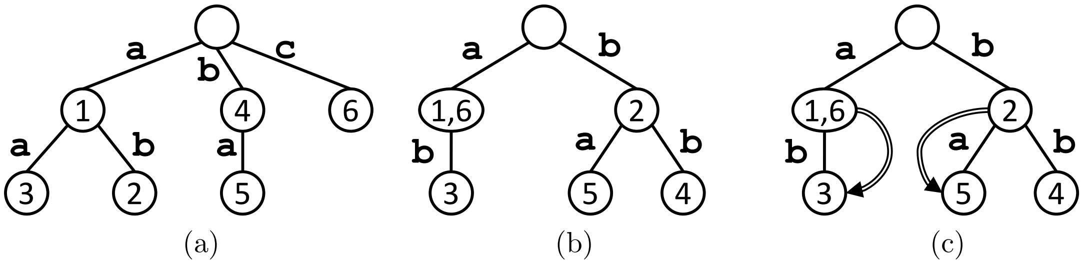


Figure 3.4: (a) Sequence hash tree of $D = (\texttt{aabab}, \texttt{abac}, \texttt{aabc}, \texttt{bacb}, \texttt{baba}, \texttt{caa})$, (b) position heap of $T = \texttt{ababba}$, and (c) augmented position heap of $T = \texttt{ababba}$. Double line arrows show the maximal reach pointer.

Definition 3.3 (Position Heap [54]). *Given a string $T \in \Sigma^n$, let $S_T = (T[1:], T[2:], \ldots, T[n:])$ be the ordered set of all suffixes of T except ε in descending order of length. The* position heap $\mathsf{PH}(T)$ *for T is* $\mathsf{SHT}(S_T)$.

Each node except the *root* in a position heap stores either one or two integers, which is beginning positions of corresponding suffixes. We call them *regular node* and *double node* respectively. Assume that i and j are positions stored by a double node v in $\mathsf{PH}(T)$ where $i < j$, then i and j are called the *primary position* and the *secondary position*, respectively. Fig. 3.4 (b) shows an example of a position heap.

Kucherov [54] also proposed online construction of position heap that run in $O(n \log \sigma)$. In order to find occurrences of the pattern in $O(m \log \sigma + occ)$ time, Ehrenfeucht et al. [32] and Kucherov [54] added additional pointers called *maximal-reach pointers* to the position heap and called this extended data structure *augmented position heap*. An example of an augmented position heap is shown in Fig. 3.4 (c).

Chapter 4

Dynamic Dictionary Matching Algorithm

The *Dictionary matching problem* is an extension of the pattern matching problem. The dictionary matching problem is a task to find occurrence positions of multiple patterns simultaneously instead of a single pattern.

Definition 4.1 (Dictionary matching [1, 5])**.** *Given a set $D = \{P_1, P_2, \ldots, P_r\}$ of patterns called* dictionary *and a text T over an alphabet Σ, find all positions i in T such that $P_j = T[i : i + |P_j| - 1]$ for all $1 \leq j \leq r$.*

The dictionary matching problem can be solved by the Aho-Corasick algorithm [1], which is based on the KMP algorithm, or the Commentz-Walter algorithm [24], which is based on the Boyer-Moore algorithm. For a dictionary of size d over an alphabet of size σ, both of the above algorithms first preprocess the dictionary in $O(d \log \sigma)$ time. Then, given a text of length n, the occurrences of all patterns $P_i \in D$ in the text can be reported in $O(n \log \sigma + occ)$ time by the AC-algorithm, and in $O(nd \log \sigma)$ time by the Commentz-Walter algorithm, where occ is the total number of occurrences of all the patterns in the text. Notice that $occ \leq nd$ always holds and hence the occ term is omitted in the latter time complexity.

Meyer [59] introduced the incremental string matching problem, which is also known as the *semi-dynamic dictionary matching problem*, a variant of dictionary matching that allows insertion of a pattern into the dictionary. He proposed an algorithm for semi-dynamic dictionary matching, which updates the AC-automaton when a new pattern is inserted into the dictionary. Amir et al. [5] introduced the *dynamic dictionary matching problem* which allows for both insertion and deletion of patterns. Several sophisticated data structures for dynamic dictionary matching have been proposed in the literature [2, 3, 6, 5, 16, 42]. All these data structures use linear $O(d)$ (words of) space. More recently, succinct data structures for dynamic dictionary matching have been produced, where the main concern is to store the dictionary in memory space close to the information theoretical minimum [34, 39].

Remark that in all the above-mentioned approaches except Meyer's, the pattern matching time to search a text for dictionary patterns is sacrificed to some extent. Tsuda et al. [67] proposed a dynamic dictionary matching algorithm, which follows and extends Meyer's method. Whilst Tsuda et al.'s method retains $O(n \log \sigma + occ)$ pattern matching time, still it requires $O(z \log \sigma)$ time to update the AC-automaton upon each insertion/deletion, where z is the size of the AC-automaton. Note that in the worst case this can be as bad as constructing the AC-automaton from scratch, since z can be as large as the dictionary size d. Ishizaki and Toyama [45] introduced a data structure called an *expect tree* which efficiently updates the dictionary for insertion of patterns and showed some experimental results, but unfortunately no theoretical analysis was provided. See Table 4.1 for a summary of the update times and pattern matching times for these algorithms.

Along this line, in this chapter, we propose new efficient algorithms for the semi-dynamic and dynamic dictionary matching problems, where pattern matching can still be performed in $O(n \log \sigma + occ)$ time.

Firstly, we show a dynamic dictionary matching algorithm which is based on Blumer et al.'s *directed acyclic word graphs* (*DAWGs*) [12, 13]. The DAWG of a dictionary D is a (partial) DFA of size $O(d)$ which recognizes the suffixes of the patterns in D. We show how to perform dynamic dictionary matching with DAWGs, by modifying Kucherov and Rusinowitch's algorithm which originally uses DAWGs for pattern matching with variable length don't cares [55]. The key idea is to use efficient *nearest marked ancestor* (*NMA*) data structures [2, 3, 73] on the tree induced from the suffix links of the DAWG. For the semi-dynamic dictionary matching, our DAWG method achieves $O(m \log \sigma)$ update time, $O(n \log \sigma + occ)$ pattern matching time, and uses linear $O(d)$ space, where m is the length of pattern p to insert. For the dynamic version of the problem, our DAWG method uses $O(\sigma m + \log d / \log\log d)$ update time, $O(n(\log d / \log\log d + \log \sigma) + occ \log d / \log\log d)$ matching time, and $O(d)$ space. The term σm in the update time is indeed unavoidable for maintainining the DAWG in the dynamic setting, namely, we will also show that there is a sequence of insertion and deletion operations for patterns of length m such that each insertion/deletion operation takes $\Omega(\sigma m)$ time.

Secondly, we present another algorithm for the semi-dynamic and dynamic dictionary matching problem which is based on the AC-automaton. This is closely related to our first approach, namely, the second algorithm updates the AC-automaton with the aid of the DAWG. The algorithm uses $O(d)$ space, finds pattern occurrences in the text in $O(n \log \sigma + occ)$ time, and updates the AC-automaton in $O(m \log \sigma + u_f + u_o)$ time with additional DAWG update time, $O(m \log \sigma)$ time for semi-dynamic and $O(\sigma m)$ time for dynamic settings, where u_f is the number of states whose failure link needs to be updated, and u_o is the number of states on which the value of the output function needs to be updated. Therefore, when u_f and u_o are sufficiently small and σ is constant, our update operation can be faster than other approaches. Notice that u_o is negligible unless the pattern p to insert/delete is a common prefix of many other patterns; in particular $u_o = 0$ for any prefix codes. Also, u_f is negligible unless the prefixes of p are common substrings of many other patterns. We emphasize that the update time of our algorithm

Table 4.1: Comparison of the algorithms for the dynamic dictionary matching. The last four in the table are our proposed methods. Here, n is the length of the text, d is the total length of the dictionary of patterns, and m is the length of the pattern to insert or delete. k and ϵ are any constants with $k \geq 2$ and $0 < \epsilon < 1$, respectively. $l_{\max}$ is the length of the longest pattern in the dictionary, and z is the size of the AC-automaton before updates. † indicates algorithms for semi-dynamic dictionary matching which allows only for insertion. The bounds for Chan et al.'s algorithm [16] hold for constant alphabets.

Algorithm	Update time	Pattern matching time
Idury & Schäffer [42]	$O(m(kd^{1/k} + \log\sigma))$	$O(n(k + \log\sigma) + k \cdot occ)$
Amir et al. [6]	$O(m(\frac{\log d}{\log\log d} + \log\sigma))$	$O(n(\frac{\log d}{\log\log d} + \log\sigma) + occ\frac{\log d}{\log\log d})$
Alstrup et al. [2, 3]	$O(m\log\sigma + \log\log d)$	$O(n(\frac{\log d}{\log\log d} + \log\sigma) + occ)$
Chan et al. [16]	$O(m\log^2 d)$	$O((n + occ)\log^2 d)$
Hon et al. [39]	$O(m\log\sigma + \log d)$	$O(n\log d + occ)$
Feigenblat et al. [34]	$O(\frac{1}{\epsilon}m\log d)$	$O(n\log\log d\log\sigma + occ)$
Meyer† [59]	$O(l_{\max} \cdot d \cdot \sigma)$	$O(n\log\sigma + occ)$
Tsuda et al. [67]	$O(z\log\sigma)$	$O(n\log\sigma + occ)$
DAWG based†	$O(m\log\sigma)$	$O(n\log\sigma + occ)$
DAWG based	$O(\sigma m + \frac{\log d}{\log\log d})$	$O(n(\frac{\log d}{\log\log d} + \log\sigma) + occ\frac{\log d}{\log\log d})$
AC-automaton based†	$O(m\log\sigma + u_f + u_o)$	$O(n\log\sigma + occ)$
AC-automaton based	$O(\sigma m + u_f + u_o)$	$O(n\log\sigma + occ)$

is optimal in the sense that *any* algorithm which explicitly maintains the AC-automaton must use at least $\Omega(u_f + u_o)$ time to update the automaton. Last, we give tight upper and lower bounds on u_f and u_o in the worst case.

4.1 Key idea of proposed algorithm

Meyer [59] and Tsuda et al. [67] used the inverse of the failure function to update the AC-automaton. Although the inverse failure function can be stored in a total of $O(d)$ space, it is not trivial whether one can efficiently access and/or update the inverse failure function, because the number of inverse failure links of each state may change dynamically and can be as large as the number of states in the AC-automaton. For instance, let us consider $\mathsf{AC}(D)$ for $D = \{\mathtt{baaaac}\}$ over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ in Fig. 4.2 (a). Its root is pointed by 6 failure links. When a new pattern $\mathtt{c}$ is inserted to D, then the above algorithms first create a

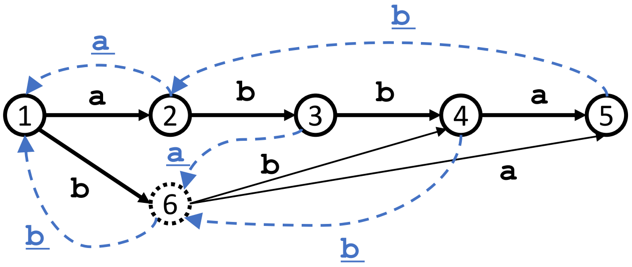


Figure 4.1: DAWG({abba}). Solid-line circles show trunk nodes and the dashed-line circle shows a non-trunk node. Thick solid lines, thin solid lines, and dashed lines show primary edges, secondary edges and suffix links, respectively.

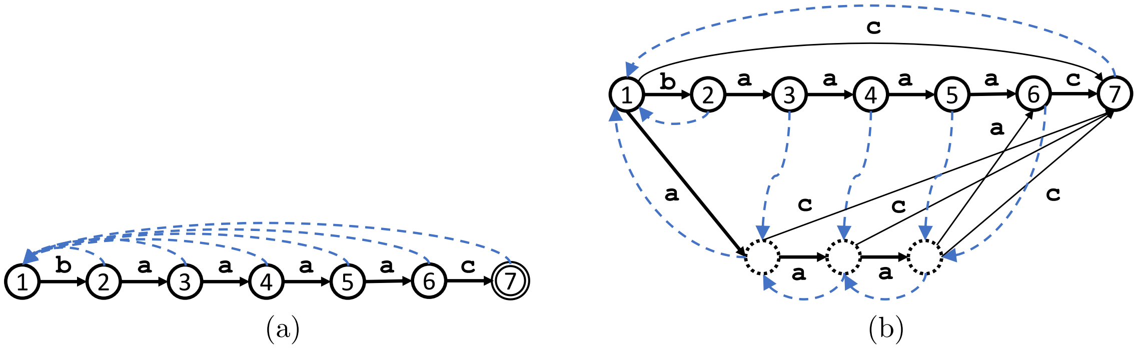


Figure 4.2: (a) AC({baaaac}) and (b) DAWG({baaaac}).

new state s, a new transition from the root to s, and a failure link from s to the root. The real difficulty arises when they try to find which suffix links should be updated to point at s; they must follow all the 6 inverse failure links from the root and get 6 states numbered 2, 3, . . . , 7, and check whether there is an edge labeled c from each of them, although only one state 7 should be updated. Ishizaki and Toyama [45] introduced an auxiliary tree structure called an *expect tree* to reduce the number of the candidates and showed some experimental results, but no theoretical analysis is provided. Unfortunately, their algorithm behaves the same for the above example. Therefore, maintaining the inverse failure links to update the AC-automaton might be inefficient.

In order to overcome this difficulty, we pay our attention to the suffix links of DAWG(D), instead of the failure links of AC(D). It is known (see, e.g. [12, 13, 30, 29]) that the inverse suffix links of all nodes in DAWG(D) form the suffix tree of the reversed patterns in D, so that for any node v in DAWG(D), each suffix link pointing at v is labeled by a distinct symbol which is the first symbol of the edge label in the suffix tree. Formally, the label of a suffix link is defined as follows. Let xy be the longest string in $[xy]_D$, y be the longest string in $[y]_D$, and $\mathsf{slink}([xy]_D) = \mathsf{slink}([y]_D)$, the label of this suffix link is $x[|x|]$. In Fig. 4.1, the label of each suffix link is showed by an underlined symbol. Therefore, the number of suffix links that point at v is at most σ, and the inverse suffix links can be accessed and updated in $O(\log \sigma)$ time using $O(d)$ total space. This means that when a pattern of length m is inserted to or deleted from the dictionary, the inverse suffix links can be maintained in $O(m \log \sigma)$ time.

By the properties of AC-automata and DAWGs, for each state s in $\mathsf{AC}(D)$, there exists a unique node v in $\mathsf{DAWG}(D)$ that corresponds to s, so that $\mathsf{AC}(D)$ can be consistently embedded into $\mathsf{DAWG}(D)$. Because $s \in \mathit{Pref}(D)$, the corresponding node v is a trunk node and each trunk node has its corresponding state. Therefore, there exists a one-to-one mapping from the set of trunk nodes in $\mathsf{DAWG}(D)$ to the states of $\mathsf{AC}(D)$. We denote this mapping by $s = \pi(v)$, where v is a DAWG trunk node and s is the corresponding AC-automaton state. We denote $v = \pi^{-1}(s)$ iff $s = \pi(v)$. Fig. 4.3 (a) and (b) show the AC-automaton and DAWG of $D = \{\mathtt{abba}, \mathtt{aca}, \mathtt{cbb}\}$, respectively, where each number in nodes and states expresses the correspondence.

Our algorithm to follow will make a heavy use of the following lemma, which characterizes the relationship between the states of $\mathsf{AC}(D)$ and the trunk nodes of $\mathsf{DAWG}(D)$.

Lemma 4.2. *Let s and s' be any states in $\mathsf{AC}(D)$, and let $v = \pi^{-1}(s)$ and $v' = \pi^{-1}(s')$ be corresponding trunk nodes in $\mathsf{DAWG}(D)$. Then, $s' = \mathsf{flink}(s)$ iff there exists an integer $k \geq 1$ such that $v' = \mathsf{slink}^k(v)$, and when $k \geq 2$, $\mathsf{slink}^i(v)$ is a non-trunk node for all $1 \leq i < k$.*

Proof: ($\Longrightarrow$) Suppose $s' = \mathsf{flink}(s)$. Then, by definition, s' is a proper suffix of s. Hence there exists an integer $k \geq 1$ such that $v' = \mathsf{slink}^k(v)$. Also, by definition, k is the smallest such that $\mathsf{slink}^k(v) \in \mathit{Pref}(D)$. Hence, when $k \geq 2$, $\mathsf{slink}^i(v)$ is a non-trunk node for all $1 \leq i < k$.

Algorithm 1: Dynamic dictionary matching algorithm by using a DAWG

```
Input: A text string $T$.
Output: Occurence positions of all pattern in the dictionary.
$activeNode \leftarrow root$;
for $1 \leq i \leq n$ do
    while checkTransition($activeNode$, T[i]) and $activeNode \neq root$ do
        $activeNode \leftarrow$ slink($activeNode$);
    if checkTransition($activeNode$, T[i]) then
        $activeNode \leftarrow trans(activeNode, T[i])$;
    $outNode \leftarrow activeNode$;
    if $outNode$ is marked then
        output($outNode$);
    while $NMA(outNode) \neq$ NULL do
        $outNode \leftarrow NMA(outNode)$;
        output($outNode$);
Function checkTransition($node$, c)
    if $node$ is not a trunk node then return false;
    if $trans(node, c) =$ NULL then return false;
    if $c$ is a secondary edge then return false;
    if $trans(node, c)$ is not a trunk node then return false;
    return true;
```

($\Longleftarrow$) Suppose there exists an integer $k \geq 1$ such that $v' = \mathsf{slink}^k(v)$. When $k = 1$, clearly $s' = \mathsf{flink}(s)$. When $k \geq 2$ and $\mathsf{slink}^i(v)$ is a non-trunk node for all $1 \leq i < k$, then k is the smallest integer such that $v' = \mathsf{slink}^k(v)$ is a trunk node. Hence $s' = \mathsf{flink}(s)$. ■

It is known that DAWGs can be used for solving the pattern matching problem with a single pattern [26]. However, it is not trivial to maintain the output function efficiently for dynamic and multiple patterns, as is pointed out by Kucherov and Rusinowitch [55]. In the next section, we shall show our algorithm which efficiently maintains the output function on the DAWG.

4.2 Dynamic Dictionary Matching by using DAWG

In this section, we will describe how to perform dynamic dictionary matching with the DAWG. This algorithm is a simple modification of Kucherov and Rusinowitch's algorithm [55] for matching multiple strings with variable length don't-care symbols.

First we will discuss the time complexity to update the DAWG in the semi-dynamic and dynamic settings. As it was shown in [13] the DAWG can be updated in $O(m \log \sigma)$ *amortized* time for an insertion of a pattern of length m in the semi-dynamic setting. For the dynamic setting, Kucherov and Rusinowitch [55] gave an algorithm which deletes a pattern from the dictionary, and claimed that the update time for deletion and insertion is the same as in the semi-dynamic setting. However, in what follows we show that this is not true when the alphabet size is super-constant. Namely, the number of edges to be constructed when we split a DAWG node can be amortized constant by the *total length* of the input strings in the semi-dynamic setting, but this amortization argument does not hold in the dynamic setting. That is, we obtain the following lower bound for updating the DAWG in the dynamic setting.

Lemma 4.3. *In the dynamic setting where both insertion and deletion of patterns are supported, there exists a family of patterns such that $\Omega(\sigma m)$ time is needed when updating the DAWG for insertion and deletion of each pattern.*

Proof: To show an $\Omega(\sigma m)$ lower bound, consider a pattern $P = (\mathtt{ba})^{\frac{m}{2}}$ and an initial dictionary $D = \{(\mathtt{ab})^i \mathtt{a}^j c \mid 1 \leq i \leq \frac{m}{2},\ j \in \{0, 1\},\ c \in \Sigma \setminus \{\mathtt{a}, \mathtt{b}\}\}$ of size $d = \Theta(\sigma m^2)$. We insert P to the dictionary and update $\mathsf{DAWG}(D)$ into $\mathsf{DAWG}(D \cup \{P\})$. In this case we need to split a node each time we read a symbol from P, and construct $\sigma - 2$ edges labeled by $c \in \Sigma \setminus \{\mathtt{a}, \mathtt{b}\}$ from the new node. Hence, we need to create $\Omega(\sigma m)$ edges when we update the DAWG. Moreover, the same computation time $\Omega(\sigma m)$ is required when we delete the same pattern P from $D \cup \{P\}$ and update the DAWG.

If we repeat this operation more than m times by inserting and deleting Pp, we can not amortize the update cost by the size of the dictionary. Therefore, we need $\Omega(\sigma m)$ operations to update the DAWG when inserting or deleting a pattern. ■

The above lower bound is tight, namely, below we will show a matching upper bound for updating the DAWG in the dynamic setting.

Lemma 4.4. *In the dynamic setting where both insertion and deletion of patterns are supported, the DAWG can be updated in $O(\sigma m)$ time for insertion and deletion of patterns.*

Proof: To show the upper bound, we will evaluate the number of edges and suffix links that are traversed and/or created during the update.

Let us first consider the insertion operation. Let P be a pattern of length m to be inserted to the dictionary. Suppose that the prefix $P[1 : i-1]$ of P has already been inserted to the DAWG for $1 \leq i \leq m$. Let v be the DAWG node that represents $P[1 : i-1]$. There are three cases for the next pattern symbol $P[i]$:

(1) There is a primary out-going edge of v labeled with $P[i]$. In this case no new edge or node is created, and it takes $O(\log \sigma)$ time to traverse this primary edge.

(2) There is no out-going edge of v labeled with $P[i]$. In this case, a new sink node and a new edge from v to this new sink labeled with $P[i]$ are created. Then, the algorithm follows a chain of suffix links from v and insert new edges leading to the new sink labeled with $P[i]$, until finding the first node which has an out-going edge labeled with $P[i]$.

(3) There is a secondary out-going edge of v labeled with $P[i]$. Let u be the node that is reachable from v via the edge labeled with $P[i]$. This node u gets split into two nodes u and u', and at most σ out-going edges of the original node u are copied to u'.

It is clear that Case (1) takes $O(\log \sigma)$ time per symbol. At most i new edges can be introduced in Case (2), but it follows from [12] that the total number of suffix links that are traversed is $O(m)$ for all m symbols of P. Hence, Case (2) takes $O(\log \sigma)$ amortized time per symbol. It is clear that Case (3) takes $O(\sigma)$ time. Overall, a pattern of length m can be inserted to the DAWG in $O(\sigma m)$ total time.

The deletion operation can also be performed in $O(\sigma m)$ time, since the deletion can be done in the same complexity as insertion by reversing the insertion procedure (see also Kucherov and Rusinowitch's result [55]). ■

Next, we will describe how to find the occurrences of the patterns in the text by using the DAWG. We will use a nearest marked ancestor (NMA) data structure on the inverse suffix link tree. In the NMA problem on a rooted tree, each node of the tree is either marked or unmarked. The NMA query returns the nearest marked ancestor of a given query node v in the tree, or returns NULL if v has no marked ancestor. The semi-dynamic NMA problem allows for marking operation only, while the dynamic NMA problem allows for both marking and unmarking operations. New leaves can be added to the tree in both of the problems, and existing leaves can be removed in the dynamic problem. There is a semi-dynamic NMA data structure [73] which allows for NMA queries, marking unmarked nodes, and inserting new leaves in amortized $O(1)$ time each. For the dynamic NMA problem, there is a data structure which permits NMA queries and both marking and unmarking operations in worst-case $O(\log t / \log \log t)$ time, and inserting new leaves in amortized $O(1)$ time, where t is the size of the tree [2, 3]. Both of the data structures use $O(t)$ space and $O(t)$ preprocessing time.

In our dictionary pattern matching algorithm using the DAWG, we mark each node v of the inverse suffix link tree iff v is a DAWG node that represents a pattern in the dictionary[1]. Now, for a given node w in the DAWG, we can find all patterns in the dictionary that are suffixes of w by performing NMA queries from w on the inverse suffix link tree as follows. If w itself is marked, then we output it. Then, we perform NMA

[1] Kucherov and Rusinowitch [55] used Sleator and Tarjan's link-cut tree data structure [64] to maintain a dynamic forest induced from the inverse suffix link tree. Our important observation here is that essentially the same operations and queries in this application can be more efficiently supported with NMA data structures.

queries in the inverse suffix link tree from w, until we find a node that has no marked ancestor. This allows us to skip all unmarked nodes in the path from w to the node, and we output all marked nodes found by NMA queries in this path.

Algorithm 1 shows a pseudo-code of our algorithm for dynamic dictionary matching by using the DAWG. The algorithm only uses the trunk nodes and primary edges to perform pattern matching. Therefore, when the algorithm reads a symbol c from the text, it checks whether or not there is a primary edge which is labeled with c and leads to a trunk node by using a function $\mathsf{checkTransition}(node, \mathsf{c})$. If there is no such node, then the algorithm follows a chain of suffix links until it reaches a trunk node, and then performs the same procedure as above. Thus, the suffix links of the DAWG replaces the failure links of the corresponding AC-automaton. The correctness is immediately justified by Lemma 4.2. As soon as the algorithm finds a primary edge which is labeled with c and leads to a trunk node, it checks whether there is an occurrence of any patterns in the dictionary by using NMA queries from this destination trunk node, as described previously. This procedure is used as a substitute for the output function of the AC-automaton.

Consider inserting a new pattern P to the dictionary. If v is the DAWG node which represents P, then v is newly marked in the inverse suffix link tree, and v is the only node that gets marked in this stage. Hence, exactly one unmarked node gets marked per inserted pattern. For the same reasoning, exactly one marked node gets unmarked per deleted pattern. To delete an existing pattern P from the dictionary and hence from the DAWG, we can use Kucherov and Rusinowitch's algorithm [55] which takes $O(\sigma m)$ time due to Lemmas 4.3 and 4.4, where m is the length of P.

Overall, we obtain the following.

Theorem 4.5. *In the semi-dynamic setting where only insertion of patterns is supported, the DAWG-based algorithm supports insertion of patterns in $O(m \log \sigma)$ time and pattern matching in $O(n \log \sigma + occ)$ time.*

In the dynamic setting where both insertion and deletion of patterns are supported, the

DAWG-based algorithm supports insertion/deletion in $O(\sigma m + \log d / \log\log d)$ time and pattern matching in $O(n(\log d / \log\log d + \log \sigma) + occ \log d / \log\log d)$ time. The size of both data structures is $O(d)$.

Proof: The update times and space requirements of both of the semi-dynamic and dynamic versions should be clear from Lemmas 4.3, 4.4 and the above arguments.

For pattern matching, we need to perform at least one NMA query each time a symbol from a text is scanned, and need to perform an NMA query each time an occurrence of a pattern is found. Hence, it takes $O(n \log \sigma + occ)$ time for the semi-dynamic setting and $O(n(\log d / \log\log d + \log \sigma) + occ \log d / \log\log d)$ time for the dynamic setting. ■

4.3 AC-Automaton Update Algorithm

In this section we will describe how to perform dynamic dictionary matching by using the AC-automaton and the DAWG for the dictionary. Our algorithm performs pattern matching in exactly the same manner as the original AC-algorithm, while updating the AC-automaton dynamically with the aid of the DAWG upon insertion/deletion of patterns. We will describe how to modify the AC-automaton by using the DAWG. Note that we can simulate the AC-automaton with the DAWG augumented with the output function. However, we will explicitly use the AC-automaton since it makes the pattern matching algorithm simpler.

4.3.1 Pattern insertion algorithm

We consider inserting a new pattern p of length m into the dictionary D, and we denote the new dictionary by $D' = D \cup \{P\} = \{P_1, P_2, \ldots, P_r, P\}$. It is known that $\mathsf{DAWG}(D)$ can be constructed in $O(d \log \sigma)$ time, and can be updated to $\mathsf{DAWG}(D')$ online in $O(m \log \sigma)$ amortized time [13]. We update $\mathsf{AC}(D)$ to $\mathsf{AC}(D')$ by using $\mathsf{DAWG}(D)$, and then update $\mathsf{DAWG}(D)$ to $\mathsf{DAWG}(D')$. We also add $\mathsf{weight}(v)$ to each state v of $\mathsf{AC}(D)$ that is the

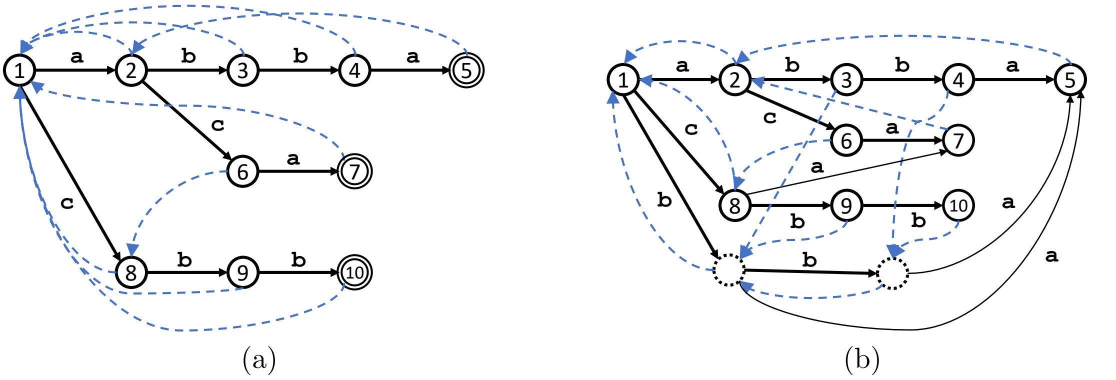


Figure 4.3: For a dictionary $D = \{\mathtt{abba}, \mathtt{aca}, \mathtt{cbb}\}$ (a) $\mathsf{AC}(D)$, and (b) $\mathsf{DAWG}(D)$.

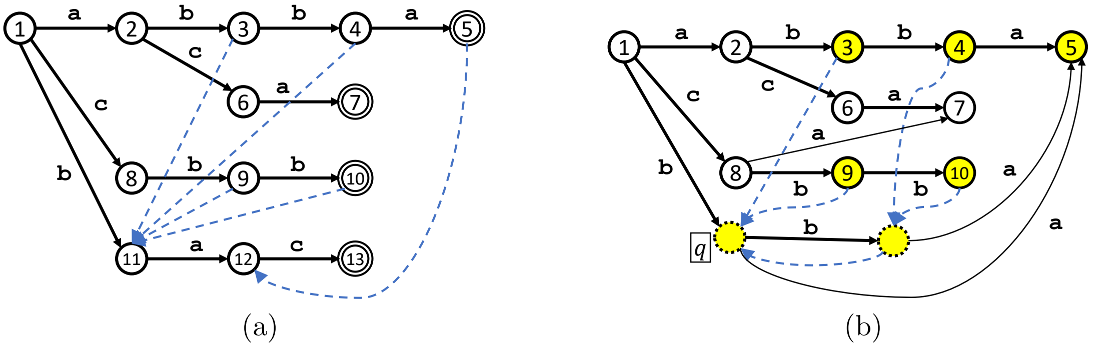


Figure 4.4: Illustration of updating process when inserting a pattern $p = \mathtt{bac}$ into the dictionary $D = \{\mathtt{abba}, \mathtt{aca}, \mathtt{cbb}\}$. Compare them with Fig. 4.3. (a) The updated automaton $\mathsf{AC}(D')$, where the only updated failure links are shown. (b) In $\mathsf{DAWG}(D)$, the only suffix links that are used for the update are shown, and the visited nodes are colored.

number of occurrences v as prefix in D. We will use $\mathsf{weight}(v)$ as a reference counter to determine whether v should be deleted or not in the deletion algorithm later.

The key point of our algorithm is to update the output and failure functions of $\mathsf{AC}(D)$ in linear time with respect to the number of states that should be modified. The *goto* function can be updated easily by adding a new transition for a new state in the same way as in the AC-automaton construction algorithm. We then update the output and failure functions efficiently by using inverse suffix links of $\mathsf{DAWG}(D)$. Algorithm 4 updates $\mathsf{AC}(D)$ when a new pattern is inserted to D, and Algorithms 2 and 3 find the states on which the output and failure functions should be updated, respectively.

For any node v in $\mathsf{DAWG}(D)$, let $\mathit{isuf}(v) = \{x \mid \mathsf{slink}(x) = v\}$ be the set of its inverse suffix links. The set $\mathit{isuf}(v)$ for each v is stored in ordered array v_a as described in

Algorithm 2: Algorithm to find the states on which the output function should be updated

```
Function getOutStates($P$)
    $outStates \leftarrow \emptyset$;
    $activeNode \leftarrow root$;
    for $1 \leq i \leq m$ and $activeNode \neq$ NULL do
        $activeNode \leftarrow trans(activeNode, P[i])$;
    if $activeNode \neq$ NULL then
        $queue \leftarrow \emptyset$;
        push $activeNode$ to $queue$;
        while $queue \neq \emptyset$ do
            pop $node$ from $queue$;
            if $node$ is a trunk node then
                $outStates \leftarrow outStates \cup \{\pi(node)\}$
            for $lnode \in isuf(node)$ do
                push $lnode$ to $queue$;
    return $outStates$;
```

Section 4.1. For the new pattern P, we can divide P to $P = xyz$ and categorize the prefixes of p into three categories, so that for any i, j, k with $1 \leq i \leq |x| < j \leq |x| + |y| < k \leq m$;

1. $P[1 : i]$ exists both in $\mathsf{AC}(D)$ and $\mathsf{DAWG}(D)$,

2. $P[1 : j]$ does not exist in $\mathsf{AC}(D)$ but exists in $\mathsf{DAWG}(D)$, and

3. $P[1 : k]$ exists in neither $\mathsf{AC}(D)$ nor $\mathsf{DAWG}(D)$.

To update both output and failure functions of $\mathsf{AC}(D)$ to $\mathsf{AC}(D')$ we only use nodes in $\mathsf{DAWG}(D)$ that represent prefixes in the second category. Algorithm 2 follows inverse suffix links of a node representing P recursively in $\mathsf{DAWG}(D)$, in order to find all the states in $\mathsf{AC}(D)$ on which the output function needs to be updated. On the other hand, Algorithm 3 follows inverse suffix links of nodes that represent $P[i : j]$ for $|x| < j \leq |x|+|y|$ (category 2) recursively, until it reaches a trunk node u, and then saves the state $s = \pi(u)$ that corresponds to the trunk node to update its failure link later.

Fig. 4.4 illustrates an example, where we insert a pattern $P =$ `bac` into the dictionary

Algorithm 3: Algorithm to find the states whose failure link should be updated

```
1  Function getFailStates($P, start$)
2      $stack \leftarrow \emptyset$;
3      $activeNode \leftarrow root$;
4      for $1 \leq i \leq m$ and $activeNode \neq$ NULL do
5          $activeNode \leftarrow trans(activeNode, P[i])$;
6          if $i \geq start$ and $activeNode \neq$ NULL then
7              push $(activeNode, i)$ to $stack$;
8      while $stack \neq \emptyset$ do
9          pop $(activeNode, i)$ from $stack$;
10         $queue \leftarrow \emptyset$;
11         push $activeNode$ to $queue$;
12         while $queue \neq \emptyset$ do
13             pop $node$ from $queue$;
14             if $node$ is a trunk node then
15                 push $(\pi(node), i)$ to $failStates$;
16             if $node$ is not marked then
17                 mark $node$;
18                 if $node$ is a branch node then
19                     for $lnode \in isuf(node)$ do
20                         push $lnode$ to $queue$;
21     return $failStates$;
```

$D = \{$abba, aca, cbb$\}$. First, we create new states 11, 12, and 13. The string b is represented by node q in DAWG(D), and by the new state 11 in AC(D'), thus there is at least one state whose failure link should be updated to point at the state 11. We will explain how to find these states below. Similarly, we know that at least one failure link should be updated to point at the state 12, because the string ba represented by the state 12 in AC(D') is also represented by node 5 in DAWG(D). However, the string bac, which is represented by the new state 13, is not represented in DAWG(D), thus we know that there is no state whose failure link should be updated to state 13. As a result, we have the set $\{11, 12\}$ of states. (Lines 4–7 in Algorithm 3)

We now explain how to find states whose failure links should be updated. We begin by the deepest state in $\{11, 12\}$, that is, state 12. We search the states from node 5 in

Algorithm 4: Pattern insertion algorithm of AC-automaton

```
Function InsertPattern($P$)
    $activeState \leftarrow rootState$;
    $newStatesSet \leftarrow \emptyset$;
    $\mathsf{weight}(activeState) \leftarrow \mathsf{weight}(activeState) + 1$;
    for $1 \leq i \leq m$ do
        if $\delta(activeState, P[i]) \neq$ fail then
            $activeState \leftarrow \delta(activeState, P[i])$;
        else
            create $newState$;
            $\delta(activeState, P[i]) \leftarrow newState$;
            $activeState \leftarrow newState$;
            $newStatesSet \leftarrow newStatesSet \cup \{newState\}$;
        $\mathsf{weight}(activeState) \leftarrow \mathsf{weight}(activeState) + 1$;
        if $i = m$ then
            $\mathsf{output}(newState) \leftarrow \mathsf{output}(newState) \cup \{P\}$;
    $failStates \leftarrow \underline{\mathsf{getFailStates}}(P, m - |newStatesSet| + 1)$;
    while $failStates \neq \emptyset$ do
        pop $(s, i)$ from $failStates$;
        $\mathsf{flink}(s) \leftarrow newStatesSet[i - |newStatesSet| + 1]$;
    $activeState \leftarrow rootState$;
    for $1 \leq i \leq m$ do
        if $\delta(activeState, P[i]) \in newStatesSet$ then
            $failureState \leftarrow \mathsf{flink}(activeState)$;
            while $goto(failureState, P[i]) =$ fail do
                $failureState \leftarrow \mathsf{flink}(failureState)$;
            $activeState \leftarrow \delta(activeState, P[i])$;
            $\mathsf{flink}(activeState) \leftarrow failureState$;
            $\mathsf{output}(activeState) \leftarrow \mathsf{output}(activeState) \cup \mathsf{output}(failureState)$;
        else
            $activeState \leftarrow \delta(activeState, P[i])$;
    $outStates \leftarrow \underline{\mathsf{getOutStates}}(P)$;
    for $s \in outStates$ do $\mathsf{output}(s) \leftarrow \mathsf{output}(s) \cup \{P\}$ ;
```

$\mathsf{DAWG}(D)$, which represents the same string `ba` as state 12 in $\mathsf{AC}(D')$. When searching from node 5, we do not search further because node 5 is a trunk node. Therefore, we update the failure link of state 5 to state 12. Next, to find states whose failure links should

Algorithm 5: Pattern deletion algorithm of AC-automaton

```
Function DeletePattern($P$)
    $activeState \leftarrow rootState$;
    $deleteStatesSet \leftarrow \emptyset$;
    for $1 \leq i \leq m$ do
        $activeState \leftarrow \delta(activeState, P[i])$;
        if $\mathsf{weight}(activeState) = 0$ then
            $deleteStatesSet \leftarrow deleteStatesSet \cup activeState$;
        else
            $\mathsf{weight}(activeState) \leftarrow \mathsf{weight}(activeState) - 1$;
    $failStates \leftarrow \underline{\mathsf{getFailStates}}(P, m - |deleteStatesSet| + 1)$;
    while $failStates \neq \emptyset$ do
        pop $(s, i)$ from $failStates$;
        $\mathsf{flink}(s) \leftarrow \mathsf{flink}(newStatesSet[i - (m - |newStatesSet|) + 1])$;
    $activeState \leftarrow rootState$;
    $outStates \leftarrow \underline{\mathsf{getOutStates}}(P)$;
    for $s \in outStates$ do
        $\mathsf{output}(s) \leftarrow \mathsf{output}(s) \setminus \{P\}$;
    for $s \in deleteStatesSet$ do
        delete $s$;
```

be updated to state 11, we search the states from node q in $\mathsf{DAWG}(D)$, which represents the same string $\mathtt{b}$ as state 11 in $\mathsf{AC}(D')$. By following the inverse suffix links recursively from node q until reaching a trunk node, we get the set $\{3, 4, 9, 10\}$ of trunk nodes (see Fig. 4.4 (b)). Therefore, we update the failure links of states 3, 4, 9, and 10 to state 11. (Lines 8–20)

4.3.2 Pattern deletion algorithm

We consider deleting a pattern P_i of length m from the dictionary D, and we denote the new dictionary by $D' = D \setminus \{P_i\} = \{P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_r\}$. From Lemma 4.4 we can delete a pattern from $\mathsf{DAWG}(D)$ in $O(\sigma m)$ time. We update $\mathsf{AC}(D)$ to $\mathsf{AC}(D')$ by using $\mathsf{DAWG}(D)$, and then update $\mathsf{DAWG}(D)$ to $\mathsf{DAWG}(D')$. The proposed deletion algorithm is also update the output and failure functions of $\mathsf{AC}(D)$ in linear time with respect to the number of states that should be modified.

Algorithm 5 shows the proposed deletion algorithm. First, the algorithm finds which states should be deleted. The algorithm finds the states by decreasing the weight of states which represent prefixes of P. The algorithm will delete the states whose weight becomes zero, which means those states do not represent any prefixes of patterns in D'.

After the algorithm has found the states which should be deleted, it will update the states whose failure links should be updated. A state should be updated if the failure link of the state is pointing at one of the nodes that will be deleted. Such states can be found by traversing reverse failure links of the states. From Lemma 4.2 we can use inverse suffix links of the DAWG instead of inverse failure links of the AC-automaton to find the states. The algorithm uses $\mathsf{getFailStates}(P, start)$ in Algorithm 3 to find the states and update them from the states of which the suffix links point to shallower states.

Next, the algorithm will update the output function of the AC-automaton. The output function of a state should be updated if and only if p is a suffix of the string that is represented by the state. The algorithm uses $\mathsf{getOutStates}(P)$ in Algorithm 2 to find the states whose output function should be updated. Last, the algorithm will delete the respective states.

4.3.3 Correctness of the algorithms

We now show the correctness of Algorithms 2 and 3.

Lemma 4.6. *Algorithm 2* correctly *returns the set of states on which output functions should be updated.*

Proof: When a new pattern P is inserted to a dictionary D, we have to update the output function of every state s in $\mathsf{AC}(D)$ such that P is a suffix of the string s. If there is no node in $\mathsf{DAWG}(D)$ representing P, we know that no such a string s exists in D. Otherwisem, let s_P be a new state created in $\mathsf{AC}(D')$ to represent the pattern P. The output function of some state s should be updated if and only if s_P is reachable from s

via a chain of failure links. From Lemma 4.2, for nodes $u = \pi^{-1}(s)$ and $v_P = [P]_D$, we have $v_P = \mathsf{slink}^i(u)$ for some i. Therefore, $s = \pi(u)$ can be found by following inverse suffix links from v_P recursively. ■

Lemma 4.7. *Algorithm 3 correctly returns the set of states whose failure links should be updated.*

Proof: By arguments similar to the proof of Lemma 4.6, all the states that should be updated are reachable via chains of inverse suffix links from the nodes in $\mathsf{DAWG}(D)$ that correspond to the new states in $\mathsf{AC}(D')$. Next, we will show that Algorithm 3 only returns the set S of the states that should be updated. Let x be a new state and $t = [x]_D$ be a node that represents the string x. Assume that S contains a state s that can be reached by following inverse failure links from x recursively but should not be updated. Let $u = \pi^{-1}(s)$ and $v = \pi^{-1}(\mathsf{flink}(s))$ be trunk nodes in $\mathsf{DAWG}(D)$ corresponding to s and $\mathsf{flink}(s)$, respectively. From Lemma 4.2, $v = \mathsf{slink}^i(u)$ and $t = \mathsf{slink}^j(v)$ for some i and j. Since Algorithm 3, started from t, stops a recursive search after reaching a trunk node (v in this case), it would not find u. Therefore, $s = \pi(u) \notin S$. ■

4.4 Algorithm Complexity Analysis

We now show the time complexity of Algorithms 2 and 3.

Lemma 4.8 ([12])**.** *A string $x \in Substr(D)$ is the longest member of $[x]_D$ if and only if either $x \in Pref(D)$ or $ax, bx \in Substr(D)$ for some distinct $a, b \in \Sigma$.*

Lemma 4.9. *For any non-trunk node in DAWG, there exist at least two suffix links that point at it.*

Proof: Let $[x]_D$ be any non-trunk node in $\mathsf{DAWG}(D)$ and $x \in Substr(D)$ be the longest member of $[x]_D$. Then $x \notin Pref(D)$ because $[x]_D$ is a non-trunk node. By Lemma 4.8, there exist two distinct $a, b \in \Sigma$ such that $ax, bx \in \mathit{Substr}(D)$. Because x is the longest member of $[x]_D$, we have $[ax]_D \neq [x]_D$. Thus, $\mathsf{slink}([ax]_D) = [x]_D$ because x is a suffix of ax. Similarly, $\mathsf{slink}([bx]_D) = [x]_D$. Because $[ax]_D \neq [bx]_D$, the non-trunk node $[x]_D$ is pointed by at least two suffix links. ■

Lemma 4.10. *Algorithm 2 runs in $O(m \log \sigma + u_o)$ time, where u_o is the number of states on which output function should be updated.*

Proof: At first, Algorithm 2 finds the node v representing the pattern P, by traversing the nodes from the root, in Lines 4–5. It takes $O(m \log \sigma)$ time. If it failed, done. Then we analyze the runnning time consumed in Lines 6–14 by counting the number ℓ of visited nodes in $\mathsf{DAWG}(D)$. These nodes form a tree, rooted at v and connected by inverse suffix links chains. Let b (resp. t) be the number of non-trunk (resp. trunk) nodes in this tree, and let q be the number of nodes (either non-trunk or trunk) that are child nodes of some non-trunk node. Because every non-trunk node has at least two child nodes by Lemma 4.9, we have $2b \leq q$, and obviously $q \leq b + t$. Thus, $b \leq t$, which yields that $\ell = b + t \leq 2t = 2|\mathit{outStates}| = 2u_o$. Therefore, Algorithm 2 runs in $O(m \log \sigma + u_o)$ time. ■

Lemma 4.11. *Algorithm 3 runs in $O(m \log \sigma + u_f)$ time, where u_f is the number of states whose failure links should be updated.*

Proof: At first, Algorithm 3 finds the set V of nodes representing the pattern $P[1 : j]$ for $1 \leq j \leq m$ such that $P[1 : j]$ does not exist in $\mathsf{AC}(D)$ but does exist in $\mathsf{DAWG}(D)$, by traversing the nodes from the root, in Lines 4–7. The algorithm saves the nodes in a stack, because the algorithm will search from the deepest node. This takes $O(m \log \sigma)$ time. Then we analyze the running time consumed in Lines 8–20 by counting the number ℓ of visited nodes in $\mathsf{DAWG}(D)$. These nodes form a forest, where each tree is rooted by some node in V and connected by inverse suffix link chains, where some node in V can be an inner node of a tree rooted by another in V. In this case, we mark the nodes that have been visited, so each node is visited at most twice. Let b (resp. t) be the number of non-trunk (resp. trunk) nodes in this forest, and let q be the number of nodes (either non-trunk or trunk) that are child nodes of some non-trunk node. Because every non-trunk node has at least two child nodes by Lemma 4.9, we have $2b \leq q$, and obviously $q \leq b + t$. Thus, $b \leq t$, which yields that $\ell = b + t \leq 2t = 2|\mathit{failStates}| = 2u_f$. ■

Theorem 4.12. *AC-automaton can be updated for each pattern in $O(m \log \sigma + u_f + u_o)$ time.*

Proof: The goto, failure and output functions of newly created states can be calculated in $O(m \log \sigma)$, similarly to the original AC-automaton construction algorithm. From Lemmas 4.10 and 4.11, the output and failure functions on existing states can be updated in $O(m \log \sigma + u_o)$ and $O(m \log \sigma + u_f)$, respectively. Therefore, AC-automaton can be updated in $O(m \log \sigma + u_f + u_o)$ time in total. ■

Note that *any* algorithm which explicitly updates the AC-automaton requires at least $\Omega(m + u_f + u_o)$ time. Hence, the bound in the above theorem is optimal except for the term $\log \sigma$ which can be ignored for constant alphabets. As it was stated in introduction, u_f and u_o can be considerably small in several cases.

On the other hand, the remaining question is how large u_f and u_o can be in the worst case. The next theorem shows matching upper and lower bounds on u_f and u_o.

Theorem 4.13. *For any pattern of length m, $u_f = O(km)$ and $u_o = O(km)$, where k is the number of patterns to insert to the current dictionary. Also, there exists a family of patterns for which $u_f = \Omega(km)$ and $u_o = \Omega(km)$.*

Proof: In this proof, we only show bounds for u_f; however, the same bounds for u_o can be obtained similarly.

First, we show an upper bound $u_f = O(km)$. We begin with an empty dictionary and insert patterns to the dictionary. Let d be the total length of the pattens in the dictionary after adding all patterns, and let $\mathit{total\_u_f}$ be the total number of AC-automaton states whose failure links need to be updated during the insertion of all patterns. If k is the number of patterns to insert, then clearly $total\_u_f \leq kd$ holds. Hence, the number of failure links to update per symbol is $total\_u_f/d \leq k$. This implies that for any pattern of length m, the number u_f of failure links to update is $O(km)$.

To show a lower bound $u_f = \Omega(km)$, consider an initial dictionary $D = \{c_i a^k \mid 1 \leq i \leq x\}$ of $x \geq 1$ patterns, where $k \geq 1$ and $c_i \neq c_j$ for any $1 \leq i \neq j \leq x$. For each $j = 1, 2, \ldots, k$ in increasing order, we insert a new pattern a^j to the dictionary. Then, the total number $total\_u_f$ of failure links to update for all a^j's is

$$total\_u_f = xk + x(k-1) + x(k-2) + \cdots + x = xk(k+1)/2.$$

Let d_{add} be the total length of patterns to insert to the initial dictionary, and d the total length of the patterns after adding all patterns to the initial dictionary. Then $d = xk + d_{add} = xk + k(k+1)/2$. Hence, the number of failure links to update for each symbol in the added patterns a^j's is

$$\frac{total\_u_f}{d_{add}} < \frac{total\_u_f}{d} = \frac{xk(k+1)}{2xk + k(k+1)} = \frac{x(k+1)}{2x+k+1} = \Omega\Big(\frac{xk}{x+k}\Big),$$

which becomes $\Omega(k)$ by choosing $x = \Omega(k)$. Hence, for each $1 \leq m \leq k$, when we add pattern a^m of length m to the dictionary, $u_f = \Omega(km)$ failure links need to be updated. ■

The arguments in the above proof consider the semi-dynamic case where only insertion of new patterns in supported. However, if we delete all patterns after they have been inserted, then exactly the same number of failure links need to be updated. Hence, the same matching upper and lower bounds hold also for the dynamic case with both insertion and deletion of patterns.

Chapter 5

Parameterized Pattern Matching Algorithm

The parameterized pattern matching introduced by Baker [11] is a variant of pattern matching that focuses on a structure of strings. Let Σ and Π be two disjoint sets of symbols. A string over $\Sigma \cup \Pi$ is called a *parameterized string* (p-string for short). Given p-strings T and P, The parameterized pattern matching problem task is find positions of substrings of T that can be transformed into P by applying a one-to-one function that renames symbols in Π. The parameterized pattern matching is motivated by applying to software maintenance [8, 9, 11], plagiarism detection [35], analysis of gene structure [63], and so on. Some indexing structures that support the parameterized pattern matching are proposed, such as parameterized suffix trees [9, 11, 62], structural suffix trees [63], and parameterized suffix arrays [31, 41].

In this chapter, we propose a new indexing structure called *parameterized position heap* for the parameterized pattern matching. The parameterized position heap of a parameterized string T is a sequence hash tree for the ordered set of prev-encoded [9] suffixes of T. We give an online construction algorithm of a parameterized position heap based on Kucherov's algorithm [54] that runs in $O(n \log (|\Sigma| + |\Pi|))$ time and an algorithm that runs in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time to find the occurrences of a pattern in the text, where n is the length of the text, m is the length of the pattern, and occ is the number of occurrences of the pattern in the text.

5.1 Notation on parameterized string and parameterized pattern matching problem

Let Σ and Π be two disjoint sets of symbols. Σ is a set of *constant* symbols of size σ called *constant* alphabet and Π is a set of *parameter* symbols of size π called *parameter* alphabet. An element of Σ^* is called a *string*, and an element of $(\Sigma \cup \Pi)^*$ is called a *parameterized string*, or *p-string* for short [11]. Let $\mathcal{N}$ denote the set of all non-negative integers.

Definition 5.1 (Parameterized match [11]). *Two p-strings W_1 and W_2 of length n are a* parameterized match *or* p-match, *denoted by $W_1 \approx W_2$, if there exists a bijection f from the symbols of W_1 to the symbols of W_2, such that $f(W_1[i]) = W_2[i]$ for $1 \leq i \leq n$ and f is identity on the constant symbols.*

We can determine whether $W_1 \approx W_2$ or not by using an encoding called *prev-encoding* defined as follows.

Definition 5.2 (Prev-encoding [11]). *For a p-string W over $\Sigma \cup \Pi$, the* prev-encoding *for w, denoted by $\mathsf{prev}(W)$, is a string X of length $|W|$ over $\Sigma \cup \mathcal{N}$ defined by*

$$X[i] = \begin{cases} W[i] & \textit{if } W[i] \in \Sigma, \\ 0 & \textit{if } W[i] \in \Pi \wedge W[i] \neq W[j] \textit{ for } 1 \leq j < i, \\ i - \max\{j \mid W[j] = W[i] \wedge 1 \leq j < i\} & \textit{otherwise.} \end{cases}$$

For any p-strings W_1 and W_2, $W_1 \approx W_2$ if and only if $\mathsf{prev}(W_1) = \mathsf{prev}(W_2)$. For example, given $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ and $\Pi = \{u, v, x, y\}$, $W_1 = uvuv\mathsf{a}uuv\mathsf{b}$ and $W_2 = xyxy\mathsf{a}xxy\mathsf{b}$ are a p-match where $\mathsf{prev}(W_1) = \mathsf{prev}(W_2) = 0022\mathsf{a}314\mathsf{b}$.

The parameterized pattern matching is a problem to find occurrences of a p-string pattern in a p-string text defined as follows.

Definition 5.3 (Parameterized pattern matching [11]). *Given two p-strings, text T and pattern P, find all positions i in T such that $T[i : i + |P| - 1] \approx P$.*

For example, let us consider a text $T = uv\mathsf{a}u\mathsf{b}u\mathsf{a}v\mathsf{b}v$ and a pattern $p = x\mathsf{a}y\mathsf{b}y$ over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ and $\Pi = \{u, v, x, y\}$. Because $p \approx t[2 : 6]$ and $p \approx t[6 : 10]$, we should output 2 and 6 in this example. Throughout this chapter, let T be a text of length n and P be a pattern of length m.

5.2 Parameterized Position Heap

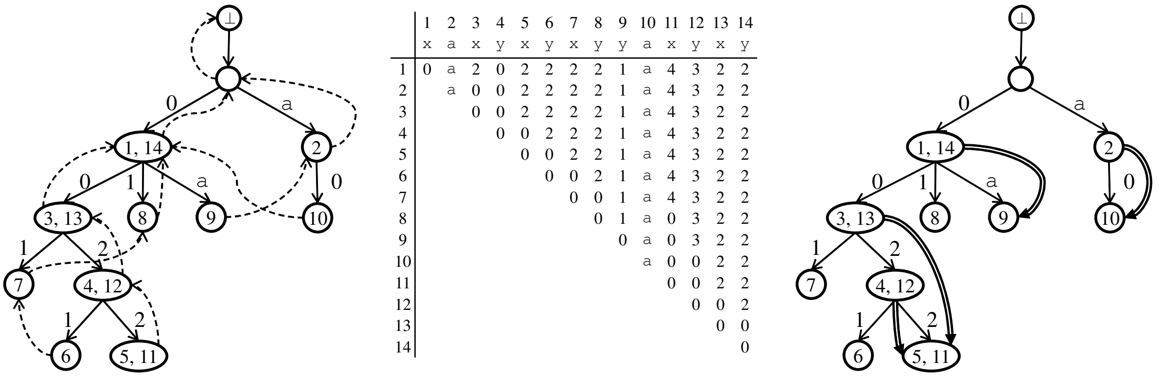
In this section, we propose a new indexing structure called *parameterized position heap*. It is based on the position heap proposed by Kucherov [54].

5.2.1 Definition and Property of Parameterized Position Heap

The parameterized position heap of a p-string T is a sequence hash tree [20] for the ordered set of prev-encoded suffixes of T in the descending order of length.

Definition 5.4 (Parameterized Position Heap). *Given a p-string $T \in (\Sigma \cup \Pi)^*$ of length n, let $\mathbf{S}_T = (\mathsf{prev}(T[1 :]), \mathsf{prev}(T[2 :]), \ldots, \mathsf{prev}(T[n :]))$ be the ordered set of all prev-encoded suffixes of the p-string T except ε in descending order of length. The* parameterized position heap $\mathsf{PPH}(T)$ *for T is $\mathsf{SHT}(\mathbf{S}_T)$.*

Fig. 5.1 (a) shows an example of a parameterized position heap. The parameterized position heap $\mathsf{PPH}(T)$ for a p-string T of length n consists of the root and nodes that correspond to $\mathsf{prev}(T[1 :]), \mathsf{prev}(T[2 :]), \ldots, \mathsf{prev}(T[n :])$, so $\mathsf{PPH}(T)$ has at most $n + 1$ nodes.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	x	a	x	y	x	y	x	y	y	a	x	y	x	y
1	0	a	2	0	2	2	2	2	1	a	4	3	2	2
2		a	0	0	2	2	2	2	1	a	4	3	2	2
3			0	0	2	2	2	2	1	a	4	3	2	2
4				0	0	2	2	2	1	a	4	3	2	2
5					0	0	2	2	1	a	4	3	2	2
6						0	0	2	1	a	4	3	2	2
7							0	0	1	a	4	3	2	2
8								0	1	a	0	3	2	2
9									0	a	0	3	2	2
10										a	0	0	2	2
11											0	0	2	2
12												0	0	2
13													0	0
14														0

(a) (b)

Figure 5.1: Let $\Sigma = \{\mathsf{a}\}$, $\Pi = \{x, y\}$ and $T = x\mathsf{a}xyxyxyy\mathsf{a}xyx$. (a) A parameterized position heap $\mathsf{PPH}(T)$. Broken arrows denote suffix pointers. (b) An augmented parameterized position heap $\mathsf{APPH}(T)$. Parameterized maximal-reach pointers for $\mathsf{pmrp}(i) \neq i$ are illustrated by double line arrows.

Each node in $\mathsf{PPH}(T)$ holds either one or two of beginning positions of corresponding suffixes. We can specify each node in $\mathsf{PPH}(T)$ by its primary position, its secondary position, or the string obtained by concatenating path labels from the root to the node.

Different from string, $\mathsf{prev}(T[i:]) = \mathsf{prev}(T)[i:]$ does not necessarily hold for some cases. For example, for $T = x\mathsf{a}xyxyxyy\mathsf{a}xyxy$, $\mathsf{prev}(T[3:]) = 0022221\mathsf{a}4322$ while $\mathsf{prev}(T)[3:] = 0222221\mathsf{a}4322$. Therefore, the construction and matching algorithms for the standard position heaps cannot be directly applied for the parameterized position heaps. However, we can use similar properties to construct parameterized position heaps efficiently.

Lemma 5.5. *For i and j, where $1 \leq i \leq j \leq n$, if $\mathsf{prev}(T[i:j])$ is represented in $\mathsf{PPH}(T)$, then a prev-encoded string for any substring of $T[i:j]$ is also represented in $\mathsf{PPH}(T)$.*

Proof: First we will show that prev-encoding of any prefix of $T[i:j]$ is represented in $\mathsf{PPH}(T)$. From the definition of prev-encoding, $\mathsf{prev}(T[i:j])[1:i-j] = \mathsf{prev}(T[i:j-1])$. In other words, $\mathsf{prev}(T[i:j-1])$ is a prefix of $\mathsf{prev}(T[i:j])$. From the definition of $\mathsf{PPH}(T)$, prefixes of $\mathsf{prev}(T[i:j])$ are represented in $\mathsf{PPH}(T)$. Therefore, $\mathsf{prev}(T[i:j-1])$ is represented in $\mathsf{PPH}(T)$. Similarly, $\mathsf{prev}(T[i:j-2])$, $\cdots$, $\mathsf{prev}(T[i:i])$ are represented in $\mathsf{PPH}(T)$.

Next, we will show that prev-encoding of any suffix of $T[i:j]$ is represented in $\mathsf{PPH}(T)$. From the above discussion, there are positions $b_0 < b_1 < \cdots < b_{j-i} = i$ in t such that $\mathsf{prev}(t[b_k : b_k + k]) = \mathsf{prev}(t[i : i+k])$. From the definition of parameterized position heap, $\mathsf{prev}(T[b_1+1 : b_1+1])$ is represented in $\mathsf{PPH}(T)$. Since $\mathsf{prev}(T[b_k+1 : b_k+k])$ is a prefix of $\mathsf{prev}(T[b_{k+1}+1 : b_{k+1}+k+1])$ for $0 < k < j-i$, if $\mathsf{prev}(T[b_k+1 : b_k+k])$ is represented in $\mathsf{PPH}(T)$ then $\mathsf{prev}(T[b_{k+1}+1 : b_{k+1}+k+1])$ is also represented in $\mathsf{PPH}(T)$ recursively. Therefore, $\mathsf{prev}(T[b_{j-i}+1 : b_{j-i}+j-i]) = \mathsf{prev}(T[i+1 : j])$ is represented in $\mathsf{PPH}(T)$. Similarly, $\mathsf{prev}(T[i+2 : j]), \cdots, \mathsf{prev}(T[j : j])$ are represented in $\mathsf{PPH}(T)$.

Since any prefix and suffix of $\mathsf{prev}(T[i:j])$ is represented in $\mathsf{PPH}(T)$, we can say that any substring of $\mathsf{prev}(T[i:j])$ is represented in $\mathsf{PPH}(T)$ by induction. ■

5.2.2 Online Construction Algorithm of Parameterized Position Heap

In this section, we propose an online algorithm that constructs parameterized position heaps. Our algorithm is based on Kucherov's algorithm [54], although it cannot be applied easily. The algorithm updates $\mathsf{PH}(T[1:k])$ to $\mathsf{PH}(T[1:k+1])$ when $T[k{+}1]$ is read, where $1 \le k \le n-1$. Updating of the position heap begins from a special node, called the *active node*. The position specified by the active node is called the *active position*. At first, we show that there exists a position similar to the active position in the parameterized position heap.

Lemma 5.6. *If $j < n$ is a secondary position of a double node in a parameterized position heap, then $j+1$ is also the secondary position of another node.*

Proof: Let i be the primary position and j be the secondary position of node v, where $i < j$. This means there is a position h such that $\mathsf{prev}(T[i:h]) = \mathsf{prev}(T[j:])$. By Lemma 5.5, there is a node that represents $\mathsf{prev}(T[i+1:h])$. Since $\mathsf{prev}(T[j+1:]) = \mathsf{prev}(T[i+1:h])$, then $j+1$ will be the secondary positions of node $\mathsf{prev}(T[i+1:h])$. ■

Lemma 5.6 means that there exists a position s which splits all positions in $T[1:n]$ into two intervals. Positions in $[1:s-1]$ and $[s:n]$ are called primary and secondary positions, respectively. We call the position s the *active position*.

Assume we have constructed $\mathsf{PPH}(T[1:k])$ and we want to construct $\mathsf{PPH}(T[1:k+1])$ from $\mathsf{PPH}(T[1:k])$. The primary positions $1,\ldots,s-1$ in $\mathsf{PPH}(T[1:k])$ become primary positions also in $\mathsf{PPH}(T[1:k+1])$, because $\mathsf{prev}(T[i:k]) = \mathsf{prev}(T[i:k+1])[1:k-1+1]$ holds for $1 \leq i \leq s-1$. Therefore, we do not need to update the primary positions.

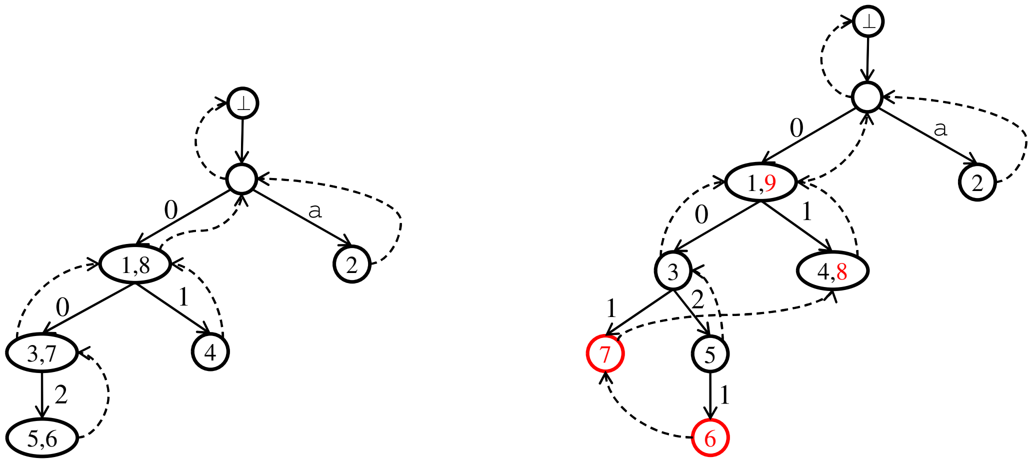
On the other hand, the secondary positions $s,\ldots,k$ require some modifications. When inserting a new symbol, two cases can occur. The first case is that $\mathsf{prev}(T[i:k+1])$ is not represented in $\mathsf{PPH}(T[1:k])$. In this case, a new node $\mathsf{prev}(T[i:k+1])$ is created as a child node of $\mathsf{prev}(T[i:k])$ and position i becomes the primary position of the new node. The second case is that $\mathsf{prev}(T[i:k+1])$ is represented in $\mathsf{PPH}(T[1:k])$. In this case, the secondary position i that is stored in $\mathsf{prev}(T[i:k])$ currently should be moved to the child node $\mathsf{prev}(T[i:k+1])$, and position i becomes the secondary position of this node.

From Lemma 5.5, if the node $\mathsf{prev}(t[i:k])$ has an edge to the node $\mathsf{prev}(T[i:k+1])$, $\mathsf{prev}(T[i+1:k])$ also has an edge to $\mathsf{prev}(T[i+1:k+1])$. Therefore, there exists r, with $1 \leq s \leq r \leq k$, that splits the interval $[s:k]$ into two subintervals $[s:r-1]$ and $[r:k]$, such that the node $\mathsf{prev}(T[i:k])$ does not have an edge to $\mathsf{prev}(T[i:k+1])$ for $s \leq i \leq r-1$, and does have such an edge for $r \leq i \leq k$.

The above analysis leads to the following lemma that specifies the modifications from $\mathsf{PPH}(T[1:k])$ to $\mathsf{PPH}(T[1:k+1])$.

Lemma 5.7. *Given $T \in (\Sigma \cup \Pi)^n$, consider $\mathsf{PPH}(T[1:k])$ for $k < n$. Let s be the active position, stored in the node $\mathsf{prev}(T[s:k])$. Let $r \geq s$ be the smallest position such that node $\mathsf{prev}(T[r:k])$ has an outgoing edge labeled with $\mathsf{prev}(T[r:k+1])[k-r+2]$. $\mathsf{PPH}(T[1:k+1])$ can be obtained by modifying $\mathsf{PPH}(T[1:k])$ in the following way:*

1. *For each node $\mathsf{prev}(t[i:k])$, $s \leq i < r$, create a new child $\mathsf{prev}(t[i:k+1])$ linked by an edge labeled $\mathsf{prev}(t[i:k+1])[k-i+2]$. Delete the secondary position i*



(a) (b)

Figure 5.2: An example of updating a parameterized position heap, from (a) $\mathsf{PPH}(x\mathtt{a}xyyxyx)$ to (b) $\mathsf{PPH}(x\mathtt{a}xyyxyxx)$. The updated positions are colored red. The secondary positions 6 and 7 in $\mathsf{PPH}(x\mathtt{a}xyyxyx)$ become primary positions in $\mathsf{PPH}(x\mathtt{a}xyyxyxx)$, while the secondary position 8 in $\mathsf{PPH}(x\mathtt{a}xyyxyx)$ becomes a secondary position of another node in $\mathsf{PPH}(x\mathtt{a}xyyxyxx)$. The active position is updated from 6 to 8.

from the node $\mathsf{prev}(t[i:k])$ *and assign it as the primary position of the new node* $\mathsf{prev}(t[i:k+1])$*.*

2. *For each node* $\mathsf{prev}(T[i:k])$*,* $r \leq i \leq k$*, move the secondary position* i *from the node* $\mathsf{prev}(T[i:k])$ *to the node* $\mathsf{prev}(T[i:k+1])$*.*

Moreover, r *will be the active position in* $\mathsf{PPH}(T[1:k+1])$*.*

Proof: Consider the first case that i is a secondary position in $\mathsf{PPH}(T[1:k])$ and $s \leq i < r$. From the definition of r, there is no node $\mathsf{prev}(T[i:k+1])$ in $\mathsf{PPH}(T[i:k])$. Therefore, i will become the primary position of the node $\mathsf{prev}(T[i:k+1])$ in $\mathsf{PPH}(T[1:k+1])$. We can update the position heap from $\mathsf{PPH}(T[1:k])$ to $\mathsf{PPH}(T[1:k+1])$ by deleting i from secondary position of the node $\mathsf{prev}(T[i:k])$ and create a new node $\mathsf{prev}(T[i:k+1])$ and assign i to its primary position for the case $s \leq i < r$.

Next case, i is a secondary position in $\mathsf{PPH}(T[1:k])$ and $r \leq i \leq k$. In this case, there is a node $\mathsf{prev}(T[i:k+1])$ in $\mathsf{PPH}(T[i:k])$ and the node $\mathsf{prev}(T[i:k+1])$ is also represented in $\mathsf{PPH}(T[i:k+1])$. Therefore, i will become a secondary position of the node $\mathsf{prev}(T[i:k+1])$ in $\mathsf{PPH}(T[1:k+1])$. We can update the position heap from $\mathsf{PPH}(T[1:k])$ to $\mathsf{PPH}(T[1:k+1])$ by deleting i from secondary position of the node $\mathsf{prev}(T[i:k])$ and assign i as the secondary position of the node $\mathsf{prev}(T[i:k+1])$ for the case $r \leq i \leq k$.

Since position i for $1 \leq i < r$ are primary positions in $\mathsf{PPH}(T[1:k+1])$ and position i for $r \leq i \leq k+1$ are secondary positions in $\mathsf{PPH}(T[1:k+1])$, r will be the active position in $\mathsf{PPH}(T[1:k+1])$. ■

Fig. 5.2 shows an example of updating a parameterized position heap. The modifications specified by Lemma 5.7 need to be applied to all secondary positions. In order to perform these modifications efficiently, we use parameterized suffix pointers.

Definition 5.8 (Parameterized Suffix Pointer). *For each node* $\mathsf{prev}(T[i:j])$ *of* $\mathsf{PPH}(T)$, *the* parameterized suffix pointer *of* $\mathsf{prev}(T[i:j])$ *is defined by* $\mathsf{psp}(\mathsf{prev}(T[i:j])) = \mathsf{prev}(T[i+1:j])$.

By Lemma 5.5, whenever the node $\mathsf{prev}(T[i:j])$ exists, the node $\mathsf{prev}(T[i+1:j])$ exists too. This means that $\mathsf{psp}(\mathsf{prev}(T[i:j]))$ always exists. During the construction of the parameterized position heap, let $\perp$ be the auxiliary node that works as the parent of *root* and is connected to *root* with an edge labeled with all symbols $c \in \Sigma \cup \{0\}$. We define $\mathsf{psp}(root) = \perp$.

When s is the active position in $\mathsf{PPH}(T[1:k])$, we call $\mathsf{prev}(T[s:k])$ the *active node*. If no node holds a secondary position, *root* becomes the active node and the active position is set to $k+1$. The nodes for the secondary positions s, $s+1$, $\dots$, k can be visited by traversing with the suffix pointers from the active node. Thus, the algorithm only has to memorize the active position and the active node in order to visit any other secondary positions.

Updating $\mathsf{PPH}(T[1:k])$ to $\mathsf{PPH}(T[1:k+1])$ specified by Lemma 5.7 is processed as the following procedures. The algorithm traverses with the suffix pointers from the active node till the node that has the outgoing edge labeled with $\mathsf{prev}(T[i:k+1])[k-i+2]$ is found, which is $i = r$. For each traversed node, a new node is created and linked by an

Algorithm 6: Parameterized position heap online construction algorithm

```
Function ConstructPPH($T$)
    create $root$ and $\bot$ nodes;
    psp($root$) $\leftarrow \bot$;
    child($\bot, c$) $\leftarrow root$ for $c \in \Sigma \cup \{0\}$;
    $currentNode \leftarrow root$;
    $s \leftarrow 1$;
    for $i = 1$ to $n$ do
        $c \leftarrow$ normalize(prev($T$)[$i$], depth($currentNode$));
        $lastCreateNode \leftarrow$ undefined;
        while child($currentNode, c$) $=$ null do
            create $newnode$;
            prim($newnode$) $\leftarrow s$;
            child($currentNode, c$) $\leftarrow newnode$;
            if $lastCreateNode \neq$ undefined then
                psp($lastCreateNode$) $\leftarrow newnode$;
            $lastCreateNode \leftarrow newnode$;
            $currentNode \leftarrow$ psp($currentNode$);
            $c \leftarrow$ normalize(prev($T$)[$i$], depth($currentNode$));
            $s \leftarrow s + 1$;
        6 $currentNode \leftarrow$ child($currentNode, c$);
        if $lastCreateNode \neq$ undefined then
            psp($lastCreateNode$) $\leftarrow currentNode$;
    while $s \leq n$ do
        sec($currentNode$) $\leftarrow s$;
        $currentNode \leftarrow$ psp($currentNode$);
        $s \leftarrow s + 1$;
```

edge labeled with $\mathsf{prev}(T[i : k+1])[k-i+2]$ to each node. A suffix pointer to this new node is set from the previously created node. When the node that has the outgoing edge labeled with $\mathsf{prev}(T[i : k+1])[k-i+2]$ is traversed, the algorithm moves to the node that is led to by this edge, and a suffix pointer to this node is set from the last created node, then the algorithm assigns this node to be the active node.

A pseudocode of our proposed construction algorithm is given as Algorithm 6. $\mathsf{prim}(v)$ and $\mathsf{sec}(v)$ denotes primary and secondary positions of v, respectively. From the property of prev-encoding, $\mathsf{prev}(T[i+1 : k+1])[k-i+1] = \mathsf{prev}(T[i : k+1])[k-i+2]$ if

$\mathsf{prev}(T[i:k+1])[k-i+2] \in \Sigma$ or $\mathsf{prev}(T[i:k+1])[k-i+2] \leq k-i$ and $\mathsf{prev}(T[i+1:k])[k-i+1] = 0$ otherwise. Therefore, we use a function $\mathsf{normalize}(c, j)$ that returns c if $c \in \Sigma$ or $c \leq j$ and returns 0 otherwise.

The construction algorithm consists of n iterations. In the i-th iteration, the algorithm reads $T[i]$ and constructs $\mathsf{PPH}(T[1:i])$. In the i-th iteration, the traversal of the suffix pointers as explained above is done. Since the depth of the current node decreases by traversing a suffix pointer, the number of the nodes that can be visited by traversal is $O(n)$. For each traversed node, all the operations such as creating a node, an edge and updating position can be done in $O(\log(|\Sigma| + |\Pi|))$ time. Therefore, the total time for the traversals is $O(n \log(|\Sigma| + |\Pi|))$.

From the above discussion, the following theorem is obtained.

Theorem 5.9. *Given $T \in (\Sigma \cup \Pi)^n$, Algorithm 6 constructs $\mathsf{PPH}(T)$ in $O(n \log(|\Sigma| + |\Pi|))$ time and space.*

5.3 Augmented Parameterized Position Heaps

We will describe *augmented parameterized position heaps*, the parameterized position heaps with an additional data structure called the *parameterized maximal-reach pointers* similar to the maximal-reach pointers for the position heap [54]. The augmented parameterized position heap gives an efficient algorithm for parameterized pattern matching.

Definition 5.10 (Parameterized Maximal-Reach Pointer)**.** *For a position i on T, the* parameterized maximal-reach pointer $\mathsf{pmrp}(i)$ *of position i is a pointer from node i to the deepest node whose path label is a prefix of* $\mathsf{prev}(T[i:])$.

Obviously, if i is a secondary position, then $\mathsf{pmrp}(i)$ is node i itself. We assume that the parameterized maximal-reach pointer for a double node applies to the primary position of this node. Fig. 5.1 (b) shows an example of an augmented parameterized position heap. Given a prev-encoded p-string $\mathsf{prev}(W)$ represented in an augmented parameterized position heap $\mathsf{APPH}(T)$ and a position $1 \leq i \leq n$, we can determine whether $\mathsf{prev}(W)$ is a prefix of $\mathsf{prev}(T[i:])$ or not in $O(1)$ time by checking whether $\mathsf{pmrp}(i)$ is a descendant of $\mathsf{prev}(W)$ or not. It can be done in $O(1)$ time by appropriately preprocessing $\mathsf{APPH}(T)$ [25].

Parameterized maximal-reach pointers can be computed by using parameterized suffix pointers, similarly to [54]. Algorithm 7 computes parameterized maximal-reach pointers. $\mathsf{pmrp}(i)$ is computed iteratively for $i = 1, 2, \cdots, n$. Assume that we have computed $\mathsf{pmrp}(i)$ for some i. Let $\mathsf{pmrp}(i) = \mathsf{prev}(T[i : l])$. Obviously, $\mathsf{prev}(T[i+1 : l])$ is a prefix of the string represented by $\mathsf{pmrp}(i+1)$. Thus, in order to compute $\mathsf{pmrp}(i+1)$, we should extend the prefix $\mathsf{prev}(T[i+1 : l]) = \mathsf{psp}(\mathsf{prev}(T[i : l]))$ in $\mathsf{PPH}(T)$ until we find l' such that node $\mathsf{prev}(T[i+1 : l'])$ does not have an outgoing edge labeled with $\mathsf{prev}(T[i+1 :])[l' - i + 1]$ and set $\mathsf{pmrp}(i+1) = \mathsf{prev}(T[i+1 : l'])$. In this time, we need re-compute $\mathsf{prev}(T[i+1 :])$ by replacing $\mathsf{prev}(T[i+1 :])[j]$ with 0 if $\mathsf{prev}(T[i+1 :])[j] \geq j$. The total number of extending $\mathsf{prev}(T[i+1 : l])$ in the algorithm is at most n because both i and l always increase in each iteration. In each iteration, operations such as traversing a child node can be done in $O(\log(|\Sigma| + |\Pi|))$ time. Therefore, we can get the following theorem.

Theorem 5.11. *Parameterized maximal-reach pointers for* $\mathsf{PPH}(T)$ *can be computed in* $O(n \log(|\Sigma| + |\Pi|))$ *time.*

5.3.1 Parameterized Pattern Matching with Augmented Parameterized Position Heaps

Ehrenfeucht et al. [32] and Kucherov [54] split a pattern P into segments $Q_1, Q_2, \cdots, Q_k$, then compute occurrences of $Q_1 Q_2 \cdots Q_j$ iteratively for $j = 1, \cdots, k$. The correctness depends on a simple fact that for strings $X = T[i : i + |x| - 1]$ and $Y = T[i + |x| : i + |x| + |y| - 1]$ imply $XY = T[i : i + |xy| - 1]$. However, when X, Y, and T are p-strings, $\mathsf{prev}(X) = \mathsf{prev}(T[i : i + |X| - 1])$ and $\mathsf{prev}(Y) = \mathsf{prev}(T[i + |X| : i + |X| + |Y| - 1])$ do not necessarily imply $\mathsf{prev}(XY) = \mathsf{prev}(T[i : i + |XY| - 1])$. Therefore, we need to modify

Algorithm 7: Augmented parameterized position heap construction algorithm

```
Function ConstructAPPH($T$)
    ConstructPPH($T$);
    let $t[n+1] \leftarrow \#$ where $\#$ is a symbol that does not appear in $t$ elsewhere;
    $currentNode \leftarrow root$;
    $l \leftarrow 1$;
    for $i = 1$ to $n$ do
        $c \leftarrow$ normalize(prev($T$)[$l$], $l - i$);
        while child($currentNode, c$) $\neq$ null do
            $currentNode \leftarrow$ child($currentNode, c$);
            $l \leftarrow l + 1$;
            $c \leftarrow$ normalize(prev($T$)[$l$], $l - i$);
        pmrp($i$) $\leftarrow currentNode$;
        $currentNode \leftarrow$ psp($currentNode$);
```

the matching algorithm for parameterized strings.

Let X, Y and W be p-strings such that $|W| = |XY|$, $\mathsf{prev}(X) = \mathsf{prev}(W[:|X|])$ and $\mathsf{prev}(Y) = \mathsf{prev}(W[|X|+1:])$. Let us consider the case that $\mathsf{prev}(XY) \neq \mathsf{prev}(W)$. From $\mathsf{prev}(X) = \mathsf{prev}(W[:|X|])$ and $\mathsf{prev}(Y) = \mathsf{prev}(W[|X|+1:])$, X and Y have the same structure of $W[:|X|]$ and $W[|X|+1:]$, respectively. However, the parameter symbols that are prev-encoded into 0 in $\mathsf{prev}(Y)$ and $\mathsf{prev}(W[|x|+1:])$, might be encoded differently in XY and $\mathsf{prev}(W)$, respectively. Therefore, we need to check whether $\mathsf{prev}(XY)[|X|+i] = \mathsf{prev}(W)[|X|+i]$ if $\mathsf{prev}(Y)[i] = 0$. Given $\mathsf{prev}(XY)$ and the set of positions of 0 in $\mathsf{prev}(Y)$, $Z = \{i \mid 1 \leq i \leq |Y|$ such that $\mathsf{prev}(Y)[i] = 0\}$. We need to verify whether $\mathsf{prev}(XY)[|X|+i] = \mathsf{prev}(W)[|X|+i]$ or not for $i \in Z$. Since the size of Z is at most $|\Pi|$, this computation can be done in $O(|\Pi|)$ time.

A pseudocode of proposed matching algorithm for the parameterized pattern matching problem is shown in Algorithm 8. $\mathsf{Des}_{\mathsf{APPH}(T)}(u)$ denotes the set of all descendants of node u in $\mathsf{APPH}(T)$ including node u itself. The occurrences of P in T have the following properties on $\mathsf{APPH}(T)$.

Lemma 5.12. *If* $\mathsf{prev}(P)$ *is represented in* $\mathsf{APPH}(T)$ *as a node* u *then* P *occurs at position* i *iff* $\mathsf{pmrp}(i)$ *is* u *or its descendant.*

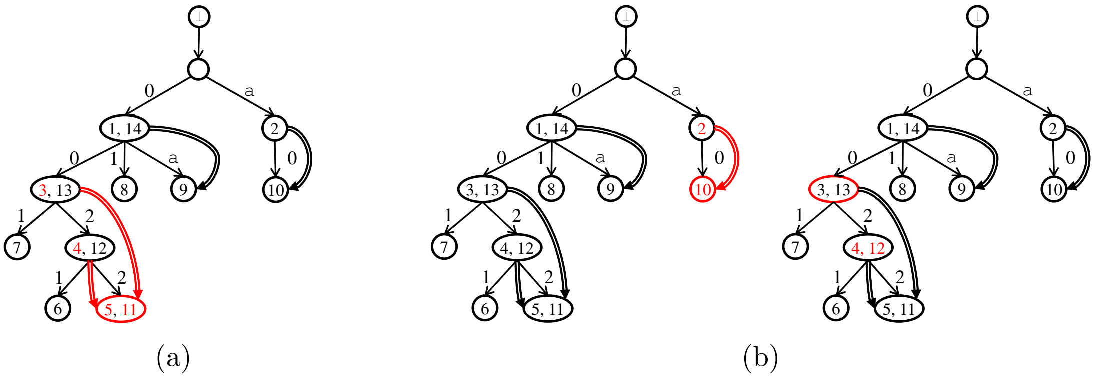


Figure 5.3: Examples of finding occurrence positions of a pattern using an augmented parameterized position heap $\mathsf{PPH}(x\mathtt{a}xyxyxyy\mathtt{a}xyxy)$. (a) Finding $xyxy$ ($\mathsf{prev}(xyxy) = 0022$). (b) Finding $\mathtt{a}xyx$ ($\mathsf{prev}(\mathtt{a}xyx) = \mathtt{a}002$).

Proof: Let u be the node that represents $\mathsf{prev}(P)$. Assume P occurs at position i in T and P is represented in $\mathsf{APPH}(T)$ as $\mathsf{prev}(T[i:k])$. Since either $\mathsf{prev}(T[i:k])$ is a prefix of $\mathsf{prev}(P)$ or $\mathsf{prev}(P)$ is a prefix of $\mathsf{prev}(T[i:k])$, then i is either an ancestor or descendant of u. For both cases $\mathsf{pmrp}(i)$ is a descendant of u, because P occurs at position i.

Next let i be a node such that $\mathsf{pmrp}(i)$ is a descendant of u and represents $\mathsf{prev}(T[i:k])$. In this case, $\mathsf{prev}(P)$ is a prefix of $\mathsf{prev}(T[i:k])$. Therefore P occurs at i. ■

Lemma 5.13. *Assume* $\mathsf{prev}(P)$ *is not represented in* $\mathsf{APPH}(T)$. *We can split* P *into* $Q_1, Q_2, \cdots, Q_k$ *such that* Q_j *is the longest prefix of* $\mathsf{prev}(P[|Q_1 \cdots Q_{j-1}| + 1 :])$ *that is represented in* $\mathsf{APPH}(T)$. *If* P *occurs at position* i *in* T, *then* $\mathsf{pmrp}(i + |Q_1 \cdots Q_{j-1}|)$ *is the node* $\mathsf{prev}(Q_j)$ *for* $1 \le j < k$ *and* $\mathsf{pmrp}(i + |Q_1 \cdots Q_{k-1}|)$ *is the node* $\mathsf{prev}(Q_k)$ *or its descendant.*

Proof: Assume that $P = Q_1 Q_2 \cdots Q_k$ occurs at position i in T. Since $\mathsf{prev}(Q_1)$ is a prefix of $\mathsf{prev}(p)$, then $\mathsf{pmrp}(i)$ is the node that represents $\mathsf{prev}(Q_1)$ or its descendant. However, if $\mathsf{pmrp}(i)$ is a descendant of node $\mathsf{prev}(Q_1)$, then we can extend Q_1 which contradicts with the definition of Q_1. Therefore, $\mathsf{pmrp}(i)$ is the node that represents $\mathsf{prev}(Q_1)$.

Similarly for $1 < j < k$, $\mathsf{prev}(Q_j)$ is a prefix of $\mathsf{prev}(p[|Q_1 \cdots Q_{j-1}| + 1 :])$ and occurs at position $i + |Q_1 \cdots Q_{j-1}|$ in T. Therefore, $\mathsf{pmrp}(i + |Q_1 \cdots Q_{j-1}|)$ is the node that represents $\mathsf{prev}(Q_j)$. Last, since Q_k is a suffix of P, then $\mathsf{pmrp}(i + |Q_1 \cdots Q_{j-1}|)$ can be the node $\mathsf{prev}(Q_k)$ or its descendant. ■

Algorithm 8 utilizes Lemmas 5.12 and 5.13 to find occurrences of P in T by using $\mathsf{APPH}(T)$. First, if $\mathsf{prev}(P)$ is represented in $\mathsf{APPH}(T)$ then the algorithm will output all positions i such that $\mathsf{pmrp}(i)$ is a node $\mathsf{prev}(P)$ or its descendant. Otherwise, it will split P into $Q_1 Q_2 \cdots Q_k$ and find their occurrences as described in Lemma 5.13. The algorithm also checks whether $\mathsf{prev}(Q_1 \cdots Q_j)$ occurs in T or not in each iteration as described the above.

Examples of parameterized pattern matching by using an augmented position heap are given in Fig. 5.3. Let $T = x\mathsf{a}xyxyxyy\mathsf{a}xyxy$ be the text. In Fig. 5.3 (a) we want to find the occurrence positions of a pattern $P_1 = xyxy$ in T. In this case, since $\mathsf{prev}(P_1) = 0022$ is represented in $\mathsf{PPH}(T)$, The algorithm outputs all positions i such that $\mathsf{pmrp}(i)$ is the node 0022 or its descendants, namely 3, 4, 5, and 11. On the other hand, Fig. 5.3 (b) shows how to find the occurrence positions of a pattern $p_2 = \mathsf{a}xyx$ in T. In this case, $\mathsf{prev}(P_2) = \mathsf{a}002$ is not represented in $\mathsf{PPH}(T)$. Therefore, The algorithm finds the longest prefix of $\mathsf{prev}(P_2)$ that is represented in $\mathsf{PPH}(i)$, which is $\mathsf{prev}(P_2)[1:2] = \mathsf{a}0$. We can see that $\mathsf{pmrp}(()2) = \mathsf{pmrp}(()10) = \mathsf{a}0$, then we save positions 2 and 10 as candidates to *ans*. Next, The algorithm finds the node that represents the longest prefix of $\mathsf{prev}(P_2[3:]) = 00$ which is $\mathsf{prev}(P_2[3:]) = 00$ itself. Since both of $\mathsf{pmrp}(2+|P_2[1:2]|) = \mathsf{pmrp}(4)$ and $\mathsf{pmrp}(10+|P_2[1:2])| = \mathsf{pmrp}(12)$ are descendants of the node 00, $\mathsf{prev}(T[2:5][3]) = \mathsf{prev}(T[10:13][3]) = \mathsf{prev}(P_2)[[3]] = 0$, and $\mathsf{prev}(T[2:5][4]) = \mathsf{prev}(T[10:13][4]) = \mathsf{prev}(P_2)[4] = 2$, then the algorithm outputs 2 and 10.

The time complexity of the matching algorithm is as follow.

Theorem 5.14. *Algorithm 8 runs in $O(m \log(|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.*

Proof: It is easily seen that we can compute line 5 to 9 in $O(m \log(|\Sigma| + |\Pi|) + occ)$ time. Assume that p can be decomposed into $Q_1, Q_2, \cdots, Q_k$ such that Q_1 is the longest prefix of P and Q_i is the longest prefix of $\mathsf{prev}(P[|Q_1 \cdots Q_{j-1}| + 1:])$ represented in $\mathsf{APPH}(T)$. The **For** loop in line 18 is iterated $k-1$ times. In the j-th iteration of the loop of line 21, Q_{j+1}

is extended up to reach $|Q_{j+1}|$ length. This can be computed in $O(|Q_{j+1}|\log(|\Sigma| + |\Pi|))$ time. After $k-1$ iterations, the total number of extending of Q_{j+1} does not exceed m, because $\sum_{j=2}^{k} |Q_j| < m$. In the **For** loop of line 28, the algorithm checks elements of ans. In the j-th iteration, the size of ans is at most $|Q_j|$. Thus, after $k-1$ iterations, the total number of elements checked in line 28 does not exceed m for the same reason for that of line 21. In each checking in line 28, the number of checks for line 30 and 32 is at most $|Q_j|$. Therefore, it can be computed from line 28 to 35 in $O(m|\Pi|)$ time. ■

Algorithm 8: Parameterized pattern matching algorithm with APPH

```
Function MatchAPPH($P$)
    let $w$ be the longest prefix of $\mathsf{prev}(P)$ represented in $\mathsf{APPH}(T)$ and $u$ be the node represents $w$;
    if $|w| = m$ then
        $v \leftarrow root$;
        for $i = 1$ to $m$ do
            $v \leftarrow \mathsf{child}(v, \mathsf{prev}(p)[i])$;
            if $\mathsf{pmrp}(v) \in \mathsf{Des}_{\mathsf{APPH}(t)}(u)$ then
                add $\mathsf{prim}(v)$ to $ans$;
        add all primary and sedondary position of decendants of $u$ to $ans$;
    else
        $v \leftarrow root$;
        $i \leftarrow 1, j \leftarrow 1$;
        while $i \leq |w|$ do
            $v \leftarrow \mathsf{child}(v, \mathsf{prev}(P)[i])$;
            $i \leftarrow i + 1$;
            if $\mathsf{pmrp}(v) = u$ then
                add $\mathsf{prim}(v)$ to $ans$;
        while $i \neq m$ do
            $j \leftarrow i, v \leftarrow root$;
            $Z \leftarrow \emptyset$;
            while $i \neq m$ do
                $c \leftarrow \mathsf{normalize}(\mathsf{prev}(p)[i], i - j)$;
                if $\mathsf{child}(v, c) = \mathit{null}$ then break;
                if $c = 0$ then add $i$ to $Z$;
                $v \leftarrow \mathsf{child}(v, c)$;
                $i \leftarrow i + 1$;
            if $v = root$ then return $\emptyset$;
            foreach $i' \in ans$ do
                if $i = m$ then
                    if $\mathsf{pmrp}(i' + j - 1) \notin \mathsf{Des}_{\mathsf{APPH}(t)}(v)$ then remove $i'$ from $ans$;
                else
                    if $\mathsf{pmrp}(i' + j - 1) \neq v$ then remove $i'$ from $ans$;
                for $k = 1$ to $|Z|$ do
                    if $\mathsf{normalize}(\mathsf{prev}(T)[i' + Z[k] - 1], Z[k] - 1) \neq \mathsf{prev}(P)[Z[k]]$ then
                        remove $i'$ from $ans$;
    return $ans$;
```

Chapter 6

Order-preserving Pattern Matching Algorithm

Order-preserving pattern matching is a variant of pattern matching that considers the relative order of elements, rather than their real values. Order-preserving pattern is more applicable for numerical data, such as sensor data, financial data, or climate data. The difficulty of order-preserving pattern matching mainly comes from the fact that we cannot determine the isomorphism by comparing the symbols in the text and the pattern on each position independently; instead, we have to consider their respective relative orders in the pattern and in the text.

Kubica et al. [53] and Kim et al. [50] independently proposed the same solution for order-preserving pattern matching based on the KMP algorithm. Their KMP-based algorithm runs in $O(n + m \log m)$ time. Cho et al. [19] brought forward another algorithm based on the Horspool algorithm that uses q-grams, which was proved to be experimentally fast. Crochemore et al. [28] proposed useful data structures for order-preserving pattern matching called order-preserving incomplete suffix tree. On the other hand, Chhabra and Tarhio [18], Faro and Külekci [33] proposed filtration methods which are practically fast. Moreover, faster filtration algorithms using SIMD (Single Instruction Multiple Data) instructions were proposed by Cantone et al. [15], Chhabra et al. [17] and Ueki et al. [68]. They showed that SIMD instructions are effective in speeding up their algorithms.

In this paper, we propose a new algorithm for order-preserving pattern matching based on the duel-and-sweep technique. Our algorithm runs in $O(n + m \log m)$ time which is as fast as the KMP based algorithm. Moreover, we perform experiments to compare those algorithms, which show that our algorithm is faster than the KMP-based algorithm.

6.1 Notation in order-preserving pattern matching

Let Σ denotes an alphabet of integer symbols such that the comparison of any two symbols can be done in constant time. We say that two strings S and T of equal length n are *order-isomorphic*, written $S \approx T$, if

$$S[i] \leq S[j] \Longleftrightarrow T[i] \leq T[j] \quad \text{for all } 1 \leq i, j \leq n.$$

For instance, $(12, 35, 5) \approx (25, 30, 21) \not\approx (11, 13, 20)$.

In order to check the order-isomorphism of two strings, Kubica et al. [53] defined useful arrays[1] Lmax_S and Lmin_S by

$$\mathit{Lmax}_S[i] = j \ (j < i) \ \text{ if } \ S[j] = \max_{k<i}\{S[k] \mid S[k] \leq S[i]\}, \tag{6.1}$$

$$\mathit{Lmin}_S[i] = j \ (j < i) \ \text{ if } \ S[j] = \min_{k<i}\{S[k] \mid S[k] \geq S[i]\}. \tag{6.2}$$

We use the rightmost (largest) j if there exist more than one such j. If there is no such j then we define $\mathit{Lmin}_S[i] = 0$ and $\mathit{Lmax}_S[i] = 0$. From the definition, we can easily observe

[1] Similar arrays Prev_S and Next_S are introduced in [38].

the following properties. Unless $Lmax_S[i] = 0$ or $Lmin_S[i] = 0$,

$$S[Lmax_S[i]] = S[i] \iff S[i] = S[Lmin_S[i]], \tag{6.3}$$

$$S[Lmax_S[i]] < S[i] \iff S[i] < S[Lmin_S[i]]. \tag{6.4}$$

Lemma 6.1 ([53])**.** *For a string S, let $sort(S)$ be the time required to sort the elements of S. $Lmax_S$ and $Lmin_S$ can be computed in $O(sort(S) + |S|)$ time.*

Thus, $Lmax_S$ and $Lmin_S$ can be computed in $O(|S| \log |S|)$ time in general. Moreover, the computation can be done in $O(|S|)$ time under a natural assumption [53] that the symbols of S are elements of the set $\{1, \ldots, |S|^{O(1)}\}$. By using $Lmax_S$ and $Lmin_S$, the order-isomorphism of two strings can be decided as follows.

Lemma 6.2 ([19])**.** *For two strings S and T of length n, assume that $S[1:j] \approx T[1:j]$ for some $j < n$. Moreover assume that $Lmax_S[j+1] \neq 0$ and $Lmin_S[j+1] \neq 0$. Let $i_{max} = Lmax_S[j+1]$ and $i_{min} = Lmin_S[j+1]$. Then $S[1:j+1] \approx T[1:j+1]$ if and only if either of the following two conditions holds.*

$$S[i_{max}] = S[j+1] = S[i_{min}] \ \wedge \ T[i_{max}] = T[j+1] = T[i_{min}], \tag{6.5}$$

$$S[i_{max}] < S[j+1] < S[i_{min}] \ \wedge \ T[i_{max}] < T[j+1] < T[i_{min}]. \tag{6.6}$$

Corollary 6.3. *Suppose that $P[1:j-1] \approx Q[1:j-1]$ and $P[1:j] \not\approx Q[1:j]$ for two strings P and Q of length at least j. For $i_{max} = Lmax_P[j]$ and $i_{min} = Lmin_P[j]$, if $i_{max}, i_{min} \neq 0$, we have*

$$\begin{aligned} & P[j] = P[i_{max}] \ \wedge \ Q[j] \neq Q[i_{max}] \\ \vee \ & P[j] = P[i_{min}] \ \wedge \ Q[j] \neq Q[i_{min}] \\ \vee \ & P[j] > P[i_{max}] \ \wedge \ Q[j] \leq Q[i_{max}] \\ \vee \ & P[j] < P[i_{min}] \ \wedge \ Q[j] \geq Q[i_{min}]\,. \end{aligned}$$

Table 6.1: Z-array of a string $S = (18, 22, 12, 50, 10, 17)$. For instance, $Z_S[3] = 3$ because $S[1:3] = (18, 22, 12) \approx (12, 50, 10) = S[3:5]$ and $S[1:4] = (18, 22, 12, 50) \not\approx (12, 50, 10, 17) = S[3:6]$. $Lmax_S$ and $Lmin_S$ are also shown.

	1	2	3	4	5	6
S	18	22	12	50	10	17
Z_S	6	1	3	1	2	1
$Lmax_S$	0	1	0	2	0	3
$Lmin_S$	0	0	1	0	3	1

The order preserving-pattern matching problem is defined as follows.

Definition 6.4 (Order-preserving pattern matching [50, 53])**.** *Given two strings, a text T of size n and a pattern P of size m, find all occurrence positions i such that of $T[i : i + m - 1] \approx P$.*

Hasan et al. [38] proposed a modification to Z-function, which Gusfield [37] defined for ordinary pattern matching, to make it useful from the order-preserving point of view. For a string S, the *Z-array for order-preserving pattern matching* of S is defined by

$$Z_S[i] = \max_{1 \le j \le |S|-i+1} \{j \mid S[1:j] \approx S[i : i + j - 1]\} \quad \text{for each } 1 \le i \le |S|.$$

In other words, $Z_S[i]$ is the length of the longest substring of S that starts at position i and is order-isomorphic to some prefix of S. An example of Z-array is illustrated in Table 6.1.

Lemma 6.5. *([38]) For a string S, the Z-array Z_S can be computed in $O(|S|)$ time, assuming that $Lmax_S$ and $Lmin_S$ are already computed.*

Note that in their original work, Hasan et al. [38] assumed that each symbol in S is distinct. However, we can extend their algorithm by using Lemma 6.2 to verify order-isomorphism even when S contains duplicate symbols.

In this chapter, we fix a text T of length n and a pattern P of length m.

6.2 Duel-and-sweep algorithm for order-preserving matching

In this section, we will propose an algorithm for OPPM based on the "duel-and-sweep" paradigm [4, 70]. The duel-and-sweep paradigm screens all substrings of length m of the text, called *candidates*, in two stages, called the *dueling* and *sweeping* stages. Suppose when P is superimposed on itself with the offset $a < m$ and the two overlapped substrings of P are not order-isomorphic. Then it is impossible that two candidates with offset a are both order-isomorphic to P. The dueling stage lets each pair of candidates with such an offset a "duel" and eliminates one based on this observation. This test is quick but not perfect. This stage can remove many candidates, although there would still remain candidates which are actually not order-isomorphic to the pattern. On the other hand, it is guaranteed that if distinct candidates that survive the dueling stage overlap, their prefixes of certain length are order-isomorphic. The sweeping stage takes the advantage of this property when checking the order-isomorphism between surviving candidates and the pattern so that this stage can be done also quickly.

Prior to the dueling stage, the pattern is preprocessed to construct a *witness table* based on which the dueling stage decides which pair of overlapping candidates should duel and how they should duel.

6.2.1 Pattern preprocessing

For each offset $0 < a < m$, the original duel-and-sweep algorithm [70] saves a position i such that $P[i] \neq P[i + a]$. However, in order-preserving pattern matching, the order-isomorphism of two strings cannot be determined by comparing a symbol in one position. We need two positions as a *witness* to say that the two strings are not order-isomorphic. Therefore, for each offset $0 < a < m$, when the overlapped regions obtained by superimposing P on itself with offset a are not order-isomorphic, we use a pair $\langle i, j \rangle$ of locations

Table 6.2: Witness table WIT_P for a string $P = (18, 22, 12, 50, 10, 17)$. For instance, the witness pair $WIT_P[2]$ for offset 2 is $(\langle 2, 4\rangle, <)$, due to $P[2] = 22 < 50 = P[4]$ and $P[2 + 2] = 50 > 17 = P[4 + 2]$. On the other hand, $WIT_P[4] = (\langle 0, 0\rangle, =)$, since $P[1 : 2] \approx P[5 : 6]$.

	1	2	3	4	5	6
P	18	22	12	50	10	17
WIT_P	$(\langle 1,2\rangle, <)$	$(\langle 2,4\rangle, <)$	$(\langle 1,2\rangle, <)$	$(\langle 0,0\rangle, =)$	$(\langle 0,0\rangle, =)$	–

called *a witness pair for the offset a* if either of the following holds:

1. $P[i] = P[j]$ and $P[i + a] \neq P[j + a]$,
2. $P[i] > P[j]$ and $P[i + a] \leq P[j + a]$,
3. $P[i] < P[j]$ and $P[i + a] \geq P[j + a]$.

For a witness pair $\langle i, j\rangle$, the *witness sign* of the witness pair is the equality/inequality sign $*$ that $P[i] * p[j]$ holds, that is $=$, $<$, and $>$ for condition 1,2, and 3, respectively. We call a tuple of a witness pair and a witness sign as a *witness*.

Next, we describe how to construct a *witness table* for P. The witness table WIT_P of P is an array of length $m - 1$, such that $WIT_P[a] = (\langle i, j\rangle, *)$ for all possible offsets a $(0 < a < m)$, where $\langle i, j\rangle$ is a witness pair and $*$ is witness sign of the witness pair. In the case when there are multiple witness pairs for offset a, we take the pair $\langle i, j\rangle$ with the smallest value of j and some $i < j$. When the overlap regions are order-isomorphic for offset a, which implies that no witness pair exists for a, we express it as $WIT_P[a] = (\langle 0, 0\rangle, =)$. Table 6.2 shows an example of a witness table.

Lemma 6.6. *For a pattern P of length m, Algorithm 9 constructs WIT_P in $O(m)$ time assuming that Z_P is already computed.*

Proof: Clearly the algorithm runs in $O(m)$ time.

We show that for each $1 \leq a < m$, Algorithm 9 computes $WIT_P[a]$ correctly. Recall that $Z_P[a+1]$ is the length of the longest prefix of $P[a+1 : m]$ that is order-isomorphic to a prefix of P. Let $j = Z_P[a + 1] + 1$, for which we have $P[1 : j - 1] \approx P[1 + a : j - 1 + a]$.

Algorithm 9: Algorithm for constructing the witness table WIT_P

```
Function Witness($P$)
    compute the Z-array $Z_P$ for the pattern $P$;
    for $a = 1$ to $m - 1$ do
        $j \leftarrow Z_P[a + 1] + 1$;
        $i_{min} \leftarrow Lmin_P[j]$;
        $i_{max} \leftarrow Lmax_P[j]$;
        if $j = m - a + 1$ then
            $WIT_P[a] \leftarrow (\langle 0, 0\rangle, =)$;
        else if $i_{max} = 0$ then
            $WIT_P[a] \leftarrow (\langle i_{min}, j\rangle, >)$;
        else if $i_{min} = 0$ then
            $WIT_P[a] \leftarrow (\langle i_{max}, j\rangle, <)$;
        else if $P[i_{min}] = P[i_{max}] = P[j]$ then
            if $P[i_{min} + a] \neq P[j + a]$ then
                $WIT_P[a] \leftarrow (\langle i_{min}, j\rangle, =)$;
            else
                $WIT_P[a] \leftarrow (\langle i_{max}, j\rangle, =)$;
        else if $P[i_{min}] > P[j] \ \wedge \ P[i_{min} + a] \leq P[j + a]$ then
            $WIT_P[a] \leftarrow (\langle i_{min}, j\rangle, >)$;
        else
            $WIT_P[a] \leftarrow (\langle i_{max}, j\rangle, <)$;
```

Suppose that $j = m-a+1$. This means that $P[1 : j-1] \approx P[1+a : j-1+a] = P[1+a : m]$, i.e., there is no witness pair for the offset a. Indeed Algorithm 9 gets $WIT_P[a] = (\langle 0, 0\rangle, =)$ for this case.

Otherwise, we have $P[1 : j] \not\approx P[1 + a : j + a]$. Let $i_{max} = Lmax_P[j]$ and $i_{min} = Lmin_P[j]$. If $i_{max} = 0$, $P[j] < P[k]$ for all $k < j$. Note that $i_{min} \neq 0$ by $j \geq 2$. Since $P[1 : j-1] \approx P[1+a : j-1+a]$ and $P[1 : j] \not\approx P[1+a : j+a]$, there exists $1 \leq k < j$ such that $P[k + a] \leq P[j + a]$. By $P[i_{min}] \leq P[k]$ and $(P[i_{min}], P[k]) \approx (P[i_{min} + a], P[k + a])$, we have $P[i_{min} + a] \leq P[k + a] \leq P[j + a]$. Therefore, $(\langle i_{min}, j\rangle, >)$ is a witness for the offset a. The case where $i_{min} = 0$ can be discussed in the exactly symmetric way.

Let us assume $i_{min} \neq 0$ and $i_{max} \neq 0$. If $P[i_{min}] = P[i_{max}] = P[j]$, either $P[i_{min} + a] \neq P[j + a]$ or $P[i_{max} + a] \neq P[j + a]$ holds. For the first case $(\langle i_{min}, j\rangle, =)$ is a witness for a

and for the second case $(\langle i_{max}, j\rangle, =)$ is a witness for a. Otherwise, by Corollary 6.3, either $P[i_{min}] > P[j] \wedge P[i_{min} + a] \leq P[j + a]$ or $P[i_{max}] < P[j] \wedge P[i_{max} + a] \geq P[j + a]$ holds, in which case either $(\langle i_{min}, j\rangle, >)$ or $(\langle i_{max}, j\rangle, <)$ is a witness pair for a, respectively. ∎

6.2.2 Dueling stage

Let us denote the candidate that starts at the location x as $T_x = T[x : x + m - 1]$. In the dueling stage, we "duel" all pairs of overlapping candidates T_x and T_{x+a} such that $WIT_P[a] \neq (\langle 0, 0\rangle, =)$. Witness pairs are used in the following manner. Suppose that $WIT_P[a] = (\langle i, j\rangle, <)$, where $P[i] < P[j]$ and $P[i + a] \geq P[j + a]$, for example. Then, it holds that

- if $T[x + a + i - 1] \geq T[x + a + j - 1]$, then $T_{x+a} \not\approx P$,
- if $T[x + a + i - 1] < T[x + a + j - 1]$, then $T_x \not\approx P$.

Based on this observation, we can safely eliminate either candidate T_x or T_{x+a} without looking into other locations. We can perform this process similarly for other equality/inequality cases. This process is called *dueling*. The procedure for all cases of the dueling is described in Algorithm 10.

On the other hand, if T_x and T_{x+a} do not overlap or the offset a has no witness pair, i.e. $P[1 : m - a] \approx P[a + 1 : m]$, no dueling is performed on them. We say that a position x *is consistent with* $x + a$ if either $0 < a < m$ and $WIT_P[a] = (\langle 0, 0\rangle, =)$ or $a \geq m$. Note that the consistency property is determined by a and P only, and x and T are irrelevant. The consistency property is transitive.

Lemma 6.7. *For any a, b and x such that $1 \leq a < a + b < m$ and $1 \leq x < m - a - b$, if x is consistent with $x + a$ and $x + a$ is consistent with $x + a + b$, then x is consistent with $x + a + b$.*

Proof: Since x is consistent with $x + a$, it follows that $P[1 : m-a] \approx P[a+1 : m]$, so that $P[b + 1 : m - a] \approx P[(a + b) + 1 : m]$. Moreover, since $x + a$ is consistent with $x + a + b$,

Algorithm 10: Dueling

```
Function Dueling($x, a$)
    $(\langle i, j\rangle, *) \leftarrow WIT_P[a]$;
    if $T[x + a + i - 1] * P[x + a + j - 1]$ then
        return $x + a$;
    else
        return $x$;
```

Algorithm 11: The dueling stage algorithm

```
Function DuelingStage($P, T$)
    create $stack$;
    for $y = 1$ to $n - m + 1$ do
        while $stack$ is not empty do
            pop $x$ from $stack$;
            if $y - x \geq m$ or $WIT_P[y - x] = (\langle 0, 0\rangle, =)$ then
                push $x$ and $y$ to $stack$;
                break;
            else
                $z \leftarrow$ Dueling($x, y - x$);
                if $z = x$ then
                    push $x$ to $stack$;
                    break;
        if $stack$ is empty then
            push $y$ to $stack$;
```

it follows that $P[1 : m - b] \approx P[b + 1 : m]$, so that $P[1 : m - b - a] \approx P[b + 1 : m - a]$. Thus, $P[1 : m - (a + b)] \approx P[(a + b) + 1 : m]$, which implies that x is consistent with $x + a + b$. ■

The whole process of the dueling stage is shown in Algorithm 11, which follows Amir et al. [5] for ordinary pattern matching. This stage eliminates candidates until all surviving candidates are pairwise consistent. The algorithm uses a stack to maintain candidates which are consistent with each other. A new candidate y will be pushed to the stack if the stack is empty. Otherwise y is checked by comparing it to the topmost element x of the stack. By Lemma 6.7, if x is consistent with y, all the other elements

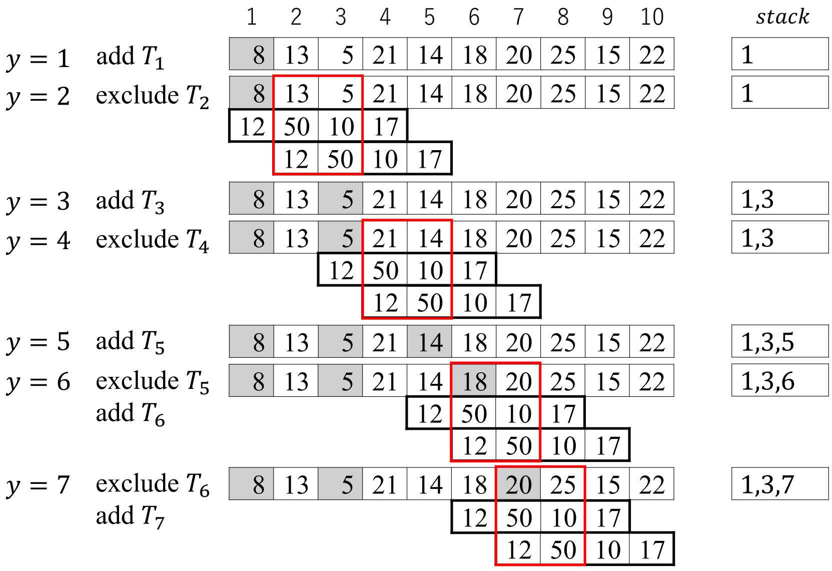


Figure 6.1: An example run of the dueling stage for $T = (8, 13, 5, 21, 14, 18, 20, 25, 15, 22)$, $P = (12, 50, 10, 17)$, and $WIT_P = ((\langle 1, 2\rangle, <), \langle 0, 0\rangle, \langle 0, 0\rangle)$. First, the position 1 is pushed to the stack. Next, T_2 duels with T_1 and then T_2 loses because $P[1] < P[2]$ and $T_2[1] > T_2[2]$. The next position 3 is pushed to the stack by $WIT_P[3-1] = (\langle 0, 0\rangle, =)$. Similarly, T_4 loses against T_3, and 5 is accepted to the stack. For $y = 6$, T_5 is removed and T_6 is added because $P[1] < P[2]$, $T_6[1] < T_6[2]$, and 3 is consistent with 6. Finally T_7 defeats T_6 and the contents of the stack become 1, 3, and 7.

in the stack are consistent with y, too. Thus we can push y to the stack. On the other hand, if x is not consistent with y, we should exclude one of the candidates by dueling them. If x wins the duel, we put x back to the stack, discard y, and get a new candidate. If y wins the duel, we exclude x and continue comparison of y with the top element of the stack unless the stack is empty. If the stack is empty, y will be pushed to the stack. Fig. 6.1 gives an example run of the dueling stage.

Lemma 6.8 ([5])**.** *The dueling stage can be done in $O(n)$ time by using WIT_P.*

6.2.3 Sweeping stage

The goal of the sweeping stage is to eliminate inconsistent candidates until all remaining candidates are order-isomorphic to the pattern P. Suppose that we need to check whether some surviving candidate T_x is order-isomorphic to P. It suffices to successively check

Algorithm 12: The sweeping stage algorithm

```
Function SweepingStage()
    while there are unchecked candidates to the right of $T_x$ do
        let $T_x$ be the leftmost unchecked candidate;
        if there are no candidates overlapping with $T_x$ then
            if $T_x \not\approx P$ then
                eliminate $T_x$;
        else
            let $T_{x+a}$ be the leftmost candidate that overlaps with $T_x$;
            if $T_x \approx P$ then
                start checking $T_{x+a}$ from the location $m - a + 1$;
            else
                let $j$ be the mismatch position;
                eliminate $T_x$;
                start checking $T_{x+a}$ from the location $j - a$;
```

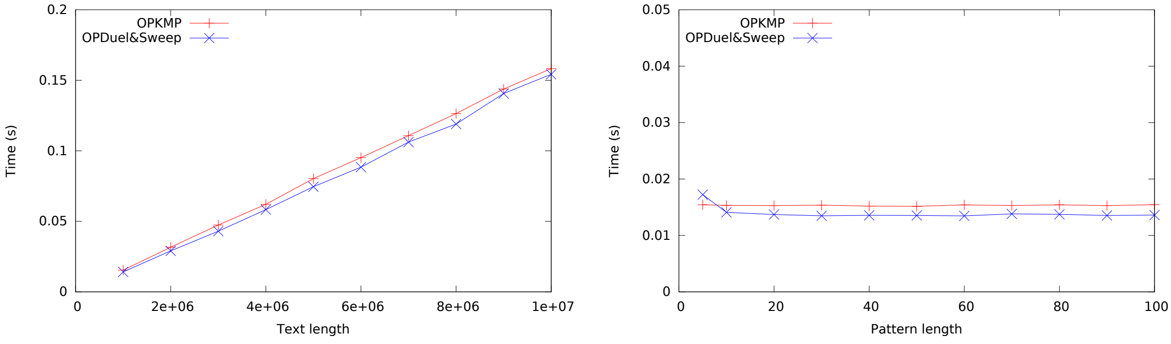
the conditions (6.5) and (6.6) in Lemma 6.2, starting from the leftmost location in T_x. If the conditions are satisfied for all locations in T_x, then $T_x \approx P$. Otherwise, $T_x \not\approx P$, and obtain a mismatch position j.

A naive implementation of sweeping requires $O(n^2)$ time. Algorithm 12 takes advantage of the fact that all the remaining candidates are pairwise consistent, we can reduce the time complexity to $O(n)$ time. Suppose there is a mismatch at position j when comparing P with T_x, that is, $T_x[1 : j-1] \approx P[1 : j-1]$ and $T_x[1 : j] \not\approx P[1 : j]$. If the next candidate is T_{x+a} with $a < j$, since $P[1 : j-a-1] \approx P[a+1 : j-1] \approx T_x[a+1 : j-1] = T_{x+a}[1 : j-a-1]$, we can start comparison of P and T_{x+a} from the position where the mismatch with T_x occurred. If $P \approx T_x$, the above discussion holds for $j = m+1$. Therefore, the total number of comparison is bounded by $O(n)$, by applying the same argument on the complexity of the KMP algorithm for exact matching.

Lemma 6.9. *The sweeping stage can be completed in $O(n)$ time.*

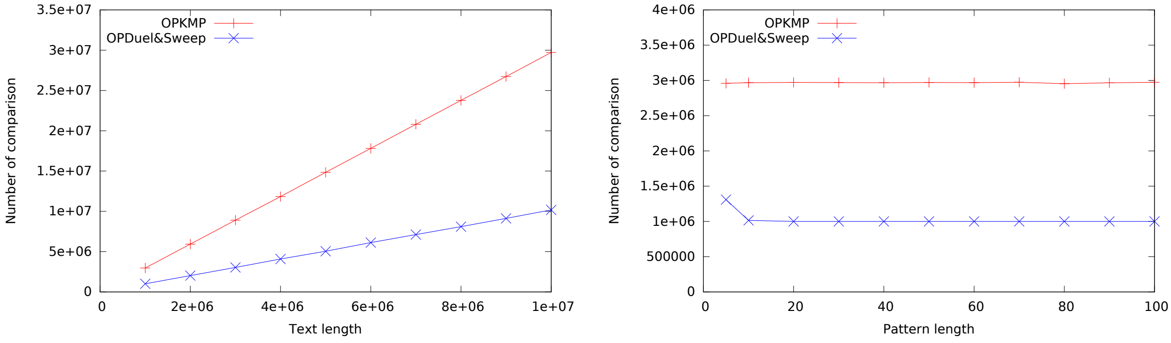
By Lemmas 6.6, 6.8, and 6.9, we summarize this section as follows.

Theorem 6.10. *Given a text T of length n and a pattern P of length m, The duel-and-*



(a) (b)

Figure 6.2: Running time of the algorithms with respect to (a) text length, and (b) pattern length.



(a) (b)

Figure 6.3: Number of comparisons in the algorithms with respect to (a) text length, and (b) pattern length.

sweep algorithm solves the OPPM Problem in $O(n+m\log m)$ time. Moreover, the running time is $O(n+m)$ under the natural assumption that the symbols of P can be sorted in $O(m)$ time.

6.3 Experiments

In order to compare the performance of proposed algorithm with the KMP-based algorithm [53, 50] on solving the OPPM problem, we performed two sets of experiments. In the first experiment set, the pattern size m is fixed at 10, while the text size n is changed from 100000 to 1000000. In the second experiment set, the text size n is fixed at 1000000 while the pattern size m is changed from 5 to 100. We measured the average of running time and the number of comparisons for 50 repetitions on each experiment. We used randomly generated texts and patterns with alphabet size $|\Sigma| = 1000$. Experiments are executed on a machine with Intel Xeon CPU E5-2609 8 cores 2.40 GHz, 256 GB memory, and Debian Wheezy operating system.

The results of our experiments are shown in Figs. 6.2 and 6.3. We can see that our algorithm is better than the KMP-based algorithm in running time and the number of comparisons when the pattern size and text size are large. However, our algorithm was slower when the pattern is very short, namely $m = 5$. The reason why the proposed algorithm makes fewer comparisons than the KMP-based algorithm may be explained as follows. The KMP-based algorithm relies on Lemma 6.2, which compares symbols at three positions (two inequalities) to check the order-isomorphism between a prefix of the pattern and a substring of the text when the prefix is extended by one. On the other hand, the dueling stage of our algorithm compares only two positions (one equality/inequality) determined by the witness table. By eliminating candidates in the dueling phase, the number of precise tests of order-isomorphism in the sweeping stage is reduced.

Chapter 7

Permuted Pattern Matching Algorithms

The *permuted pattern matching problem*, proposed by Katsura et al. [48, 49], is a generalization of the pattern matching problem, where we compare tuples of strings. Tuples of strings can model various types of real data such as multiple-sensor data, polyphonic music data, and multiple genomes. We call a tuple of strings of the same length a *multi-track string*. The permuted pattern matching problem is, given two multi-track strings $\mathbb{T} = (T_1, T_2, \ldots, T_N)$ and $\mathbb{P} = (P_1, P_2, \ldots, P_M)$ where $M = N$ and $|T_1| = \cdots = |T_N| = n \geq |P_1| = \cdots = |P_M| = m$, to find all positions i such that $\mathbb{P}$ is a permutation of $(T_1[i : i+m-1], \ldots, T_N[i : i+m-1])$. The problem is called *sub-permuted pattern matching* when $N \geq M$, where we need to find all positions i such that $\mathbb{P}$ is a sub-permutation of $(T_1[i : i+m-1], \ldots, T_N[i : i+m-1])$. The problem can be solved by constructing some data structure from the text such as a multi-track suffix tree [48] and a multi-track position heap [49], or by preprocessing the pattern such as AC automaton based algorithm [48].

In this chapter, we propose several algorithms that solve permuted pattern matching fast. The algorithms can be grouped into three groups and an algorithm that does not

Table 7.1: Comparison of the algorithms for permuted pattern matching. Multi-track AC-automaton can find occurrences of *multiple* patterns.

Algorithm	Preprocessing time	Matching time
AC-automaton based [48]	$O(mM \log \sigma)$	$O(nN \log \sigma)$
Multi-track KMP	$O(mM)$	$O(nN)$
MT AC-automaton	$O(dM \log \sigma)$	$O(nN \log \sigma)$
MT permuted matching automaton	$O(mM \log \sigma)$	$O(nN \log \sigma)$
MT Boyer-Moore	$O(m(M \log \sigma + \sigma))$	$O(nN(m + \log \sigma) + n(N + \sigma))$
MT Horspool	$O(m(M \log \sigma + \sigma))$	$O(nN(m + \log \sigma) + n(N + \sigma))$
MT duel-and-sweep	$O(mM)$	$O(nN)$

belong to any group. The first group is a group of algorithms that are based on the KMP algorithm [51]. This group includes *multi-track KMP*, *multi-track AC-automaton*, and *multi-track permuted matching automaton* algorithms. The second group is a group of algorithms that are based on the Boyer-Moore algorithm [14] and the Horspool algorithm [40]. This group includes *multi-track Boyer-Moore* and *multi-track Horspool* algorithms. The third group is a group of filtration algorithms. The last algorithm that does not belong to any group is the multi-track duel-and-sweep algorithm. This algorithm is based on the duel-and-sweep algorithm for pattern matching [4, 70]. Moreover, we conduct experiments and show that our algorithms perform permuted pattern matching faster than existing algorithms. The worst case running time of proposed algorithms and existing algorithms are summarized in Table 7.1, where d is the total length of the patterns and σ is the size of the alphabet.

7.1 Notation and definition on multi-track strings

A *multi-track symbol* $\mathbb{C} = (c_1, c_2, \ldots, c_N)$ is an N-tuple of symbols $c_i \in \Sigma$. A *multi-track string* (or *multi-track* for short) $\mathbb{W} = (W_1, W_2, \ldots, W_N)$ is an N-tuple of strings $W_i \in \Sigma^*$ where $|W_1| = |W_2| = \cdots = |W_N|$, and each W_i is called the *i-th track* of $\mathbb{W}$. The length n of strings in $\mathbb{W}$ is called the *length* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{len}$, and the number N

of tracks in $\mathbb{W}$ is called the *track count* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{num}$. For a multi track $\mathbb{W}$, $\mathbb{W}[i] = (W_1[i], W_2[i], \ldots, W_N[i])$ denotes the i-th multi-track symbol, $\mathbb{W}[i][j]$ denotes $W_j[i]$, and $\mathbb{W}[i:j] = (W_1[i:j], W_2[i:j], \ldots, W_N[i:j])$ denotes the substring of $\mathbb{W}$ that begins at position i and ends at position j for $1 \leq i \leq j \leq |\mathbb{W}|_{len}$. Similarly to the notation for strings, $\mathbb{W}[:i]$ and $\mathbb{W}[i:]$ mean $\mathbb{W}[1:i]$ and $\mathbb{W}[i:|\mathbb{W}|_{len}]$, which are called a *prefix* and a *suffix* of $\mathbb{W}$, respectively.

Let $\mathbf{r} = (r_1, r_2, \ldots, r_N)$ be a permutation of $(1, 2, \ldots, N)$. For a multi-track $\mathbb{W} = (W_1, W_2, \ldots, W_N)$, $\mathbb{W}\langle\mathbf{r}\rangle = \mathbb{W}\langle r_1, r_2, \ldots, r_N\rangle = (W_{r_1}, \ldots, W_{r_N})$ is called a *permuted multi-track* of $\mathbb{W}$. The *sorted index* $\mathsf{SI}(\mathbb{W})$ of a multi-track $\mathbb{W}$ is a permutation $(r_1, \ldots, r_N)$ such that $W_{r_i} \preceq W_{r_{i+1}}$ for any $1 \leq i < N$, where we assume $r_i < r_{i+1}$ in the case $W_{r_i} = W_{r_{i+1}}$. The sorted multi-track $\mathsf{sort}(\mathbb{W})$ is defined as $\mathbb{W}\langle SI(\mathbb{W})\rangle$. The *reverse* of a multi-track $\mathbb{W} = (W_1, \ldots, W_N)$ is $\mathbb{W}^R = (W_1^R, \ldots, W_N^R)$. The sorted index of the reverse multi-track, denoted by $\mathsf{RI}(\mathbb{W})$, is a permutation $(r_1, \ldots, r_N)$ such that $w_{r_i}^R \preceq w_{r_{i+1}}^R$ for any $1 \leq i < N$. Note that $\mathsf{SI}(\mathbb{W}[i:])$ and $\mathsf{RI}(\mathbb{W}[:i])$ for $1 \leq i \leq n$ can be computed in $O(nN)$ time *offline* by using suffix arrays [52, 61], and $\mathsf{RI}(\mathbb{W}[:i])$ for $1 \leq i \leq n$ can be computed in $O(n(N+\sigma))$ time *online* by using radix sort.

For two multi-tracks $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \bowtie \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}\langle\mathbf{r}\rangle$ for some permutation $\mathbf{r}$. Moreover, for two multi-tracks $\mathbb{X} = (x_1, x_2, \ldots, x_M)$ and $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$ for $M \leq N$, we say $\mathbb{X}$ *sub-permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \overset{\bowtie}{\sqsubseteq} \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}\langle\mathbf{r}\rangle$ for some partial permutation $\mathbf{r}$.

Lemma 7.1 ([48])**.** *For two multi-tracks* $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ *and* $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X} \bowtie \mathbb{Y}$ *if and only if* $\mathsf{sort}(\mathbb{X}) = \mathsf{sort}(\mathbb{Y})$.

Lemma 7.2. *For two multi-tracks* $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ *and* $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X} \not\bowtie \mathbb{Y}$ *if there is a position* i *such that* $\mathbb{X}[i] \not\bowtie \mathbb{Y}[i]$.

Lemma 7.3. *For two multi-tracks* $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ *and* $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X} \not\bowtie \mathbb{Y}$ *if and only if there are two positions* i *and* j *such that* $\mathbb{X}[i]\mathbb{X}[j] \not\bowtie \mathbb{Y}[i]\mathbb{Y}[j]$.

Proof: ($\Longrightarrow$) From Lemma 7.1, if $\mathbb{X} \not\bowtie \mathbb{Y}$, then $\mathsf{sort}(\mathbb{X}) \neq \mathsf{sort}(\mathbb{Y})$. Thus there are two position i and j such that $\mathsf{sort}(\mathbb{X})[i]\mathsf{sort}(\mathbb{X})[i][j] \neq \mathsf{sort}(\mathbb{Y})[i]\mathsf{sort}(\mathbb{Y})[j]$. Therefore, there are two position i and j such that $\mathbb{X}[i]\mathbb{X}[j] \not\bowtie \mathbb{Y}[i]\mathbb{Y}[j]$, by Lemma 7.1.

($\Longleftarrow$) Obviously, if $\mathbb{X} \bowtie \mathbb{Y}$ then $\mathbb{X}[i]\mathbb{X}[j] \bowtie \mathbb{Y}[i]\mathbb{Y}[j]$ for any i and j. Thus if there are positions i and j such that $\mathbb{X}[i]\mathbb{X}[j] \not\bowtie \mathbb{Y}[i]\mathbb{Y}[j]$ then $\mathbb{X} \not\bowtie \mathbb{Y}$. ∎

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

Definition 7.4 (Permuted pattern matching[48]). *Given a multi-track text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$, compute all positions i that satisfy $\mathbb{P} \bowtie \mathbb{T}[i : i + m - 1]$.*

Definition 7.5 (Sub-permuted pattern matching[48]). *Given a multi-track text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$, compute all positions i that satisfy $\mathbb{P} \overset{\bowtie}{\sqsubseteq} \mathbb{T}[i : i + m - 1]$.*

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

7.2 KMP based permuted pattern matching algorithms

In this section, we propose algorithms for permuted pattern matching based on KMP algorithm. The first algorithm is called *multi-track KMP algorithm* (shortly MTKMP) that uses the border array of a multi-track pattern.

Algorithm 13: Multi-track border array contruction algorithm

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

7.2.1 Multi-track KMP algorithm

The *Multi-track KMP algorithm* is an extension of the KMP algorithm for multi-track. MTKMP uses *multi-track border array* to compute the shift amount when the algorithm found a mismatch. First we define a border of am ulti-track.

Definition 7.6 (Border of multi-track)**.** *Given a multi-track* $\mathbb{P}$*, a border of* $\mathbb{P}$ *is any multi-track that permuted matches with a prefix and a suffix of* $\mathbb{P}$*.*

A *proper border* of $\mathbb{P}$ is any border of $\mathbb{P}$ that is not permuted match with $\mathbb{P}$. By using the definition of border and proper border, we define the border array of a multi-track.

Definition 7.7 (Multi-track border array)**.** *Given a multi-track* $\mathbb{P}$ *of* $|\mathbb{P}|_{len} = m$ *and* $|\mathbb{P}|_{num} = M$*, the border array* $Border_{\mathbb{P}}$ *of* $\mathbb{P}$ *is an array of length* m*, where the* i*-th element of* $Border_{\mathbb{P}}$ *is the length the longest proper border of* $\mathbb{P}[1:i]$*. Formally,*

$$Border_{\mathbb{P}}[i] = \max\{j|\mathbb{P}[1:j] \bowtie \mathbb{P}[i-j+1:i], 0 \leq j < i\} \tag{7.1}$$

For algorithm purpose, we define $Border_{\mathbb{P}}[0] = -1$. Algorithm 13 constructs the multi-track border array of a multi-track $\mathbb{P}$.

Algorithm 14: MTKMP pattern matching algorithm

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

Lemma 7.8. *Given a multi-track $\mathbb{P}$, Algorithm 13 constructs $Border_{\mathbb{P}}$ in $O(mM)$ time.*

Proof: Suffix index $\mathsf{SI}(\mathbb{P}[i:])$ for $1 \le i \le m$ can be computed in $O(mM)$. Both **while** loops are called $O(m)$ time at most, since the $i - j \le i \le m + 1$ and the value of i always increases each time the outer loop is called, and the value of $i - j$ always increases each time the inner loop is called. Each comparison of $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[1:]) \rangle$ and $\mathbb{P}[i]\langle \mathsf{SI}(\mathbb{P}[i-j:]) \rangle$ consumes $O(M)$ time, hence Algorithm 13 runs in $O(mM)$ time. ■

Algorithm 14 shows MTKMP pattern matching algorithm. The MTKMP algorithm performs permuted pattern matching from left to right of the pattern and the text. Sorted indexes of the pattern $\mathsf{SI}(\mathbb{P}[i:])$ for $1 \le i \le m$ and the text $\mathsf{SI}(\mathbb{T}[i:])$ for $1 \le i \le n$ are used to perform permuted-matching between the pattern and a substring of the text. If symbols on text and pattern are mismatched, we shift the matching position of the pattern by using $Border_{\mathbb{P}}$. If the pattern matches a substring of the text, then MTKMP outputs the occurrence position at text and shifts the pattern according to $Border_{\mathbb{P}}$.

Theorem 7.9. *The MTKMP algorithm runs in $O(nN + mM)$ time for matching.*

Proof: Sorted indexes $\mathsf{SI}(\mathbb{P}[i:])$ for $1 \le i \le m$ and $\mathsf{SI}(\mathbb{T}[i:])$ for $1 \le i \le n$ can be computed in $O(mM)$ and $O(nN)$, respectively. From Lemma 7.8, constructMTBorderArray($\mathbb{P}$)

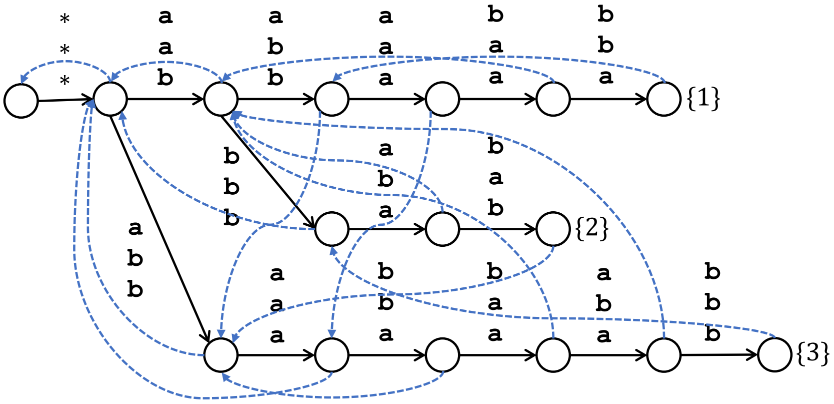


Figure 7.1: Multi-track AC-automaton of $D = \{\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3\}$, where $\mathbb{P}_1 = (\texttt{aaabb}, \texttt{abaab}, \texttt{bbaaa})$, $\mathbb{P}_2 = (\texttt{abab}, \texttt{abba}, \texttt{bbab})$, and $\mathbb{P}_3 = (\texttt{aabbab}, \texttt{bababb}, \texttt{baaaab})$. The asterisk '*' is a special symbol that matches with any symbols in Σ.

runs in $O(mM)$. Both **while** loops are called $O(n)$ times at most, since $i - j \leq i \leq n + 1$ and the value of i always increases each time the outer loop is called, and the value of $i - j$ always increases each time the inner loop is called. Each comparison of $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[1:]) \rangle$ and $\mathbb{T}[i]\langle \mathsf{SI}(\mathbb{T}[i-j:]) \rangle$ consumes $O(N)$ time. Hence Algorithm 14 runs in $O(nN + mM)$ time. ■

7.3 Multi-track AC-automaton

In this section, we introduce a data structure called a multi-track AC-automaton that can perform dictionary matching on multi-tracks, called *permuted dictionary matching*. Given a set $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ of multi-track patterns called a *dictionary* and a multi-track text $\mathbb{T}$, by preprocessing the patterns, the multi-track AC-automaton can find all occurrence positions of each pattern in the text. Let $d = \sum_{i=1}^{r} m_i$ be the total length of the patterns in D, where $m_i = |\mathbb{P}_i|_{len}$. The multi-track AC-automaton of D, denoted by $\mathsf{MTAC}(D)$, consists of three functions; *goto*, *failure*, and *output* functions. Fig. 7.1 shows an example of $\mathsf{MTAC}(D)$.

Unlike the original AC-automaton, the multi-track AC-automaton uses a multi-track symbol, instead of a single symbol to define the goto function. The states and goto function in $\mathsf{MTAC}(D)$ construct a trie of $\mathsf{sort}(\mathbb{P}_i)$ for all $\mathbb{P}_i \in D$. Each state in $\mathsf{MTAC}(D)$

Algorithm 15: Multi-track AC-automaton goto function construction algorithm

```
Function constructGotoFunction($D$)
    compute $\mathsf{SI}(\mathbb{P}_i[j :])$ for $1 \leq i \leq r$ and $1 \leq j \leq m_i$;
    create states $rootState$ and $\perp$;
    $\delta(\perp, \mathbb{C}) \leftarrow rootState$ for all multi-track symbols $\mathbb{C}$ of track count $M$;
    for $i \leftarrow 1$ to $r$ do
        $activeState \leftarrow rootState$;
        for $1 \leq j \leq m_i$ do
            if $\delta(activeState, \mathbb{P}_i[j]\langle \mathsf{SI}(\mathbb{P}_i[1 :])\rangle) \neq$ NULL then
                $activeState \leftarrow \delta(activeState, \mathbb{P}_i[j]\langle \mathsf{SI}(\mathbb{P}_i[1 :])\rangle)$;
            else
                create $newState$;
                $\delta(activeState, \mathbb{P}_i[j]\langle \mathsf{SI}(\mathbb{P}_i[1 :])\rangle) \leftarrow newState$;
                $label(newState) \leftarrow i$;
                $activeState \leftarrow newState$;
```

represents a prefix of $\mathsf{sort}(\mathbb{P}_i)$, thus each state can be identified with a multi-track $\mathbb{W}$, where $\mathbb{W}$ is the string obtained by concatenating the labels of the edges from the root to the state. Let δ be the goto function of a multi-track AC-automaton. We can define $\delta(\mathsf{sort}(\mathbb{P}_i)[: j]), \mathsf{sort}(\mathbb{P}_i)[j+1]) = \mathsf{sort}(\mathbb{P}_i)[: j+1]$ for $1 \leq i \leq r$ and $1 \leq j < m_i$. For convenience, we denote $\delta(s, \mathbb{C}\mathbb{W}) = \delta(\delta(s, \mathbb{C}), \mathbb{W})$ for any state s, multi-track symbol $\mathbb{C}$ and multi-track $\mathbb{W}$. For a state s and a multi-track symbol $\mathbb{C}$, $\delta(s, \mathbb{C})$ can be implemented by using a multi-track symbol trie of depth at most M nodes, thus $\delta(s, \mathbb{C})$ can be computed in $O(M \log \sigma)$ time. The function *goto* can be constructed by using Algorithm 15.

Lemma 7.10. *Algorithm 15 constructs the goto function of the multi-track AC-automaton of a dictionary $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ in $O(dM \log \sigma)$ time.*

Proof: The goto function $\delta(s, \mathbb{C})$ is computed $d = \sum_{i=1}^{r} m_i$ times and $\delta(s, \mathbb{C})$ can be computed in $O(M \log \sigma)$ time. ■

Next, the failure link of a state $\mathsf{sort}(\mathbb{P}_i)[: j]$ is defined as $\mathit{flink}(\mathsf{sort}(\mathbb{P}_i)[: j]) = \mathsf{sort}(\mathbb{P}_i[k : j])$, where $\mathbb{P}_i[k : j]$ is the longest proper suffix of $\mathbb{P}_i[: j]$ such that $\mathbb{P}_i[k : j]$ is permuted-matched with a prefix of any $\mathbb{P}_\ell \in D$. Algorithm 16 shows a construction algorithm for

Algorithm 16: Multi-track AC-automaton failure function construction algorithm

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

the failure function of a multi-track AC-automaton. In order to simplify the construction algorithm, we use a special state that reads any multi-track symbol to get to the root state.

Lemma 7.11. *Algorithm 16 constructs the failure function of the multi-track AC-automaton of a dictionary $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ in $O(dM \log \sigma)$ time.*

Proof: We can bound the running time of Algorithm 16 by counting the number of executions of $\delta(s, \mathbb{C})$. For each pattern $\mathbb{P}_i$, let $s_{i,j}$ be a state such that $s_{i,j} = \delta(\mathit{root}, \mathbb{P}_i[:j])$ for $1 \le j \le m_i$. Let $f_{i,j}$ be the number of executions of flink when finding $\mathsf{flink}(s_{i,j})$. The maximum value of $f_{i,j}$ is bounded by $\mathsf{depth}(\mathsf{flink}(s_{i,j-1})) + 1$. Because the depth of $\mathsf{flink}(s_{i,j})$ is at most $\mathsf{depth}(\mathsf{flink}(s_{i,j-1})) - f_{i,j} + 1$, we get $f_{i,j} \le \mathsf{depth}(\mathsf{flink}(s_{i,j-2})) - f_{i,j-1} + 2$ recursively. By solving this formula, we get $\sum_{j=1}^{m_i} f_{i,j} \le 2m_i$, and $\sum_{i=1}^{r} \sum_{j=1}^{m_i} f_{i,j} \le \sum_{i=1}^{r} 2m_i = 2d$. Moreover, each $\delta(s, \mathbb{C})$ is executed in $O(M \log \sigma)$ time. ■

Finally, the output function of the multi-track AC-automaton is similar to the original AC-Automaton. For a state $\mathsf{sort}(\mathbb{P}_i)[:j]$, the output function of the state $\mathsf{output}(S(\mathbb{P}_i[:j]))$ is the set of patterns $\mathbb{P}_\ell \in D$ such that $\mathbb{P}_\ell \bowtie \mathbb{P}_i[k:j]$ for some $1 \le k \le j$. Algorithm 16

Algorithm 17: Multi-track AC-automaton output function construction algorithm

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

constructs the output function of a multi-track AC-automaton.

Lemma 7.12. *Algorithm 17 constructs the output function of the multi-track AC-automaton of a dictionary $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ in $O(dM \log \sigma)$ time.*

Proof: Updating the output function can be performed in $O(1)$ time by using lists to save the output function, and update it by concatenating the list (see [1]). The proof is similar to the proofs of Lemmas 15 and 16. ■

From Lemmas 15, 15, 16, and 17, we get the following theorem.

Theorem 7.13. *The multi-track AC-automaton of a dictionary $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ can be constructed in $O(dM \log \sigma)$ time.*

By using the multi-track AC-automaton of D, we can perform permuted dictionary matching on a text $\mathbb{T}$ as shown in Algorithm 18. Let *activeState* be the current state of the multi-track AC-automaton. For each position i on $\mathbb{T}$, Algorithm 18 uses the sorted index of $\mathbb{T}[i - \mathsf{depth}(activeState) :]$ to determine permutation of $\mathbb{T}[i]$ used in the goto function.

Algorithm 18: Permuted dictionary matching algorithm by using multi-track AC-automaton.

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

Theorem 7.14. *Algorithm 18 performs permuted dictionary matching on a multi-track text* $\mathbb{T}$ *in* $O(nN \log \sigma)$ *time.*

Proof: The running time of Algorithm 18 can be evaluated by counting the number of executions of $\delta(s, \mathbb{C})$. First, for each i, $\delta(s, \mathbb{C})$ is executed at least once on *activeState* transition. Next, $\delta(s, \mathbb{C})$ is executed to check whether the transition is NULL or not. In this case, the number of executions of $\delta(s, \mathbb{C})$ is the same as that of flink. The latter is at most n, because whenever $\delta(s, \mathbb{C})$ is executed, the depth of *activeState* is increased by one, and whenever flink is executed, the depth of *activeState* is decreased by at least one. Therefore, the number of executions of $\delta(s, \mathbb{C})$ is $O(n)$. ■

7.4 Multi-track permuted matching automaton

In this section, we will describe a data structure called a multi-track permuted matching automata that can perform permuted pattern matching on a multi-track text $\mathbb{T}$ online, by preprocessing a multi-track pattern $\mathbb{P}$. A multi-track permuted matching automaton consists two functions, goto and function. In addition, each state of the multi-track permuted matching automaton has a weight in order to determine whether flink should be executed or not. Fig. 7.2 shows an example of a multi-track permuted matching

Algorithm 19: Multi-track permuted matching automaton goto function construction algorithm

```
Input: Multi-track $\mathbb{P}$
Output: Goto function
Function constructGotoFunction($\mathbb{P}$)
    create states $rootState$ and $\bot$;
    $\delta(\bot, c) \leftarrow rootState$ for all symbol $c \in \Sigma$;
    $newState \leftarrow rootState$;
    $\mathsf{weight}(\bot) \leftarrow \mathsf{weight}(rootState) \leftarrow M$;
    for $1 \leq i \leq M$ do
        $activeState \leftarrow rootState$;
        for $1 \leq j \leq m$ do
            if $\delta(activeState, \mathbb{P}[j][i]) = \mathsf{NULL}$ then
                create $newState$;
                $\delta(activeState, \mathbb{P}[j][i]) \leftarrow newState$;
                $\mathsf{weight}(newState) \leftarrow 1$;
                $activeState \leftarrow newState$;
            else
                $activeState \leftarrow \delta(activeState, \mathbb{P}[j][i])$;
                $\mathsf{weight}(activeState) \leftarrow \mathsf{weight}(activeState) + 1$;
            if $k = m$ then
                set $activeState$ as an accept state;
```

automaton.

For a multi-track pattern $\mathbb{P} = (P_1, P_2, ..., P_m)$, the multi-track permuted matching automaton of the pattern is denoted by $\mathsf{MTPMA}(\mathbb{P})$. The goto function of the multi-track permuted matching automaton is similar to that of an AC-automaton, thus, each state in $\mathsf{MTPMA}(\mathbb{P})$ represents a prefix of P_i, which is denoted by $S(w)$, where w is the string obtained by concatenating the labels of the edges from the root to the state. Each state W has a weight, which is a number of tracks containing W as a prefix. Moreover, a state W is called an *accept state* if $W = P_i$ for some i. Algorithm 19 constructs the goto function of a multi-track permuted matching automaton.

Lemma 7.15. *Algorithm 19 constructs the goto function of* $\mathsf{MTPMA}(\mathbb{P})$ *in* $O(mM \log \sigma)$ *time.*

Algorithm 20: Multi-track permuted matching automaton failure function construction algorithm

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

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

Next, we will define the failure function of a multi-track permuted matching automaton. Let S_j be the set of states that have depth j where $S(P_i[:j]) \in S_j$. The failure link of the state is $\mathsf{flink}(P_i[:j]) = P_k[:\ell] \in S_\ell$ such that $P_k[:\ell]$ is a proper suffix of $P_i[:j]$ and $\mathbb{P}[:\ell]$ is the longest prefix of $\mathbb{P}$ that permuted matches with a proper suffix of $\mathbb{P}[:j]$. The latter condition is similar to that of the multi-track KMP algorithm, which can be computed by using a multi-track border array.

Algorithm 20 constructs the failure function of the MTPMA($\mathbb{P}$). For each state s, we use multi-track border array to determine the depth of flink(s).

Theorem 7.16. *Algorithm 20 constructs the failure function of* $\mathbb{P}$ *in* $O(mM \log \sigma)$ *time.*

Proof: Similarly to the proof of Theorem 7.11, the failure and goto functions are executed $O(mM)$ times. Moreover, execution time of the failure function is $O(1)$ and that of the goto function is $O(\log \sigma)$. ■

Algorithm 21: Multi-track permuted matching automaton matching algorithm

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

Finally, by using MTPMA($\mathbb{P}$), Algorithm 21 can perform permuted pattern matching on a multi-track text $\mathbb{T}$. Algorithm 21 uses N pointers *activeStates* to pointing the current states. Note that all *activeStates* always have the same depth. Algorithm 21 uses two conditions to determine whether it should execute the failure function or not. The first condition is when it cannot find the goto transition, and the second condition is when the number of state pointers in the state is more than the weight of the state. If any of the *activeStates* is fail, then all of the *activeStates* execute the failure function, otherwise *activeStates* execute the goto function.

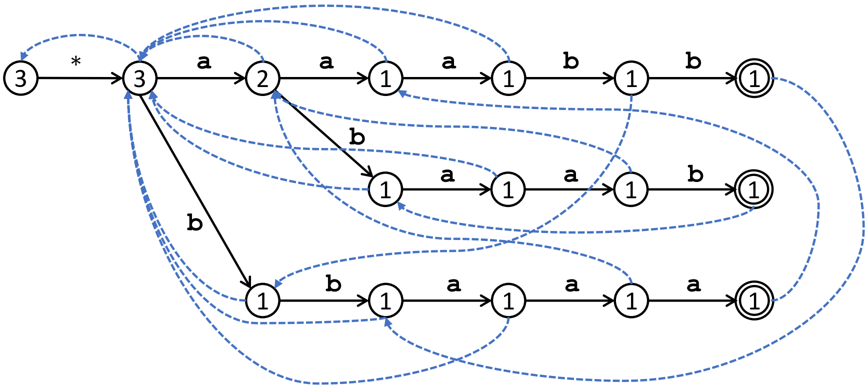


Figure 7.2: Multi-track permuted matching automaton of $\mathbb{P} = (\texttt{aaabb}, \texttt{abaab}, \texttt{bbaaa})$. The asterisk * is a special symbol that matches with any symbols in Σ.

Theorem 7.17. *By using* MTPMA($\mathbb{P}$) *Algorithm 21 performs permuted pattern match on a multi-track string* $\mathbb{T}$ *in* $O(nN \log \sigma)$ *time.*

Proof: Similarly to the proof of Theorem 7.14, the number of executions of the failure and goto function is $O(nN)$. Since the execution time of the failure function is $O(1)$ and the goto function is $O(\log \sigma)$, Algorithm 21 runs in $O(nN \log \sigma)$ time. ■

7.5 Multi-track Boyer-Moore and Horspool algorithms

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

7.5.1 Multi-track Boyer-Moore algorithm

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

Algorithm 22: MT-BM and MT-H preprocessing functions

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

Algorithm 23: Multi-track Boyer-Moore algorithm

```
Function MTBM($\mathbb{T}, \mathbb{P}$)
    compute $\mathsf{RI}(\mathbb{T}[: i])$ for $1 \leq i \leq n$;
    compute $\mathsf{RI}(\mathbb{P}[: i])$ for $1 \leq i \leq m$;
    $BadSym \leftarrow \mathsf{ComputeBadSym}(\mathbb{P})$;
    $GoodSuf \leftarrow \mathsf{ComputeGoodSuf}(\mathbb{P})$;
    $j \leftarrow 0$;
    while $j \leq n - m + 1$ do
        $i \leftarrow m$;
        while $i > 0$ and $\mathbb{T}[i + j]\langle \mathsf{RI}(\mathbb{T}[: j + m]) \rangle = \mathbb{P}[i]\langle \mathsf{RI}(\mathbb{P}[: m]) \rangle$ do
            $i \leftarrow i - 1$;
        if $i \leq 0$ then
            output $j + 1$;
            $j \leftarrow j + GoodSuf[0]$;
        else
            $j \leftarrow j + \max(GoodSuf[i], BadSym(\mathbb{T}[i + j]\langle \mathsf{RI}(\mathbb{T}[: i + j]) \rangle) - (m - i))$;
```

Definition 7.18 (Good suffixes on multi-track). *For multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, let $A[i] = \{0 < s < i \mid \mathbb{P}[i - s + 1 : m - s] \bowtie \mathbb{P}[i + 1 : m], \mathbb{P}[i - s : m - s] \not\bowtie \mathbb{P}[i : m]\}$ and $B[i] = \{i \leq s < m \mid \mathbb{P}[1 : m - s] \bowtie \mathbb{P}[s + 1 : m]\}$. The good suffix array is defined as $GoodSuf_{\mathbb{P}}[m] = 1$ and $GoodSuf_{\mathbb{P}}[i] = \min A[i] \cup B[i] \cup \{m\}$ for $0 \leq i < m$.*

Definition 7.19 (Bad symbol on multi-track). *For multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$ and a multi-track symbol $\mathbb{C}$, $BadSym(\mathbb{C})$ is the first occurrence position of $\mathsf{sort}(\mathbb{C})$ in $\mathbb{P}^R[2 :]$. The function $BadSym(\mathbb{C})$ returns m if there is no occurrence of $\mathsf{sort}(\mathbb{C})$ in $\mathbb{P}^R[2 :]$.*

In the implementation, *suf* and *GoodSuf* can be represented as arrays, while *BadSym* can be realized in a trie of the multi-track symbols. We perform permuted-match instead of exact match when computing *GoodSuf*. We also use another array *suf* to compute *GoodSuf*.

Definition 7.20 (Suffixes). *For a multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, $suf[i]$ is the maximum value of l such that $\mathbb{P}[i - l + 1 : i] \bowtie \mathbb{P}[m - l + 1 : m]$ for $1 \leq i \leq m$.*

Algorithm 22 shows how to construct *GoodSuf* and *BadSym*. The array *GoodSuf* is

Algorithm 24: Multi-track Horspool algorithm

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

computed by ComputeGoodSuf, which uses array *suf* computed by ComputeSuf. Note that we compute RI at the beginning (Lines 2 and 29) of the algorithm and will not recompute them when we use the values later.

Lemma 7.21. *The function ComputeSuf computes the array suf in $O(m(M + \sigma))$ time.*

Proof: First, $\mathsf{RI}(\mathbb{P}[: i])$ can be computed in $O(m(M + \sigma))$ time by using radix sort. The **for** loop is executed $m - 1$ times and the **while** loop at line 12 is executed at most m times through the whole run, because k is always reduced in each loop. Comparison of two multi-track symbols of the pattern that executed in each loop can be computed in $O(M)$ time. ■

Lemma 7.22. *The function ComputeGoodSuf computes GoodSuf in $O(m)$ time.*

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

Lemma 7.23. *The function ComputeBadSym computes BadSym in $O(m(M \log \sigma + \sigma))$ time.*

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

By using both *GoodSuf* and *BadSym*, MT-BM outputs the positions of the text that are permuted-matched with the pattern. The matching algorithm of MT-BM is shown in Algorithm 23.

Theorem 7.24. *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-BM outputs the positions of the text that permuted-match with the pattern online in $O(nN(m+\log\sigma)+n(N+\sigma))$ time in the worst case with $O(m(M\log\sigma+\sigma))$ time preprocessing.*

Proof: From Lemmas 7.21, 7.22, and 7.23, Algorithm 23 needs $O(m(M\log\sigma+\sigma))$ time for preprocessing. Next, $\mathsf{RI}(\mathbb{T}[:i])$ can be computed in $O(n(N+\sigma))$ time by using radix sort. In the outer **while** loop starting at line 7, the value of j is increased by at least 1, so the loop is executed at most $n-m+2$ times. In each execution of the outer loop, the inner **while** loop is executed at most m times, where multi-track symbols of the pattern and the text can be compared in $O(N)$ time. *BadSym* can be accessed in $O(N\log\sigma)$ time and *GoodSuf* can be executed in $O(1)$ time. ■

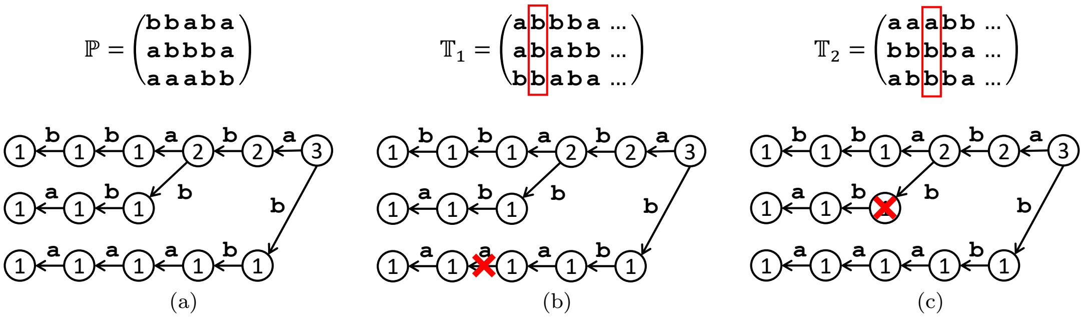
7.5.2 Multi-track Horspool algorithm

Similarly to the original Horspool algorithm, the multi-track Horspool algorithm (MT-H) uses *BadSym* to shift the pattern. Algorithm 24 shows a pseudocode of MT-H. The *BadSym* to shift the pattern that can be computed in the same way as *BadSym* of MT-BM shown in Algorithm 22.

Theorem 7.25. *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-H outputs the positions of the text that are permuted-matched with the pattern in $O(nN(m+\log\sigma)+n(N+\sigma))$ time in the worst case with $O(m(M\log\sigma+\sigma))$ time preprocessing.*

Algorithm 25: Track-trie construction algorithm

```
Function constructTrackTrie($\mathbb{P}$)
    $newNode \leftarrow root$;
    weight($root$) $\leftarrow M$;
    for $i \leftarrow 1$ to $M$ do
        $activeNode \leftarrow root$;
        for $j \leftarrow m$ to $1$ do
            if $\delta(activeNode, \mathbb{P}[j][i]) =$ Null then
                $newNode \leftarrow newNode + 1$;
                weight($newNode$) $\leftarrow 1$;
                $\delta(activeNode, \mathbb{P}[j][i]) \leftarrow newNode$;
                $activeNode \leftarrow newNode$;
            else
                $activeNode \leftarrow \delta(activeNode, \mathbb{P}[j][i])$;
                weight($activeNode$) $\leftarrow$ weight($activeNode$) $+ 1$;
```



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

Figure 7.3: (a) Track trie of $\mathbb{P} = (\mathtt{bbaba}, \mathtt{abbba}, \mathtt{aaabb})$, (b) example of mismatch when the track trie cannot find the transition, (c) example of mismatch when the number of tracks is more than the weight of the node.

Proof: Similar to the proof of Theorem 7.24, beside MT-H uses *BadSym* only. ■

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

The two algorithms presented in the previous subsections decide if two multi-tracks permuted-match by sorting them. In this subsection, we present another idea for this task using a data structure called a *track trie*. The track trie TrackTrie($\mathbb{P}$) of a multi-track $\mathbb{P}$ stores all the reversed strings of the tracks of $\mathbb{P}$, that is, $\{P_1^R, P_2^R, \ldots, P_M^R\}$. Fig. 7.3(a)

Algorithm 26: Track-trie matching algorithm

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

shows the track trie of a multi-track pattern $\mathbb{P} = (\mathtt{aaabb}, \mathtt{abbba}, \mathtt{bbaba})$.

Algorithm 25 is the construction algorithm of $\mathsf{TrackTrie}(\mathbb{P})$. For a node s of $\mathsf{TrackTrie}(\mathbb{P})$ and a symbol $c \in \Sigma$, the goto function $\delta(s, c)$ returns the child of s that has an edge labeled c. We naturally extend it to the domain Σ^* by $\delta(s, \varepsilon) = s$ and $\delta(s, aw) = \delta(\delta(s, a), w)$ for any $a \in \Sigma$ and $w \in \Sigma^*$. We also associate a weight with each node to find mismatch on a text, as we will explain later.

Theorem 7.26. *Algorithm 25 constructs* $\mathsf{TrackTrie}(\mathbb{P})$ *in* $O(mM \log \sigma)$ *time.*

Proof: The function *goto* can be calculated in $O(\log \sigma)$ time by binary search. On each execution of the inner **for** loop (line 6), Algorithm 25 executes *goto* to check child nodes of *activeNode*. If there is no node with an edge labeled $\mathbb{P}[j][i]$, then a new node is constructed, which can be done in $O(1)$ time. On the other hand, if there is a node with an edge labeled $\mathbb{P}[j][i]$, Algorithm 25 accesses the child node and then increases its weight by one. The total number of iterations of the inner loop is mM. ■

For a given multi-track text $\mathbb{T}$ and a position i, Algorithm 26 finds a mismatch position in two cases; (1) when a track cannot find its *goto* destination, and (2) when the number of tracks that have the same string w is more than the weight of the node that represents the string $\delta(root, w)$. Those mismatch conditions are illustrated in Fig. 7.3 (b) and (c), respectively. Fig. 7.3 (b) shows that the track trie cannot find a transition for the second symbol `b` of the third track. On the other hand, Fig. 7.3 (c) shows that $\mathbb{T}_2[3:]$ has two '`bba`' on its track, however the $\mathbb{P}[3:]$ has only one '`bba`' on its track, i.e. the node that represents '`bba`' has one on its weight.

Theorem 7.27. *Given a multi-track text $\mathbb{T}$ and a position j, Algorithm 26 finds a mismatch position in the pattern in $O(mM \log \sigma)$ time.*

Proof: For each position $i + j$ on the text, Algorithm 26 executes *goto* to check whether *activeNodes*$[k]$ has a child node with an edge labeled $\mathbb{T}[i+j][k]$ for $1 \leq k \leq M$. If there is no child node with an edge labeled $\mathbb{T}[i+j][k]$, then Algorithm 26 considers it as mismatch and returns the mismatch position. On the other hand, if there is such a child node, Algorithm 26 changes *activeNodes*$[k]$ to the child node, and then checks whether the number of tracks of $\mathbb{T}[i+j:]$ that contain $\mathbb{T}[k][i+j:i+m]$ as a prefix is more than the weight of the child node. If the number of tracks exceeds the weight, then Algorithm 26 treats it as mismatch and returns the mismatch position. The total number of iterations of the inner loop is at most mM. ■

Although the worst case time complexity remains the same, by using track-trie, both MT-BM and MT-H can match the pattern to the text practically faster, because we do not need to compute the reverse sorted index of the text. First, we construct the track-trie of the pattern by using constructTrackTrie($\mathbb{P}$). Then, we replace line 10 (resp. line 9) of Algorithm 23 (resp. Algorithm 24) by matchTrackTrie($\mathbb{T}, j$) to find a mismatch position.

7.6 Filtration algorithm on multi-track string

In this section we propose filtration algorithms for permuted pattern matching. The filtration algorithm uses a function to transform a multi-track string into another sequence. Then, the algorithm implement a pattern matching algorithm to the transformed pattern

Algorithm 27: Filtration algorithm for permuted pattern matching

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

and text to get candidate positions where the pattern may permuted-matches. Finally, every candidate position is checked whether the pattern is permuted-matched in each position or not.

The filtration algorithm uses a function ϕ that inputs a multi-track $\mathbb{W}$ and outputs a sequence of length $|\mathbb{W}|_{len}$. The function ϕ must have a *false-positive* property, that is for two multi-tracks $\mathbb{X}$ and $\mathbb{Y}$, if $\mathbb{X} \bowtie \mathbb{Y}$ then $\phi(\mathbb{X}) = \phi(\mathbb{Y})$. We can use any hash function such as Karp-Rabin fingerprint [47] and locality-sensitive hashing [44]. We will describe the filtration algorithm by using a simple function β that is defined as follows.

Definition 7.28 (bucket function β)**.** *For* $\mathbb{W} = (W_1, W_2, \ldots, W_N)$*,* $\beta(\mathbb{W}) = (B_1, B_2, \ldots, B_\sigma)$ *such that* $B_j[i]$ *is the number of* j*-th symbol in* $\mathbb{Z}[i]$*.*

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

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

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

Algorithm 27 shows an example of filtration algorithm for permuted pattern matching. This algorithm uses β as a hash function and KMP algorithm as a patten matching algorithm. First, the algorithm computes $\mathbb{P}' = \beta(\mathbb{P})$ and $\mathbb{T}' = \beta(\mathbb{T})$, and then constructs the border array $Border_{\mathbb{P}'}$ by the same procedure as the KMP algorithm. Next, the algorithm performs pattern matching on $\mathbb{P}'$ and $\mathbb{T}'$. When mismatch is occurred, the pattern is shifted by using $Border_{\mathbb{P}'}$. If $\mathbb{P}'$ matches $\mathbb{T}'[i : j]$, then the algorithm checks whether the pattern $\mathbb{P}$ permuted-matches $\mathbb{T}[i : j]$ or not, and outputs the position if true.

Theorem 7.29. *Algorithm 27 runs in* $O(m(\sigma + M) + n(\sigma + N) + cmM)$ *time, where* c *is the number of candidates.*

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

Algorithm 28: Filtration algorithm for sub-permuted pattern matching

```
Function multiTrackFiltration($\mathbb{P}$)
    $\mathbb{T}' \leftarrow \beta(\mathbb{T})$;
    $\mathbb{P}' \leftarrow \beta(\mathbb{P})$;
    constructBorderArray($\mathbb{P}'$);
    $i \leftarrow 1$;
    $j \leftarrow 1$;
    while $i \leq n$ do
        while $j > 0$ and $\mathsf{diff}(\mathbb{P}'[j], \mathbb{T}'[i]) > 0$ do
            $j = Border_{\mathbb{P}'}[j]$;
        $i \leftarrow i + 1$;
        $j \leftarrow j + 1$;
        if $j > m$ then
            if $\mathbb{T}[i - j + 1 : i - 1] \overset{\bowtie}{\sqsubseteq} \mathbb{P}$ then
                output $(i - j + 1)$;
            $j \leftarrow Border_{\mathbb{P}'}[j]$;
Function constructBorderArray($\mathbb{P}'$)
    $i \leftarrow 1$;
    $j \leftarrow 1$;
    $Border_{\mathbb{P}'}[1] \leftarrow 0$;
    $AllBorder_{\mathbb{P}'}[1] \leftarrow \{0\}$;
    for $1 < i \leq m$ do
        $AllBorder_{\mathbb{P}'}[i] \leftarrow \{0\}$;
        for $b \in AllBorder_{\mathbb{P}'}[i - 1]$ do
            if $\mathsf{diff}(\mathbb{P}'[i], \mathbb{P}'[b + 1]) \leq N - M$ then
                $AllBorder_{\mathbb{P}'}[i] \leftarrow AllBorder_{\mathbb{P}'}[i] \cup \{b + 1\}$;
        $Border_{\mathbb{P}'}[i] \leftarrow \max AllBorder_{\mathbb{P}'}[i]$;
Function diff($X$, $Y$)
    $sum \leftarrow 0$;
    for $k = 1$ to $|X|$ do
        $sum \leftarrow sum + \max(0, Y[k] - X[k])$;
    return $sum$;
```

We can extend Algorithm 27 to apply it to the sub-permuted pattern matching problem. This algorithm also uses β to transform the pattern and the text, and uses the KMP algorithm on the transformed pattern and text. Moreover, this algorithm uses $\mathsf{diff}(X, Y)$ function instead of $X \neq Y$ to loosen the condition to perform sub-permuted pattern matching.

Algorithm 28 shows a pseudocode of the filtration algorithm for sub-permuted pattern matching. First, it transforms $\mathbb{P}$ to multi-track bucket $\mathbb{P}'$ and constructs the border array $Border_{\mathbb{P}'}$ of the multi-track bucket by using a function $\mathsf{diff}(X, Y) = \sum_{i=1}^{\sigma} \max(0, Y[i] - X[i])$. We need to find all borders of $\mathbb{P}'[1 : i]$ for $1 \leq i \leq m$ in order to compute the border array of $\mathbb{P}'$ in this algorithm. Then, it finds candidates of occurrence position by using $Border_{\mathbb{P}'}$.

7.7 Duel-and-sweep algorithm for permuted pattern matching

In this section, we will propose an algorithm for permuted pattern matching based on the duel-and-sweep algorithm for pattern matching [4, 70]. Similarly to duel-and-sweep algorithm described in Chapters 2 and 6, the duel-and-sweep algorithm for permuted pattern matching consists of two stages, dueling stage and sweeping stage. In the dueling stage, the duel-and-sweep algorithm for permuted pattern matching uses positions and *witness tries* as witnesses. We use *longest common extension* queries and *Z-arrays* for multi-track in order to compute the witnesses efficiently. In the sweeping stage, the algorithm uses multi-track permuted matching automaton to check remaining candidates efficiently. First, we will introduce the longest common extension and Z-array for multi-track.

7.7.1 Longest common extension and Z-array for multi-tracks

The longest common extension (LCE) problem takes a string W and some position pairs (i, j) and computes the longest prefix of $W[i :]$ that matches with some prefix of $W[j :]$ for each pair (i, j), denoted by $\mathsf{LCE}_W(i, j)$. For a string W, by $O(|W|)$ time preprocessing, $\mathsf{LCE}_W(i, j)$ can be computed on $O(1)$ time [43, 56, 60]. We can naturally extend this problem for multi-tracks.

Definition 7.30 (LCE query on multi-track)**.** *Given a multi-track $\mathbb{W}$ and some position pairs, computes the longest prefix of $\mathbb{W}[i:]$ that matches with some prefix of $\mathbb{W}[j:]$ for each position pair (i,j), denoted by $\mathsf{LCE}_{\mathbb{W}}(i,j)$.*

Katsura et al. [48] proposed permuted pattern matching algorithm by using LCE queries for string and generalized suffix arrays. We adapt their algorithm to answer LCE queries on $\mathbb{W}$. First, we compute the suffix array and LCP array of $W_1W_2...W_{|\mathbb{W}|_{num}}$, then compute $\mathsf{SI}(\mathbb{W}[i:])$ for $1 \le i \le |\mathbb{W}|_{len}$. Let $\mathsf{SI}(\mathbb{W}[i:]) = (r_{i,1}, r_{i,2}, \ldots, r_{i,|\mathbb{W}|_{num}})$ and $\mathsf{SI}(\mathbb{W}[j:]) = (r_{j,1}, r_{j,2}, \ldots, r_{j,|\mathbb{W}|_{num}})$. $\mathsf{LCE}_{\mathbb{W}}(i,j)$ can be computed by computing $\min\{\mathsf{LCP}(W_{r_{i,1}}, W_{r_{j,1}}), \ldots, \mathsf{LCP}(W_{r_{i,|\mathbb{W}|_{num}}}, W_{r_{j,|\mathbb{W}|_{num}}}), |\mathbb{W}|_{len}\}$.

Lemma 7.31. *$\mathsf{LCE}_{\mathbb{W}}(i,j)$ can be computed in $O(|\mathbb{W}|_{num})$ time with $O(|\mathbb{W}|_{num}|\mathbb{W}|_{len})$ preprocessing time.*

Proof: The suffix array and LCP array of $W_1W_2...W_{|\mathbb{W}|_{num}}$, also $\mathsf{SI}(\mathbb{W}[i:])$ for $1 \le i \le |\mathbb{W}|_{len}$ can be computed in $O(|\mathbb{W}|_{num}|\mathbb{W}|_{len})$ time. $\mathsf{LCP}(W_{r_{i,k}}, W_{r_{j,k}})$ can be computed in $O(1)$ time by using an LCE query on string $W_1W_2...W_{|\mathbb{W}|_{num}}$. Since we need to compute $\mathsf{LCP}(W_{r_{i,k}}, W_{r_{j,k}})$ for $1 \le k \le |\mathbb{W}|_{num}$, $\mathsf{LCE}_{\mathbb{W}}(i,j)$ can be computed in $O(|\mathbb{W}|_{num})$ time. ■

Next, we will describe the Z-arrays for multi-tracks. Z-array or Z-function of a string is an array that store the length of the longest common prefix of suffixes of the string with the string itself [37]. Later, Hasan et al. [38] proposed a modification to Z-array for order-preserving pattern matching. We propose another modification of Z-array that can be used for permuted pattern matching. For a multi-track $\mathbb{P}$, the *Z-array for a multi-track* of $\mathbb{P}$ is defined by

$$Z_{\mathbb{P}}[i] = \max_{1 \le j \le |\mathbb{P}|_{len}-i+1} \{j \mid \mathbb{P}[1:j] \bowtie \mathbb{P}[i:i+j-1]\} \quad \text{for each } 1 \le i \le |\mathbb{P}|_{len}.$$

In other words, $Z_{\mathbb{P}}[i]$ is the length of the longest prefix of $\mathbb{P}[i]$ that permuted matches some prefix of $\mathbb{P}$. Algorithm 29 shows a multi-track Z-array construction algorithm.

Algorithm 29: Multi-track Z-array algorithm

```
1  Function constructMTZArray($\mathbb{P}$)
2      compute $SI(\mathbb{P}[i:])$ for $1 \leq i \leq m$;
3      $Z_{\mathbb{P}}[1] \leftarrow m$;
4      $L \leftarrow 1$;
5      $R \leftarrow 1$;
6      for $i = 2$ to $m$ do
7          $Z_{\mathbb{P}}[i] \leftarrow 0$;
8          if $i \leq R$ then
9              $Z_{\mathbb{P}}[i] \leftarrow \min(R - i + 1, Z_{\mathbb{P}}[i - L + 1])$;
10         while $i + Z_{\mathbb{P}}[i] \leq m$ and $\mathbb{P}[Z_{\mathbb{P}}[i] + 1]\langle \mathsf{SI}(\mathbb{P}[1:]) \rangle \neq \mathbb{P}[Z_{\mathbb{P}}[i] + i]\langle \mathsf{SI}(\mathbb{P}[i:]) \rangle$ do
11             $Z_{\mathbb{P}}[i] \leftarrow Z_{\mathbb{P}}[i] + 1$;
12         if $i + Z_{\mathbb{P}}[i] - 1 > R$ then
13             $L \leftarrow i$;
14             $R \leftarrow i + Z_{\mathbb{P}}[i] - 1$;
15     return $Z_{\mathbb{P}}$;
```

Lemma 7.32. *For a multi-track $\mathbb{P}$ of length m and track count M, the multi-track Z-array $Z_{\mathbb{P}}$ of $\mathbb{P}$ can be computed in $O(mM)$ time.*

Proof: $\mathsf{SI}(\mathbb{P})[i:]$ for $1 \leq i \leq m$ can be computed in $O(mM)$ time. The key point of the computation time of the algorithm is the number of executions of line 11 in the while loop. For each iteration of the for loop, if $i > R$, the number of executions of line 11 is the same or less than the amount of increase in the value of R. Otherwise, there are two cases when $i \leq R$. The first case is $Z_{\mathbb{P}}[i] \leftarrow \min(R-i+1, Z_{\mathbb{P}}[i-L+1]) = R-i+1$. In this case, the number of executions of line 11 is the same or less than the amount of increase in the value of R. The second case is $Z_{\mathbb{P}}[i] \leftarrow \min(R - i + 1, Z_{\mathbb{P}}[i - L + 1]) = Z_{\mathbb{P}}[i - L + 1]$. In this case the line 11 will not be executed. Therefore, the number of executions of line 11 is the same or less than the amount of increase in the value of R overall. Since $R \leq m$ and a comparison of multi-track symbol takes $O(M)$ time, the algorithm runs in $O(mM)$ time. ■

7.7.2 Witness table

The duel-and-sweep algorithm for permuted pattern matching uses one or two positions as a witness. For each offset $0 < a < m$, when the overlapped regions not permuted match, from Lemma 7.2, we can use one position if there is a position i such that $\mathbb{P}[i] \not\bowtie \mathbb{P}[i+a]$ or a pair of positions $\langle i, j \rangle$ such that $\mathbb{P}[i]\mathbb{P}[j] \not\bowtie \mathbb{P}[i+a]\mathbb{P}[j+a]$ as a witness by Lemma 7.3. Note that there is at least one witness for the second case by Lemma 7.3, but not for the first case.

Consider dueling a pair of candidate positions x and $x+a$ on the text. For the case that one position i as a witness, we can duel the candidates by comparing $\mathsf{sort}(\mathbb{T}[x+a+i-1])$ with $\mathsf{sort}(\mathbb{P}[i])$. If $\mathsf{sort}(\mathbb{T}[x+a+i-1]) = \mathsf{sort}(\mathbb{P}[i])$ we eliminate x or we eliminate $x+a$ otherwise. For the case a pair of positions $\langle i, j \rangle$ as a witness, we can duel the candidates by comparing $\mathsf{sort}(\mathbb{T}[x+a+i-1]\mathbb{T}[x+a+j-1])$ with $\mathsf{sort}(\mathbb{P}[i]\mathbb{P}[j])$. If $\mathsf{sort}(\mathbb{T}[x+a+i-1]\mathbb{T}[x+a+j-1]) = \mathsf{sort}(\mathbb{P}[i]\mathbb{P}[j])$ we eliminate x or we eliminate $x+a$ otherwise. However, for both cases, we need to sort the multi-track that be used for duel, which is inefficient. In order perform duel faster, we use track-tries (see Subsection 7.5.3) instead of sorting to compare the multi-tracks.

Next, we describe how to construct a *witness table* for $\mathbb{P}$. Algorithm 30 shows a witness table construction algorithm. The witness table $WIT_{\mathbb{P}}$ of $\mathbb{P}$ is an array of length $m-1$, such that $WIT_{P}[a]$ contains a tuple of a witness and its trie for offset a. We find a witness for offset a as follows. Let $j = Z_{\mathbb{P}}[a+1]+1$. If $j = m-a+1$, then no witness exists for a and define $WIT_{P}[a] = \mathsf{NULL}$. Otherwise, if $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[j:]) \rangle \neq \mathbb{P}[a+j]\langle \mathsf{SI}(\mathbb{P}[a+j:]) \rangle$ then position j is a witness by Lemma 7.2 and $WIT_{\mathbb{P}}[a] = \langle j, \mathsf{TrackTrie}(\mathbb{P}[j]) \rangle$. If $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[j:]) \rangle = \mathbb{P}[a+j]\langle \mathsf{SI}(\mathbb{P}[a+j:]) \rangle$ we need to find another position i such that $\langle i, j \rangle$ is a witness. We can find such position by using following lemma.

Lemma 7.33. *Let $\mathbb{P}^R$ be the reversed multi-track of $\mathbb{P}$ and $R(k) = m-k+1$ for any integer $1 \leq k \leq m$. The pair $\langle i, j \rangle$ where $i = R(\mathsf{LCE}_{\mathbb{P}}(R(j), R(a+j))+1)$ and $j = Z_{\mathbb{P}}[a+1]+1$ is a witness pair of offset a. Note that $R(k)$ in $\mathbb{P}^R$ is corresponded to position k in $\mathbb{P}$.*

Algorithm 30: Algorithm for constructing the witness table $WIT_{\mathbb{P}}$

```
Function MTWitness($P$)
    compute $Z_{\mathbb{P}}$ and LCE query data structure for $\mathbb{P}$;
    for $a = 1$ to $m - 1$ do
        $j \leftarrow Z_P[a + 1] + 1$;
        if $j = m - a + 1$ then
            $WIT_P[a] \leftarrow \mathsf{NULL}$;
        else if $\mathbb{P}[j]\langle \mathsf{SI}(\mathbb{P}[j :]) \rangle \neq \mathbb{P}[a + j]\langle \mathsf{SI}(\mathbb{P}[a + j :]) \rangle$ then
            $WIT_P[a] \leftarrow (j, \mathsf{constructTrackTrie}(\mathbb{P}[j]))$;
        else
            $i = R(\mathsf{LCE}_{\mathbb{R}}(R(j), R(a + j)) + 1)$;
            $WIT_P[a] \leftarrow (\langle i, j \rangle, \mathsf{constructTrackTrie}(\mathbb{P}[i]\mathbb{P}[j]))$;
```

Proof: From $j = Z_{\mathbb{P}}[a+1]+1$ and $i = R(\mathsf{LCE}_{\mathbb{R}}(R(j), R(a+j)) + 1)$, we can derive that

$$\mathbb{P}[i : j - 1] \bowtie \mathbb{P}[a + i : a + j - 1], \tag{7.2}$$

$$\mathbb{P}[i + 1 : j] \bowtie \mathbb{P}[a + i + 1 : a + j], \tag{7.3}$$

$$\mathbb{P}[i : j] \not\bowtie \mathbb{P}[a + i : a + j]. \tag{7.4}$$

From equation (7.4), there is a pair of positions k and l such that $\mathbb{P}[k]\mathbb{P}[l] \not\bowtie \mathbb{P}[a+k]\mathbb{P}[a+l]$ and $i \leq k < l \leq j$. On the other hand, from equation (7.2) and (7.3), $\mathbb{P}[k]\mathbb{P}[l] \bowtie \mathbb{P}[a + k]\mathbb{P}[a + l]$ for $i \leq k < l \leq j - 1$ and $i + 1 \leq k < l \leq j$, respectively. Therefore, a pair of positions $\langle k, l \rangle$ that satisfy $\mathbb{P}[k]\mathbb{P}[l] \not\bowtie \mathbb{P}[a + k]\mathbb{P}[a + l]$ for $i \leq k < l \leq j$ is only $\langle i, j \rangle$ which is a witness for offset a. ■

Lemma 7.34. *For a pattern $\mathbb{P}$ of length m and track count M, Algorithm 30 constructs $WIT_{\mathbb{P}}$ in $O(mM)$ time.*

Proof: From Lemma 7.33, the algorithm constructs $WIT_{\mathbb{P}}$ correctly. The preprocessing takes $O(mM)$ time. Each multi-track symbol comparison and LCE query takes $O(M)$ time. Therefore, the algorithm runs in $O(mM)$ time. ■

7.7.3 Dueling stage

Let $\mathbb{T}_x = \mathbb{T}[x : x+m-1]$ denote the candidate that starts at the location x. In the dueling stage, we duel pairs of overlapping candidates $\mathbb{T}_x$ and $\mathbb{T}_{x+a}$ such that $\mathit{WIT}_{\mathbb{P}}[a] \neq \mathsf{NULL}$. The procedure of the dueling is described in Algorithm 31. We will explain dueling for the case where the witness contains two positions, while duel for the case where the witness contains one position can be performed similarly. Let $(\langle i, j\rangle, \mathcal{T}_a) = \mathit{WIT}_{\mathbb{P}}[a]$ be a witness and its trie for offset a. We use $\mathcal{T}_a$ to determine whether $\mathbb{T}_{x+a}[i]\mathbb{T}_{x+a}[j] \bowtie \mathbb{P}[i]\mathbb{P}[j]$ or not. If true, the algorithm eliminates position x and returns $x + a$, otherwise the algorithm returns x.

On the other hand, if T_x and T_{x+a} do not overlap or $\mathit{WIT}_{\mathbb{P}}[a] \neq \mathsf{NULL}$, i.e. $\mathbb{P}[1 : m - a] \bowtie \mathbb{P}[a + 1 : m]$, no dueling is performed on them. We say that a position x *is consistent with* $x + a$ if either $0 < a < m$ and $\mathit{WIT}_{\mathbb{P}}[a] = \mathsf{NULL}$ or $a \geq m$. Note that the consistency property is determined by a and $\mathbb{P}$ only, and x and $\mathbb{T}$ are irrelevant. The consistency property is transitive.

Lemma 7.35. *For any a, b and x such that $1 \leq a < a + b < m$ and $1 \leq x < m - a - b$, if x is consistent with $x + a$ and $x + a$ is consistent with $x + a + b$, then x is consistent with $x + a + b$.*

Proof: Since x is consistent with $x + a$, it follows that $\mathbb{P}[1 : m-a] \bowtie \mathbb{P}[a+1 : m]$, so that $\mathbb{P}[b+1 : m-a] \bowtie \mathbb{P}[(a+b)+1 : m]$. Moreover, since $x + a$ is consistent with $x + a + b$, it follows that $\mathbb{P}[1 : m - b] \bowtie \mathbb{P}[b+1 : m]$, so that $\mathbb{P}[1 : m - b - a] \bowtie \mathbb{P}[b+1 : m - a]$. Thus, $\mathbb{P}[1 : m - (a + b)] \bowtie \mathbb{P}[(a + b) + 1 : m]$, which implies that x is consistent with $x + a + b$. ■

The whole process of the dueling stage is shown in Algorithm 32. This stage eliminates candidates until all surviving candidates are pairwise consistent. The algorithm uses a stack to maintain candidates which are consistent with each other. A new candidate y will be pushed to the stack if the stack is empty. Otherwise, y is checked by comparing it

Algorithm 31: Dueling for multi-track

```
Function MTDueling($x, a$)
    $(v, \mathcal{T}_a) \leftarrow WIT_{\mathbb{P}}[a]$;
    if $v$ is a pair $\langle i, j \rangle$ then
        if matchTrackTrie($\mathcal{T}_a, \mathbb{W}[i]\mathbb{W}[j]$) then
            return $x + a$;
        else
            return $x$;
    else
        if matchTrackTrie($\mathcal{T}_a, \mathbb{W}[i]$) then
            return $x + a$;
        else
            return $x$;
Function matchTrackTrie($\mathcal{T}, \mathbb{W}$)
    $activeNode[k] \leftarrow root$ for $1 \leq k \leq |\mathbb{W}|_{num}$;
    $temp(node) \leftarrow 0$ for all $node$ in $\mathcal{T}$;
    for $i \leftarrow 1$ to $|\mathbb{W}|_{len}$ do
        for $k \leftarrow 1$ to $M$ do
            if $\delta(activeNodes, \mathbb{W}[i][k]) =$ Null then
                return false;
            else
                $activeNodes[k] \leftarrow \delta(activeNodes[k], \mathbb{W}[i][k])$;
                $temp(activeNodes[k]) \leftarrow temp(activeNodes[k]) + 1$;
                if $temp(activeNodes[k]) >$ weight($activeNodes[k]$) then
                    return false;
    return true;
```

to the topmost element x of the stack. By Lemma 7.35, if x is consistent with y, all the other elements in the stack are consistent with y. Thus we can push y to the stack. On the other hand, if x is not consistent with y, we should exclude one of the candidates by dueling them. If x wins the duel, we put x back to the stack, discard y, and get a new candidate. If y wins the duel, we exclude x and continue the comparison of y with the top element of the stack unless the stack is empty. If the stack is empty, y will be pushed to the stack.

Lemma 7.36 ([5])**.** *The dueling stage can be done in $O(n)$ time by using $WIT_{\mathbb{P}}$.*

Algorithm 32: The dueling stage algorithm for multi-track

```
Function MTDuelingStage($\mathbb{P}, \mathbb{T}$)
    create $stack$;
    for $y = 1$ to $n - m + 1$ do
        while $stack$ is not empty do
            pop $x$ from $stack$;
            if $y - x \geq m$ or $WIT_{\mathbb{P}}[y - x] = \mathsf{NULL}$ then
                push $x$ and $y$ to $stack$;
                break;
            else
                $z \leftarrow \mathsf{MTDueling}(x, y - x)$;
                if $z = x$ then
                    push $x$ to $stack$;
                    break;
        if $stack$ is empty then
            push $y$ to $stack$;
```

7.7.4 Sweeping stage

The sweeping stage for permuted pattern matching can be performed efficiently by using the consistent property of the remaining candidates. When checking T_x, we compare $\mathbb{P}\langle \mathsf{SI}(\mathbb{P}[1:])\rangle$ with $\mathbb{T}_x\langle \mathsf{SI}(\mathbb{T}[x:])\rangle$ from the left to the right. Suppose there is a mismatch at position j when comparing $\mathbb{P}$ with $\mathbb{T}_x$. If the next candidate is $\mathbb{T}_{x+a}$ where $a < j$, since $\mathbb{P}[1 : j - a - 1] \bowtie \mathbb{P}[a + 1 : j - 1] \bowtie \mathbb{T}_x[a + 1 : j - 1] = \mathbb{T}_{x+a}[1 : j - a - 1]$, we can start comparison of $\mathbb{P}$ and $\mathbb{T}_{x+a}$ from the position where the mismatch occurred. Therefore, the total number of comparison is bounded by $O(nN)$, by applying the same argument of the complexity of the multi-track KMP algorithm. Algorithm 33 shows how to perform the sweeping stage.

The comparison can be performed more efficiently by using multi-track permuted matching automaton. Although the theoretical complexity is the same, it can be computed faster practically because we do not need to compute the sorted index array of the text. Suppose there is a mismatch at position j when comparing $\mathbb{P}$ with $\mathbb{T}_x$. Let $\mathbb{T}_{x+a}$ be the next candidate. In this case, the pointers in the automaton should follow the failure link

Algorithm 33: The sweeping stage algorithm for multi-track

```
Function SweepingStage()
    while there are unchecked candidates to the right of $\mathbb{T}_x$ do
        let $\mathbb{T}_x$ be the leftmost unchecked candidate;
        if there are no candidates overlapping with $\mathbb{T}_x$ then
            if $\mathbb{T} \not\approx P$ then
                eliminate $\mathbb{T}_x$;
        else
            let $\mathbb{T}_{x+a}$ be the leftmost candidate that overlaps with $\mathbb{T}_x$;
            if $\mathbb{T}_x \approx P$ then
                start checking $\mathbb{T}_{x+a}$ from the location $m - a + 1$;
            else
                let $j$ be the mismatch position;
                eliminate $\mathbb{T}_x$;
                start checking $\mathbb{T}_{x+a}$ from the location $j - a$;
```

until their depths become $j-a-1$ and start matching from the position in the text where mismatch occurred.

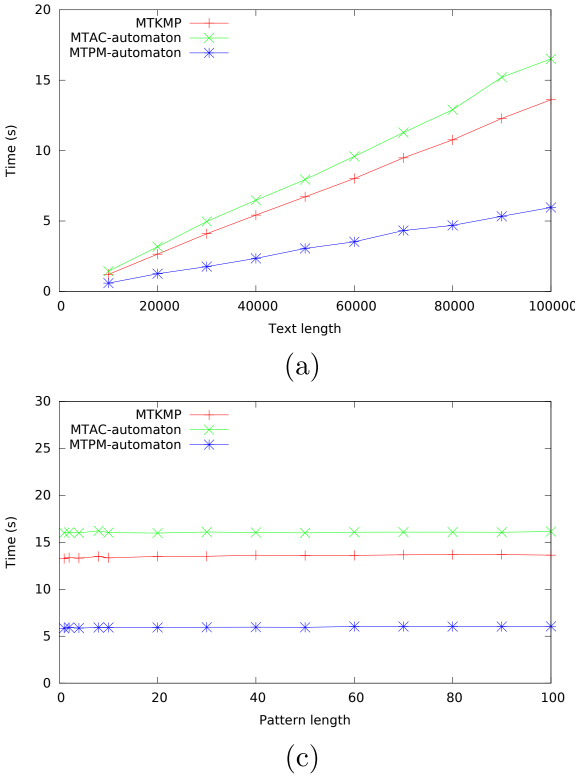
Lemma 7.37. *The sweeping stage can be completed in $O(nN)$ time.*

By Lemmas 7.32, 7.36, and 7.37, we summarize this section as follows.

Theorem 7.38. *Given a text $\mathbb{T}$ of length n and track count N, and a pattern $\mathbb{P}$ of length m and track count M, The duel-and-sweep algorithm solves the OPPM Problem in $O(nN + mM)$ time.*

7.8 Experiments

We performed some sets of experiments to evaluate the performance of the proposed algorithms practically. We measured the running time of our algorithms and the AC-automaton based algorithm [48]. All experiments are performed on a computer with Intel Xeon CPU E5-2609 8 cores 2.40 GHz, 256 GB memory, and Debian Wheezy operating system. We set the parameter values as follows, $n = 100000$, $m = 10$, $N = M = 1000$, and



(a)

(b)

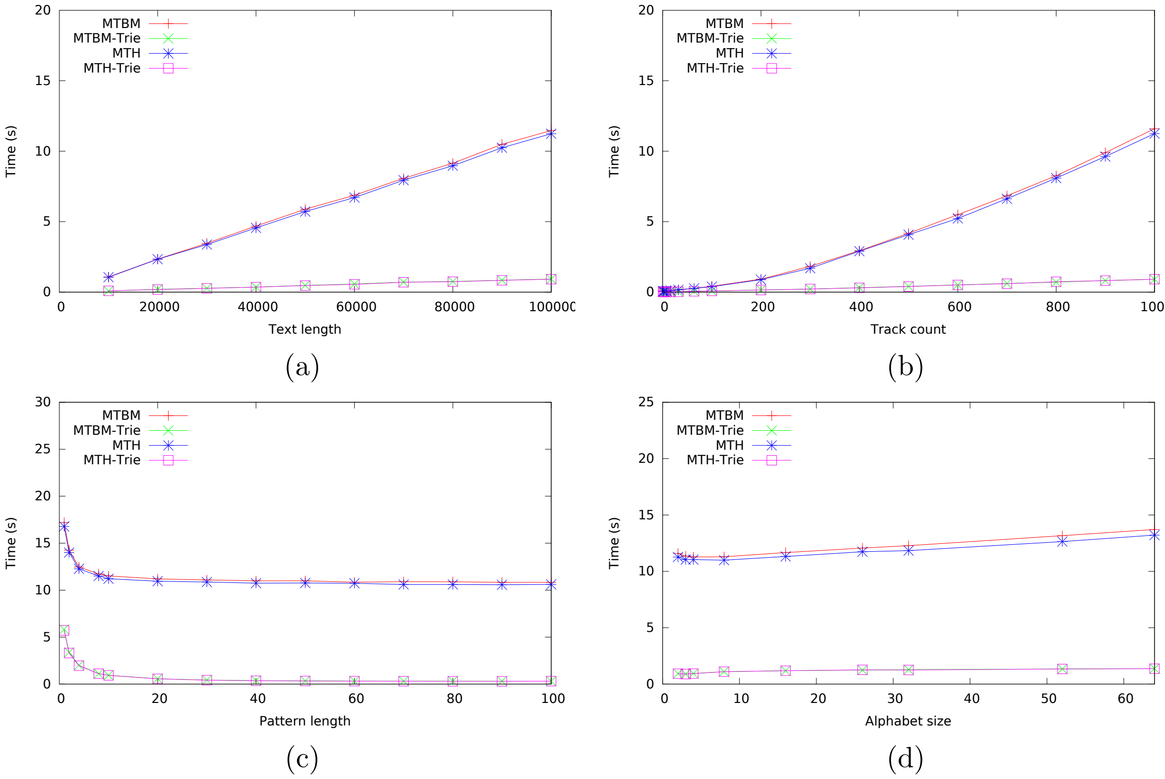
(c)

(d)

Figure 7.4: Running time of the KMP based algorithms on permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

$\sigma = 2$, and changed one of the parameters in each experiment to see the relation between the parameters and the running time of the algorithms. We used randomly generated texts and patterns.

The first set of experiments compares the running time of KMP based algorithms, namely the multi-track KMP algorithm, the multi-track AC-automaton algorithm, and the multi-track permuted matching automaton algorithm. The purposes of these experiments is to see the changes of the running time of the multi-track KMP algorithm when extended to the multi-track AC-automaton algorithm and the multi-track permuted matching automaton algorithm. The results of the experiments are shown in Fig. 7.4. From the results, we can see that the multi-track AC-automaton algorithm is slower than the multi-track KMP algorithm, because the multi-track AC-automaton algorithm uses state transitions instead of compare multi-track symbols directly. This lost is small since the multi-track AC-automaton algorithm focuses on the dictionary matching instead of



(a)

(b)

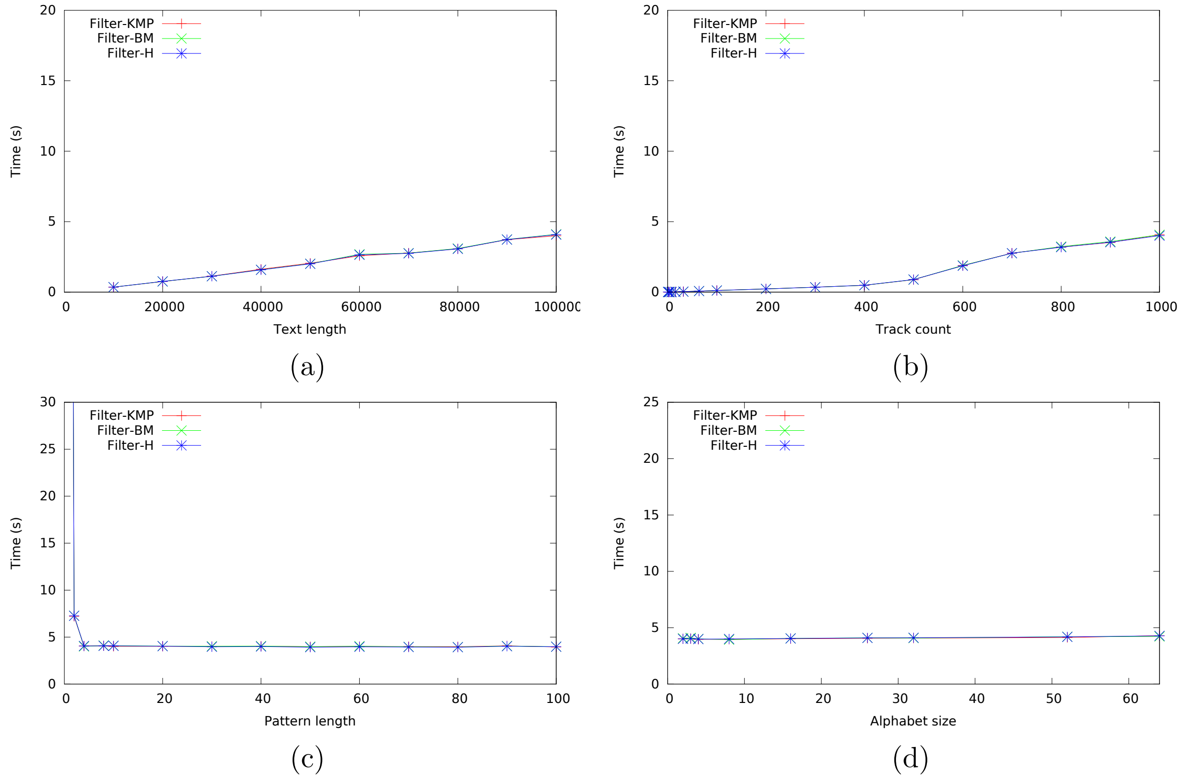
(c)

(d)

Figure 7.5: Running time of the Boyer-Moore and Horspool based algorithms on permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

the single pattern matching. Next, we can see that the multi-track permuted matching automaton algorithm is faster than the multi-track KMP algorithm, since the multi-track permuted matching automaton algorithm does not need to sort the text. However, the multi-track permuted matching automaton algorithm is slower than the multi-track KMP algorithm when the alphabet size is big, because the multi-track permuted matching automaton algorithm needs $O(\log \sigma)$ time to find state transitions.

The next experiment compares the running time of the multi-track Boyer-Moore algorithm and the multi-track Horspool algorithm. We also check the effectiveness of track tries to reduce the matching time of both algorithms. Fig. 7.5 shows the result of the experiment. While the multi-track Horspool is slightly faster than the multi-track Boyer-Moore algorithm, there is no significant difference between these algorithms. We can also see that track-tries significantly reduce the matching time of both algorithms, since we



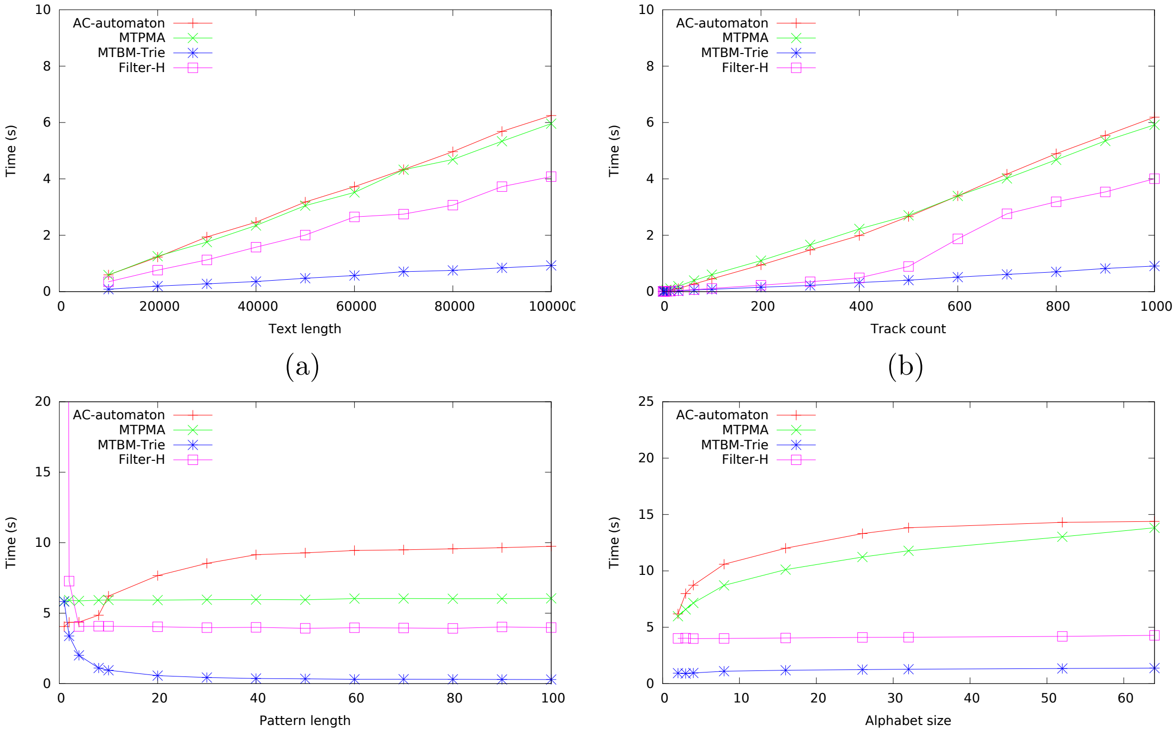
(a) (b) (c) (d)

Figure 7.6: Running time of the filtration algorithms on permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

do not need to sort the text when using track tries.

The third experiment set compares the running time of filtration algorithms. We use three algorithms, the KMP-algorithm, the Boyer-Moore algorithm, and the Horspool algorithm on filtration algorithm. Fig. 7.6 shows the running time of the filtration algorithms. There is no significant difference in running time between the filtration algorithms. We can see that the filtration algorithms are fast besides when the pattern length $m = 1$. The algorithms become very slow because there are many of occurrence position candidates when $m = 1$. The occurrence position candidates will be fewer if the pattern length is longer.

Last, we compare proposed algorithms with AC-automaton based algorithm [48]. Fig. 7.6 shows the running time of the AC-automaton based algorithm [48] and proposed algorithms. We choose the fastest algorithms among their groups. We can see that the proposed algorithms are faster than the AC-automaton based algorithm overall. The



(a)

(b)

(c)

(d)

Figure 7.7: Comparison of the running time of the AC-automaton based algorithm with proposed algorithms on permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

multi-track Boyer-Moore algorithm with track-trie is the fastest among other algorithms.

Chapter 8

Conclusion and Future Work

In this dissertation, we proposed algorithms for four types of pattern matching, including dynamic dictionary matching, parameterized pattern matching, order-preserving pattern matching, and permuted pattern matching.

In Chapter 4, we discussed the dynamic dictionary matching problem and proposed DAWG-based and AC-automaton-based algorithms to solve it. For the DAWG based algorithm, we used an additional Nearest Marked Ancestor (NMA) queries data structure to support DAWGs. In contrast, for the AC-automaton-based algorithm, no additional data structure is used for matching and only we used DAWGs are used to update the failure and output functions of the AC-automaton. Because we only use DAWGs and AC-automaton, our AC-automaton-based algorithm is easier to implement and performs dictionary matching faster than other algorithms. Furthermore, though our AC-automaton update algorithm can be considered optimal for the semi-dynamic dictionary matching, it is slower by a factor of $O(\sigma/\log\sigma)$ for the dynamic dictionary matching problem because of the increased time complexity in the case of the pattern deletion algorithm in DAWGs. The task to find faster AC-automaton update algorithms for dynamic dictionary matching remains a future work. In addition, finding an optimal algorithm for dynamic-dictionary matching for which the lower bound on solving dynamic dictionary matching problem is still unknown will be considered as well.

In Chapter 5, we present a data structure called the parameterized position heap. We can perform parameterized pattern matching efficiently by adding additional data structures to these position heaps. In our study, we achieved an optimal time for construction of parameterized position heaps; the construction time was the same as that of parameterized suffix trees. However, the matching time is slower when the parameterized position heap is used compared with the case when the suffix tree is used by $O(m|\Pi|)$ time. Though the difference is not significant if the alphabet size is constant or considerably small, we will still consider finding a method to reduce the matching time to $O(m \log(|\Sigma| + |\Pi|) + occ)$.

In Chapter 6, we discuss our duel-and-sweep algorithm for order-preserving pattern matching. The primary difference between our algorithm and the original duel-and-sweep algorithm is we use a pair of positions for a witness instead of just one position used in the original algorithm. Our duel-and-sweep algorithm is theoretically as fast as the KMP algorithm for order-preserving pattern matching, and practically faster. Because the original duel-and-sweep algorithm was proposed as a parallel computation algorithm, we will consider developing a parallel version of our duel-and-sweep algorithm. In addition, as a future work, we might consider developing a sampling algorithm for order-preserving pattern matching, which has similar properties to the duel-and-sweep algorithm as a future work.

Finally, in Chapter 6, we present some algorithms for permuted pattern matching on multi-track strings. We primarily focused on the algorithms that preprocess the pattern before performing permuted pattern matching, instead of constructing indexing structures from the text. We showed that our proposed algorithms are faster than the AC-automaton based algorithm. Moreover, we proposed an algorithm for dictionary matching on multi-track strings, which, to the best of our knowledge, is the first algorithm for this problem. However, because of its considerable difficulty owing to a larger number of permutations, only a few algorithms for sub-permuted pattern matching have been proposed. Therefore, developing an algorithm for the sub-permuted pattern matching problem remains as a future work.

In summary, we can see that some algorithms that had been proposed for a particular pattern matching problem can be applied to some other variant of pattern matching problem as well; in doing so, however, some of the algorithms are easy to modify, whereas some are not depending on the properties of the strings that are utilized in these algorithms. Therefore, by analyzing the properties of the strings, pattern matching problems, and algorithms, we can define pattern matching problem classes that can be solved a group of algorithms, thus generalizing these algorithms.

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] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*, number 1, pages 534–543. IEEE Comput. Soc, 1998.

[3] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. Technical Report RS-98-7, BRICS, 1998.

[4] Amihood Amir, Gary Benson, and Martin Farach. An Alphabet Independent Approach to Two-Dimensional Pattern Matching. *SIAM Journal on Computing*, 23(2):313–323, apr 1994.

[5] Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[6] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. Lapoutre, and Alejandro A. Schaffer. Improved Dynamic Dictionary Matching. *Information and Computation*, 119(2):258–282, jun 1995.

[7] Amihood Amir, Martin Farach, and S Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, feb 1994.

[8] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[9] Brenda S. Baker. A theory of parameterized pattern matching. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing - STOC '93*, pages 71–80, New York, New York, USA, 1993. ACM Press.

[10] Brenda S Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, 1995.

[11] Brenda S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

[12] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M.T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(C):31–55, 1985.

[13] Anselm Blumer, J. Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[14] Robert S. Boyer and J S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[15] Domenico Cantone and Simone Faro. An Efficient Skip-Search Approach to the Order-Preserving Pattern Matching Problem. In *Proceedings of the Prague Stringology Conference 2015*, pages 22–35, 2015.

[16] Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Proceedings of the sixteenth annual ACM-SIAM symposium on discrete algorithms*, pages 13–22, 2005.

[17] Tamanna Chhabra, Oğuzhan Kulekci, and Jorma Tarhio. Alternative Algorithms for Order-Preserving Matching. In *Proceedings of the Prague Stringology Conference 2015*, pages 36–46, 2015.

[18] Tamanna Chhabra and Jorma Tarhio. Order-Preserving Matching with Filtration. In *Experimental Algorithms: 13th International Symposium, SEA 2014*, volume 8504 LNCS, pages 307–314. 2014.

[19] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, feb 2015.

[20] Edward G. Coffman and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, jul 1970.

[21] Richard Cole and Ramesh Hariharan. Faster Suffix Tree Construction with Missing Suffix Links. *SIAM Journal on Computing*, 33(1):26–42, 2003.

[22] Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-Dimensional Parameterized Matching. *ACM Transactions on Algorithms*, 11(2):12:1—-12:30, 2014.

[23] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*, 72(2):450–466, 2015.

[24] Beate Commentz-Walter. A string matching algorithm fast on the average. In *the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132. 1979.

[25] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[26] Maxime Crochemore. String matching with constraints. In *Mathematical Foundations of Computer Science 1988*, pages 44–58. Springer-Verlag, Berlin/Heidelberg, 1988.

[27] Maxime Crochemore, Chiara Epifanio, Roberto Grossi, and Filippo Mignosi. Linear-size suffix tries. *Theoretical Computer Science*, 638:171–178, jul 2016.

[28] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval: 20th International Symposium, SPIRE 2013*, volume 8214 LNCS, pages 84–95. 2013.

[29] Maxime Crochemore and Wojciech Rytter. *Text algorithm.* 1994.

[30] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology.* World Scientific Publishing Co. Pte. Ltd., 2002.

[31] Satoshi Deguchi, Fumihito Higashijima, and Hideo Bannai. Parameterized suffix arrays for binary strings. In *Proceedings of the Prague Stringology Conference 2008*, pages 84–94, 2008.

[32] 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.

[33] Simone Faro and M. Oğuzhan Külekci. Efficient Algorithms for the Order Preserving Pattern Matching Problem. In Riccardo Dondi, Guillaume Fertin, and Giancarlo Mauri, editors, *Algorithmic Aspects in Information and Management: 11th International Conference, AAIM 2016*, volume 9778 of *Lecture Notes in Computer Science*, pages 185–196. Springer International Publishing, 2016.

[34] Guy Feigenblat, Ely Porat, and Ariel Shiftan. An Improved Query Time for Succinct Dynamic Dictionary Matching. In *Combinatorial Pattern Matching: 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014.*, volume 8486 LNCS, pages 120–129. 2014.

[35] Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Information Processing Letters*, 100(3):91–96, 2006.

[36] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[37] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[38] Md Mahbubul Hasan, A.S.M.Shohidull Islam, Mohammad Saifur Rahman, and M.Sohel Rahman. Order preserving pattern matching revisited. *Pattern Recognition Letters*, 55:15–21, 2015.

[39] Wing-Kai Hon, Tak-Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Succinct Index for Dynamic Dictionary Matching. In *Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009.*, volume 5878 LNCS, pages 1034–1043. 2009.

[40] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[41] Tomohiro I, Satoshi Deguchi, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Lightweight Parameterized Suffix Array Construction. In *Combinatorial Algorithms: 20th International Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28–July 2, 2009, Revised Selected Papers*, volume 5874 LNCS, pages 312–323. 2009.

[42] Ramana M. Idury and Alejandro A. Schäffer. Dynamic dictionary matching with failure functions. *Theoretical Computer Science*, 131(2):295–310, 1994.

[43] 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, dec 2010.

[44] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, pages 604–613, New York, New York, USA, 1998. ACM Press.

[45] Fuminori Ishizaki and Motomichi Toyama. An incremental update algorithm for large Aho-Corasick automaton. In *Proceedings of the 4th Forum on Data Engineering and Information Management, F11-5*, pages 1–6, 2012. (In Japanese).

[46] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Combinatorial Pattern Matching: 6th Annual Symposium, CPM 95 Espoo, Finland, July 5–7, 1995 Proceeding.*, volume 937, pages 191–204, 1995.

[47] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[48] Takashi Katsura, Kazuyuki Narisawa, Ayumi Shinohara, Hideo Bannai, and Shunsuke Inenaga. Permuted Pattern Matching on Multi-track Strings. In *SOFSEM 2013: Theory and Practice of Computer Science: 39th International Conference on Current Trends in Theory and Practice of Computer Science*, pages 280–291. 2013.

[49] Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara. Position Heaps for Permuted Pattern Matching on Multi-Track Strings. In *Proceedings of Student Research Forum Papers and Posters at SOFSEM 2015*, pages 41–53, 2015.

[50] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

[51] 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.

[52] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[53] Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

[54] Gregory Kucherov. On-line construction of position heaps. *Journal of Discrete Algorithms*, 20(August 2011):3–11, 2013.

[55] Gregory Kucherov and Michaël Rusinowitch. Matching a set of strings with variable length don't cares. *Theoretical Computer Science*, 178(1-2):129–154, may 1997.

[56] Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *The eighteenth annual ACM symposium on Theory of computing.*, pages 220–230, 1986.

[57] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[58] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[59] Bertrand Meyer. Incremental string matching. *Information Processing Letters*, 21(5):219–227, 1985.

[60] Rodrigo de Castro Miranda and Mauricio Ayala-Rincón. A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays. In *Brazilian Symposium on Bioinformatics*, pages 210–213, 2005.

[61] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *2009 Data Compression Conference*, number 60873056, pages 193–202. IEEE, mar 2009.

[62] S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 631–638. IEEE Comput. Soc. Press, 1995.

[63] Tetsuo Shibuya. Generalization of a Suffix Tree for RNA Structural Pattern Matching. *Algorithmica*, 39(1):1–19, 2004.

[64] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81*, pages 114–122, New York, New York, USA, 1981. ACM Press.

[65] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, aug 1990.

[66] Takuya Takagi, Keisuke Goto, Yuta Fujishige, Shunsuke Inenaga, and Hiroki Arimura. Linear-Size CDAWG: New Repetition-Aware Indexing and Grammar Compression. volume 10508, pages 304–316. 2017.

[67] Kazuhiko Tsuda, Masao Fuketa, and Jun-Ichi Aoe. An incremental algorithm for string pattern matching machines. *International Journal of Computer Mathematics*, 58(1-2):33–42, 1995.

[68] Yohei Ueki, Kazuyuki Narisawa, and Ayumi Shinohara. A Fast Order-Preserving Matching with q -neighborhood Filtration Using SIMD Instructions. In *SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science*. 2016.

[69] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[70] Uzi Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67(1-3):91–113, 1985.

[71] Uzi Vishkin. Deterministic Sampling – A New Technique for Fast Pattern Matching. *SIAM Journal on Computing*, 20:22–40, 1991.

[72] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

[73] Jeffery Westbrook. Fast incremental planarity testing. In *Automata, Languages and Programming*, number 1, pages 342–353, 1992.

List of Publications

Journals articles

1. Ichinari Sato, Diptarama, Kaizaburo Chubachi, Ryo Yoshinaka, and Ayumi Shinohara. Analysis of laboratories electrical energy consumption by visualization for saving electrical energy. *International Journal of Institutional Research and Management*, 1(1):53–66, 2017.

2. Diptarama, Kazuyuki Narisawa, and Ayumi Shinohara. Extension of generalized tic-tac-toe: p stones for one move. *IPSJ Journal*, 55(11):2344–2352, 2014.

Conference papers

3. Yuki Igarashi, Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. New Variants of Pattern Matching with Constants and Variables. In *the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2018)*, 611–623, 2018.

4. Davaajav Jargalsaikhan, Diptarama, Yohei Ueki, Ryo Yoshinaka, and Ayumi Shinohara. Duel and sweep algorithm for order-preserving pattern matching. In *the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2018)*, 624–635, 2018.

5. Hayato Mizumoto, Shota Todoroki, Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. An efficient query learning algorithm for zero-suppressed binary decision diagrams In *the 28th International Conference on Algorithmic Learning Theory (ALT 2017)*, 360–371, Kyoto, Japan, October 2017

6. Diptarama, Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara. Position heaps for parameterized strings. In *the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pp. 8:1–8:13, 2017.

7. Shintaro Narisada, Diptarama, Kazuyuki Narisawa, Shunsuke Inenaga, and Ayumi Shinohara. Computing longest single-arm-gapped palindromes in a string. In *the 43rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2017)*, pp. 375–386, 2017.

8. Yohei Ueki, Diptarama, Masatoshi Kurihara, Yoshiaki Matsuoka, Kazuyuki Narisawa, Ryo Yoshinaka, Hideo Bannai, Shunsuke Inenaga, and Ayumi Shinohara. Longest common subsequence in at least k length order-isomorphic substrings. In *the 43rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2017)*, pp. 363–374, 2017.

9. Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. AC-Automaton update algorithm for semi-dynamic dictionary matching. In *the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pp. 110–121, 2016.

10. Ichinari Sato, Kaizaburo Chubachi, and Diptarama. Evaluation of machine learning methods on SPiCe. In *the Sequence PredictIction ChallengE (SPiCe)*, pp. 149–153, 2016.

11. Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. Fast full permuted pattern matching algorithms on multi-track Strings. In *the Prague Stringology Conference 2016 (PSC 2016)*, pp. 7–21, 2016.

12. Ichinari Sato, Diptarama, and Ayumi Shinohara. Visualization and analysis of electrical energy consumption in laboratories. In *5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pp. 509–512, 2016.

13. Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. QBF encoding of generalized tic-tac-toe. In *the 4th International Workshop on Quantified Boolean Formulas (QBF 2016)*, pp. 14–26, 2016.

14. Diptarama, Yohei Ueki, Kazuyuki Narisawa, and Ayumi Shinohara. KMP based pattern matching algorithms for multi-track strings. In *Student Research Forum Papers and Posters at SOFSEM2016*, pp. 100–107, 2016.

15. Diptarama, Kazuyuki Narisawa, and Ayumi Shinohara. Drawing strategies for generalized tic-tac-toe (p, q). In *the International Conference on Progress in Applied Mathematics in Sciences and Engineering (PIAMSE 2015)*, 8 pages, 2015.

16. Diptarama, Yuya Ishiguro, Kazuyuki Narisawa, Ayumi Shinohara, and Charles Jordan. Solving generalized tic-tac-toe by using QBF solver. In *the 20th Game Programming Workshop 2015 (GPW-15)*, pp. 154–161, 2015.

17. Yuya Ishiguro, Diptarama, Kazuyuki Narisawa, and Ayumi Shinohara. Analysis of generalized tic-tac-toe in torus board. In *the 20th Game Programming Workshop 2015 (GPW-15)*, pp. 162–167, 2015.

18. Diptarama, Kazuyuki Narisawa, and Ayumi Shinohara. Extension of Generalized Tic-Tac-Toe: p stones for one move. In *the 18th Game Programming Workshop 2013 (GPW-13)*, pp. 19–26, 2013.