Static and Dynamic Compressed Data Structures for String Processing

西本, 崇晃

https://doi.org/10.15017/1807062

Static and Dynamic Compressed Data Structures for String Processing

Takaaki Nishimoto

January, 2017

Abstract

In recent years, computer technologies has been developed rapidly. In particular, development of storage devices enables us to store large-scale data, and it has become popular to discover new knowledge by processing text data, which is called text data mining. However, mining from large-scale text data with realistic running time and working space requires to perform basic string processing operations (e.g., reporting all occurrences of a given pattern in a text) in efficient running time and working space. For this purpose, studies of string processing algorithms has been developed in recent years. In this thesis, we focus on data structures for string processing in compressed space for the following basic string processing operations.

(1) A *signature encoding* [Mehlhorn et al., Algorithmica 17(2):183-198, 1997] of a string T is an extended context free grammar representing the single T, which is determined by *locally consistent parsing* [Mehlhorn et al., Algorithmica 17(2):183-198, 1997]. Since it has many applications, it is important to update and construct efficiently signature encodings. We show that signature encodings can be updated in compressed space, and also, let T be a dynamic string of current length N and maximal length N_{max} (i.e, $N \leq N_{max}$ always holds), $LZ77_{wo}(T)$ be the Lempel-Ziv77 (LZ77) factorization without self reference of size $z = |LZ77_{wo}(T)|$ representing T, and $\mathcal{S}$ be an SLP of size n generating T. Then, we show that the signature encoding $\mathcal{G}$ of size w for T can be constructed (i) in $O(Nf_{\mathcal{A}})$ time and $O(w)$ working space from T, (ii) in $O(N)$ time and working space from T, (iii) in $O(zf_{\mathcal{A}} \log N \log^* N_{max})$ time and $O(w)$ working space from $LZ77_{wo}(T)$, (iv) in $O(nf_{\mathcal{A}} \log N \log^* N_{max})$ time and $O(w)$ working space from $\mathcal{S}$, and (v) in $O(n \log\log(n \log^* N_{max}) \log N \log^* N_{max})$ time and $O(n \log^* N_{max} + w)$ working space from $\mathcal{S}$, where $f_{\mathcal{A}} = O(\min\{\frac{\log\log N_{max} \log\log w}{\log\log\log N_{max}}, \sqrt{\frac{\log w}{\log\log w}}\})$.

(2) A *Longest Common Extension* (LCE) query on a text T of length N asks for the length of the longest common prefix of suffixes starting at given two positions. We show

that the signature encoding $\mathcal{G}$ of a dynamic string T has a capability to support LCE queries in $O(\log N + \log \ell \log^* N_{max})$ time, where ℓ is the answer to the query, N_{max} is the maximal length of T. Since signature encodings can support update T and the size $w = O(\min(z \log N \log^* N_{max}, N)$ holds, This data structure is the first fully dynamic LCE data structure working in compressed space, where $z = |LZ77_{wo}(T)|$. On top of the above contributions, we show several applications of our data structures which improve previous best known results on grammar-compressed string processing.

(3) A *find* query on a text T asks for all occurrences positions of a given pattern P in T. A data structure supporting find queries for T is called *index* for T. In this thesis, we propose a new *dynamic compressed index* of $O(w)$ space for a dynamic text T of current length N and maximal length N_{max}, where $w = O(\min(z \log N \log^* N_{max}, N))$ is the size of the signature encoding of T and $z = |LZ77_{wo}(T)|$. Our index supports find queries in T in $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* N_{max}(\log N + \log |P| \log^* N_{max}) + occ \log N)$ time and insertion/deletion of a substring of length y in $O((y + \log N \log^* N_{max}) \log w \log N \log^* N_{max})$ time, where *occ* is the number of occurrences of P in T. We also propose a new space-efficient LZ77 factorization algorithm for a given text of length N, which runs in $O(N f_{\mathcal{A}} + z \log w \log^3 N (\log^* N)^2)$ time with $O(w)$ working space.

(4) A *dictionary matching* query on a set of patterns (dictionary) Π asks for all occurrences positions of patterns in Π in a text T given in a streaming fashion. We address a variant of the dictionary matching problem where the dictionary is represented by an SLP. For a given SLP-compressed dictionary Π of size n and height h representing m patterns of total length N, we present an $O(n^2 \log N)$-size representation of Aho-Corasick automaton which recognizes all occurrences of the patterns in Π in amortized $O(h + m)$ running time per character, where $m = |\Pi|$ and h is the height of the derivation tree of the SLP representing Π. We also propose an algorithm to construct this compressed Aho-Corasick automaton in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space. In a spacial case where Π represents only a single pattern, we present an $O(n \log N)$-size representation of the Morris-Pratt automaton which permits us to find all occurrences of the pattern in amortized $O(h)$ running time per character, and we show how to construct this representation in $O(n^3 \log n \log N)$ time with $O(n^2 \log N)$ working space.

Acknowledgements

First, I would like to thank Professor Masayuki Takeda who is my supervisor and the committee chair of this thesis. I would also like to thank Professor Eiji Takimoto, Associate Professor Daisuke Ikeda, and Associate Professor Shunsuke Inenaga, who are the members of the committee of my thesis. I would like to thank Kyushu University.

I would like to thank Professor Masayuki Takeda. I could take the exam for the doctor's cource of Kyushu University thanks to him. I would like to thank Associate Professor Hideo Bannai. He maintains the server of our laboratory. We appreciated him whenever the server was recovered. I would like to thank Associate Professor Shunsuke Inenaga. I discussed problems of stringology with him many times, and also, he gave cup noodles to me every year. I would like to Professor Eiji Takimoto and Associate Professor Kohei Hatano. They gave me some comments in weekly seminar. I would like to thank Tomohiro I who is my research colleague. I would like to thank Gitlab, Subversion, TeXstudio, SourceTree, PowerPoint, Skype and Mattermost which are usuful tools for my research.

The results in the thesis were partially published in the proc. of CIAA' 13, the proc. of MFCS' 16, and the proc. of PSC' 16, and also, the journal version of CIAA' 13 were published in Theoretical Computer Science by Elsevier. I am thankful for all editors, committees, anonymous referees, and publishers.

Last, I thank my parents for their support.

Contents

Chapter 1

Introduction

In recent years, computer technologies has been developed rapidly. In particular, development of storage devices enables us to store large-scale data, and it has become popular to discover new knowledge by processing text data, which is called *text data mining*. However, mining from large-scale text data with realistic running time and working space requires to perform basic string processing operations (e.g., reporting all occurrences of a given pattern in a text) in efficient running time and working space. For this purpose, studies of string processing algorithms has been developed in recent years.

In this thesis, we focus on data structures for string processing in compressed space. The 'compressed space' means that the size of data structure for a text T can be small if we can represent T in small space. For example, Bille et al. proposed a data structure of $O(n)$ space for a *straight line program* (SLP in short) of size n which supports random access queries [12], where an SLP is a context-free grammar in the Chomsky normal form which generates a single string. This means that the size of the data structure is small when the grammar size n is small compared to the original text length N It is well known that outputs of various grammar-based compression algorithms (e.g., [49, 38]), as well as those of dictionary-based compression algorithms (e.g., [66, 64, 65, 59]), can be regarded as, or be quickly transformed to, SLPs [54]. This means that if we can efficiently compress T by such grammar-based compression algorithms, we can construct the data structure for T in small space. Hence this data structure is in compressed space.

A data structure for a text supporting pattern search queries, is called the *text index*. For another example, Gagie et al. [23] proposed an index of $O(z \log \log N)$ space for T of length N which supports the searching for a given pattern in T, where z is the number of factors in the LZ77 factorization [65] of T (we formally define LZ77 factorization later

in Definition 1). This means that the size of the index is small when z is small. Note that the LZ77 factorization can represent T in $O(z)$ space and z is very small when T is a highly repetitive text. Hence this text index is in compressed space. In addition, there exist many studies of compressed data structures for a text (e.g. [9, 32] for recent work).

Also, we consider *dynamic* data structures for string processing. The 'dynamic data structure' means the data structure for a data supporting the update of the data. For example, a dynamic index for a text T supports pattern search queries for T and updating T. Note that such text(string), which allows to be updated, is called a dynamic text(string).

In this thesis, a dynamic string T allows the following two update operations.

- $INSERT(Y, i)$: update $T \leftarrow T[1..i-1]YT[i..|T|]$ for a given string Y and an integer i.
- $DELETE(j, y)$: update $T \leftarrow T[1..j-1]T[j+y..|T|]$ for two given integers j and y.

A static data structure for T can be the dynamic data structure by reconstructing the data structure every time T is updated. However, the update of such data structure is clearly inefficient when T is a large-scale text. Hence, it is important to develop an efficient update algorithm for the data structure. There exist some studies on dynamic data structures for dynamic text (or dynamic set of strings) (e.g. [4, 19, 48, 47] for recent work).

In this thesis, we address some basic string processing problems and present a static or dynamic data structures for these problems.

1.1 Signature Encoding

A *signature encoding* of a string T is an extended context free grammar generating T, which is determined by *locally consistent parsing* [44]. We formally define the signature encoding later in Chapter 2.

The signature encoding is first proposed by Mehlhorn et al. [44] for equality testing on a dynamic set of strings [44]. They also showed that signature encodings for strings can be updated under concatenate/split update operations.

Signature encodings have many applications. Alstrup et al. used signature encodings to present a pattern matching algorithm on a dynamic set of strings [4, 3]. In their papers,

they also showed that signature encodings can support the longest common prefix (LCP) and the longest common suffix (LCS) queries on a dynamic set of strings. We describe the detail later in Chapter 5.

Sahinalp and Vishkin showed that the upper bound of the size of signature encodings of a text T of length N is $O(z \log N \log^* N)$ space, where z is the number of factors in the LZ77 factorization of T [55]. This means that signature encodings can be used as data compression.

Cormode and Muthukrishnan proposed a new data structure called *the edit sensitive parsing*(ESP), which is an another version of signature encodings, and they showed it can efficiently solve a special edit distance problem [16], and also, there exists a compressed index called ESP-Index which uses ESP [60, 61]. We describe the detail of ESP-Index in Chapter 5.

To use these applications of signature encodings efficiently, it is important to consider efficient algorithms of construction of signature encodings. Hence we consider the following construction problem for various inputs representing a text.

Problem 1 *Construct the signature encoding of an input string T of length N represented by a plain text T, an SLP $\mathcal{S}$ of size n, or the LZ77 factorization of size z.*

Next, if we can update the signature encoding of a dynamic text efficiently, then we can use these applications for dynamic texts. Hence it is also important to consider dynamic data structures maintaining a signature encoding. We consider the following problem.

Problem 2 *Construct a data structure supporting $Expr(e)$ queries for a dynamic string T of current length N and maximal length N_{max}(i.e, $N \leq N_{max}$ always holds), where $Expr(e)$ returns $expr$ for a given variable e in the signature encoding $\mathcal{G}$ of T such that $e \to expr$ is in $\mathcal{G}$.*

Related work

We describe related work about the update problem of signature encodings. Mehlhorn et al. [44] proposed the first data structure maintaining signature encodings on dynamic strings. A dynamic strings $\Pi \subset \Sigma^*$ allows the following update operations.

- $CONCAT(s_1, s_2)$: update $\Pi \leftarrow \Pi \cup \{s_1 s_2\}$ for two given strings $s_1, s_2 \in \Pi$.

- $SPLIT(s, i)$: update $\Pi \leftarrow \Pi \cup \{s[1..i-1], s[i..|s|]\}$ for a given string $s \in \Pi$ and a given integer i.
- $CHAR(c)$: update $\Pi \leftarrow \Pi \cup \{c\}$ for a given character $c \in \Sigma$.

Let N_{max}^{Π} be the maximal length of sum of string length in Π, i.e. $\sum_{s \in \Pi} |s| \leq N_{max}^{\Pi}$ always holds. Alstrup et al. improved the data structure and update algorithms of Mehlhorn et al. [4, 3]. Their data structure supports $CONCAT$ and $SPLIT$ operations in $O(\mu(w, N_{max}^{\Pi}) \log N' \log^* N_{max}^{\Pi})$ time, where N' is the maximal length in input strings and created strings, w is the size of the signature encoding, and $\mu(a, b)$ is the time for membership queries on a set of a integers from an b-element universe. Note that they did not consider an update operation which removes a string from Π. Hence the size of their data structure depends on the number of update operations.

Next we describe related work about the construction problem of signature encodings. As already mentioned, Mehlhorn et al. proposed concatenation and split algorithms. We can construct signature encodings by these algorithms. In addition to this, there exists a few work of construction algorithms of ESP.

Cormode and Muthukrishnan proposed a construction algorihtm which outputs an ESP of a single text T of length N for T in $O(N \log^* N)$ running time and $O(N)$ space [16]. Goto et al. proposed a construction algorithm which outputs an ESP of T for a run-length encoding text R representing T in $O(N \log^* N)$ running time [26]. They also proposed a construction algorithm which outputs an ESP of T for an SLP of size n representing T in $O(n \log^2 N + m)$ running time, where m is the size of the output ESP.

1.2 Dynamic Longest Common Extension Problem

A Longest Common Extension (LCE) query on a text T of length N asks to compute the length of the longest common prefix of suffixes starting at given two positions. This fundamental query appears at the heart of many string processing problems (see text book [28] for example), and hence, efficient data structures to answer LCE queries gain a great attention.

We consider a *Dynamic LCE problem* in the following description.

Problem 3 (Dynamic LCE) *For a dynamic string T of current length N and maximal length N_{max}, construct a data structure which supports LCE queries on T.*

The above problem is called *Static LCE problem* if the data structure does not support update operations of T.

We describe research history of the static LCE problem. A classic solution is to use a data structure for lowest common ancestor queries [8] on the suffix tree of T. Although this achieves constant query time, the $\Theta(|T|)$ space needed for the data structure is too large to apply it to large scale data. Hence, recent work focuses on reducing space usage at the expense of query time. For example, time-space trade-offs of LCE data structure have been extensively studied [11, 62]. In [21], LCE data structures on edit sensitive parsing, a variant of signature encoding, was used for sparse suffix sorting, but again, they did not focus on working in compressed space.

Another direction to reduce space is to utilize a compressed structure of T, which is advantageous when T is highly compressible. There are several LCE data structures working on grammar-compressed string T represented by an SLP of size n. The best known deterministic LCE data structure is due to I et al. [32], which supports LCE queries in $O(h \log N)$ time, and occupies $O(n^2)$ space, where h is the height of the derivation tree of a given SLP. Their data structure can be built in $O(hn^2)$ time directly from the SLP.

Bille et al. [10] showed a Monte Carlo randomized data structure which supports LCE queries in $O(\log N \log \ell)$ time, where ℓ is the output of the LCE query. Their data structure requires only $O(n)$ space, but requires $O(N)$ time to construct. Very recently, Bille et al. [9] showed a faster Monte Carlo randomized data structure of $O(n)$ space which supports LCE queries in $O(\log N + \log^2 \ell)$ time. The preprocessing time of this new data structure is not given in [9]. Note that, given the LZ77-compression of size z of T, we can convert it into an SLP of size $n = O(z \log \frac{N}{z})$ [54] and then apply the above results.

Very recently, a more faster LCE data structure is proposed by I [31]. His data structure uses $O(z \log(N/z))$ space and supports LCE queries in $O(\log N)$ deterministic time, where z is the number of factors in LZ77 factorization of T. Note that $z \leq n$ holds for an SLP of size n representing T [54].

Next, we describe research history of the dynamic LCE problem. To our knowledge, data structures based on signature encodings (see Section 1.1) for a dynamic set of strings proposed by Alstrup et al. is the first data structures which can answer efficiently LCE queries on dynamic strings [4, 3]. Note that their data structures do not support LCE queries directly, however, they support LCP queries and concatenate/split update operations for dynamic strings. This means that they can answer LCE queries on

a string $T \in \Pi$ for dynamic strings Π using constant LCP queries and split operations in $O(\mu(w, N^{\Pi}_{max}) \log N \log^* N^{\Pi}_{max})$ deterministic time, where $N = |T|$, w is the size of the signature encodings.

Very recently, Gawrychowski et al. improved the results by pursuing advantages of randomized approach other than the hash table [25]. It should be noted that the algorithms in [4, 3, 25] can support LCE queries in $O(\log N)$ time by combining split operations and LCP queries although it also is not explicitly mentioned, where N is the length of the handled dynamic string. However, [4, 3] and [25] do not focus on the fact that signature encodings can work in compressed space, and also, their data structures do not consider an update operation which removes a string from the dynamic strings. These mean that the size of their data structures depends on the number of update operations.

1.3 Dynamic Index Problem

The *dynamic text indexing problem* is defined formally in the following description.

Problem 4 (Dynamic Text Indexing Problem) *For a dynamic string T of current length N and maximal length N_{max}, construct an index which supports FIND(P) queries, where FIND(P) returns all occurrences of a given pattern P in T.*

The above problem is called the *static text index problem* if the index is not required to support update operations of T.

We describe the research history of the text indexing problem. The static text index problem is a classical problem in computer science and there exists some indexes in linear space, i.e., suffix array [40], suffix tree [43], etc. However, these many classical indexes are not compressed.

As the size of data is growing rapidly in the last decade, many recent studies have focused on indexes working in compressed text space (see e.g. [22, 23, 15, 14]). However they are static, i.e., they have to be reconstructed from scratch when the text is modified, which makes difficult to apply them to a dynamic text.

The dynamic text index problem is also a classical problem. In 1994, the first dynamic index was proposed by Gu et al. [27]. Afterwards, many dynamic indexes was proposed (see e.g. [56, 4, 19, 25] for recent work). Note that these dynamic indexes are not compressed.

There also exists a compressed version of dynamic text index as the static text index. In 2004, Hon et al. [30] proposed the first dynamic compressed index of $O(\frac{1}{\epsilon}(NH_0 + N))$ bits of space which supports $FIND(P)$ in $O(|P| \log^2 N(\log^\epsilon N + \log|\Sigma|) + occ \log^{1+\epsilon} N)$ time and $INSERT(Y, i)$ and $DELETE(j, y)$ in $O((y + \sqrt{N}) \log^{2+\epsilon} N)$ amortized time, where $0 < \epsilon \leq 1$ and $H_0 \leq \log|\Sigma|$ denotes the zeroth order empirical entropy of the text of length N [30]. Salson et al. [58] also proposed a dynamic compressed index, called the *dynamic FM-Index*. Although their approach works well in practice, updates require $O(N \log N)$ time in the worst case. To our knowledge, these are the only existing dynamic compressed indexes to date.

In relation to the dynamic index problem, there exists the library management problem of maintaining a text collection (a set of text strings) allowing for insertion/deletion of texts (see [47] for recent work). While in the dynamic index problem a single text is edited by insertion/deletion of substrings, in the library management problem a text can be inserted to or deleted from the collection. Hence, algorithms for the library management problem cannot be directly applied to the dynamic index problem.

1.4 LZ77 Factorization Problem

A string sequence $f_1, \ldots, f_k$ is called a *factorization* of a string s if $f_1 \cdots f_k = s$ holds. The *LZ77 factorization without self-reference* $LZ77_{wo}(T)$ [65] of T is formally defined as follows.

Definition 1 *For a string s, let $LZ77_{wo}(s)$ denote the factorization $f_1, \ldots, f_z$ of s such that for $1 \leq i \leq z$, if $s[|f_1 \cdots f_{i-1}| + 1]$ is a character not occurring $f_0 \cdots f_{i-1}$ then $f_i = s[|f_1 \cdots f_{i-1}| + 1]$, otherwise f_i is the longest prefix of $f_i \cdots f_z$ such that f_i is a substring of $f_1 \cdots f_{i-1}$, where $f_0 = \epsilon$.*

Since each f_i can be represented by an integer pair $(x_i, |f_i|)$, we can represent a string T in $2z \log|T|$ bits of space by the LZ77 factorization, where x_i is an occurrence position of f_i in $f_1 \cdots f_{i-1}$. Hence the LZ77 factorization is used as data compression.

Although the primary use of the LZ77 factorization is data compression, it has been shown that it is a powerful tool for many string processing problems [24, 23]. Hence the importance of algorithms to compute the LZ77 factorization is growing. Particularly, in order to apply algorithms to large scale data, reducing the working space is an important

matter.

In this thesis, we focus on the LZ77 factorization algorithms working in compressed space. Goto et al. [26] showed how, given the grammar-like representation for string T generated by the LCA algorithm [57], to compute the LZ77 factorization of T in $O(z \log^2 m \log^3 N + m \log m \log^3 N)$ time and $O(m \log^2 m)$ space, where m is the size of the given representation. Sakamoto et al. [57] claimed that $m = O(z \log N \log^* N)$, however, it seems that in this bound they do not consider the production rules to represent maximal runs of non-terminals in the derivation tree. The bound we were able to obtain with the best of our knowledge and understanding is $m = O(z \log^2 N \log^* N)$, and hence our algorithm seems to use less space than the algorithm of Goto et al. [26]. Recently, Fischer et al. [20] showed a Monte-Carlo randomized algorithms to compute an approximation of the LZ77 factorization with at most $2z$ factors in $O(N \log N)$ time, and another approximation with at most $(i+\epsilon)z$ factors in $O(N \log^2 N)$ time for any constant $\epsilon > 0$, using $O(z)$ space each.

Another line of research is the LZ77 factorization working in compressed space in terms of Burrows-Wheeler transform (BWT) based methods. Policriti and Prezza recently proposed algorithms running in $NH_0 + o(N \log |\Sigma|) + O(|\Sigma| \log N)$ bits of space and $O(N \log N)$ time [52], or $O(R \log N)$ bits of space and $O(N \log R)$ time [53], where R is the number of runs in the BWT of the reversed string of T.

1.5 Grammar Compressed Dictionary Matching Problem

A dictionary matching query on a set of patterns Π (called the *dictionary*) asks for all occurrences positions of patterns in Π within a text T given in a streaming fashion. A dictionary pattern matching problem is, for a given dictionary Π, to construct a data structure which answer dictionary matching queries for Π. This is a classical pattern matching problem and we can efficiently answer these queries by constructing an Aho-Corasick (AC) automaton [2] for Π.

In this thesis, we introduce a new, yet another variant of the problem, where the dictionary is given in compressed form. In particular, we are interested in a setting where a dictionary is given in compressed form in advance, and the text is given in a streaming fashion. A typical application would be an SDI (Selective Dissemination of Information)

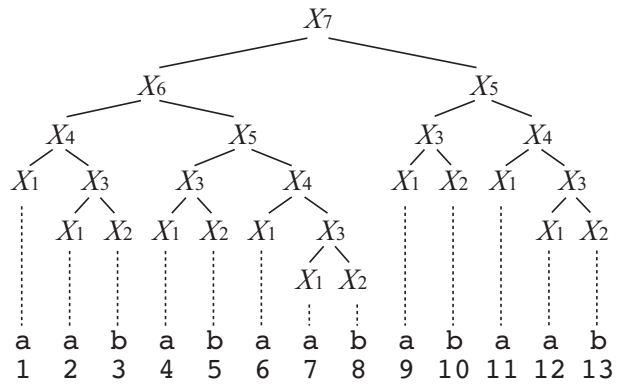

Figure 1.1: Let SLP $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, where $\mathcal{D} = \{X_1 \to \mathtt{a},\ X_2 \to \mathtt{b},\ X_3 \to X_1X_2,\ X_4 \to X_1X_3,\ X_5 \to X_3X_4,\ X_6 \to X_4X_5,\ X_7 \to X_6X_5\}$, $\mathcal{V} = \{X_1, \ldots, X_7\}$, $\Sigma = \{a, b\}$ and $S = X_7$. Then $\mathcal{S}$ represents the string aababaababaab and Figure 1.1 is the derivation tree of $\mathcal{S}$.

service.

As already mentioned, SLP can be used as a compressed representation of a text. In this thesis, we consider representing a dictionary by an SLP. Specifically, we use an SLP to represent a dictionary consisting of m patterns, by designating m variables in the SLP as the start symbols.

More formally, we extend SLPs so as to represent dictionaries as follows:

Definition 2 *A* dictionary SLP *(DSLP) is an ordered pair $\langle \mathcal{S}, m \rangle$ of an SLP of size n and a positive integer $m \in [1..n]$. The last m variables $X_{n-m+1}, \ldots, X_n$ of $\mathcal{S}$ are designated as the* start variables*.*

Let $\Pi_{\langle \mathcal{S},m \rangle}$ denote the dictionary consisting of the strings derived from the start variables. We note that DSLP $\langle \mathcal{S}, 1 \rangle$ is equivalent to SLP $\mathcal{S}$. We are now ready to give a formal definition of our problem.

Problem 5 (Grammar Compressed Dictionary Matching Problem) *For a given DSLP $\langle \mathcal{S}, m \rangle$, construct a data structure supporting dictionary matching queries for $\Pi_{\langle \mathcal{S},m \rangle}$.*

See also Figure 1.1 and Example 1.

Example 1 *Consider DSLP $\langle \mathcal{S}, 3 \rangle$ with $\mathcal{S}$ that is shown in Figure 1.1, i.e., $\Pi_{\langle \mathcal{S},3 \rangle} = \{val(X_5), val(X_6), val(X_7)\} = \{\mathtt{abaab}, \mathtt{aababaab}, \mathtt{aababaababaab}\}$, where $val(X)$ represents the derived string by X. Given text "$T =$ abaababaab", the dictionary matching*

query for $\Pi_{\langle \mathcal{S},3 \rangle}$ *answers* $(1, X_5)$, $(3, X_6)$ *and* $(6, X_5)$.

Related Work

As already mentioned, we can efficiently solve the dictionary matching problem by constructing an Aho-Corasick (AC) automaton for Π. A *succinct* representation of AC automaton has been proposed [7], which requires $k(\log \sigma + 3.443 + o(1)) + m(3 \log(N/m) + O(1))$ bits of space, where k is the number of states in the AC automaton, σ is the alphabet size, and N is the total length of patterns in Π Using this succinct AC automaton one can conduct dictionary matching for a given text t in $O(|t| + occ)$ time, where occ is the output size. Recently an LZ78 based compressed string dictionary was considered [5], which stores a set of strings and identifies each string with unique identifier in a compressed space. Using the data structure one can retrieve the string of a given ID and reversely lookup the ID of a given string.

To the best of our knowledge, there do not exist studies which directly solves the grammar compressed dictionary matching problem. Note that we can solve this problem by decompressing $\mathcal{S}$ and constructing AC automaton for $\Pi_{\langle \mathcal{S},m \rangle}$, however, the total length N can be as large as $\Theta(2^n)$. This means that a naïve method which decompresses $\mathcal{S}$ takes exponential time and space in the worst case.

1.6 Our Contributions

Our contributions are as follows.

Signature Encoding

Our main contribution is the following theorems.

Theorem 1 *For Problem 2, there exists a data structure of* $O(w)$ *space which supports* $Expr(e)$ *queries in* $O(1)$ *time,* $INSERT(Y, i)$ *and* $DELETE(j, y)$ *operations in* $O(f_{\mathcal{A}}(y + \log N \log^* N_{max}))$ *time, and* $INSERT'(j, y, i)$ *in* $O(f_{\mathcal{A}} \log N \log^* N_{max})$ *time, where* $f(a, b)$ *denotes the time for predecessor/successor queries on a set of a integers from an* b*-element universe,* $f_{\mathcal{A}} = f(w, 4N_{max})$*, and* $INSERT'(j, y, i)$ *updates* $T \leftarrow T[..i-1]T[j..j+y-1]T[i..]$ *for given integers* i, j *and* y.

If we use the best known deterministic and dynamic predecessor/successor data structure of linear space, then $f(a,b) = O(\min\{\frac{\log\log b \log\log a}{\log\log\log b}, \sqrt{\frac{\log a}{\log\log a}}\})$.

Mehlhorn et al. [44] and Alstrup et al. [4, 3] showed signature encodings can be updated under concatenate/split operations. However, they did not explicitly show that the update algorithm maintains $O(w)$ space because they did not focus on the working space. Our update algorithm of Theorem 1 clearly runs in $O(w)$ space. Hence this result is significant.

Theorem 2 *For Problem 1, we can construct the signature encoding $\mathcal{G}$ of size w for T*

(1a) *in $O(Nf_{\mathcal{A}})$ time and $O(w)$ working space from T,*

(1b) *in $O(N)$ time and working space from T,*

(2) *in $O(zf_{\mathcal{A}} \log N \log^* N_{max})$ time and $O(w)$ working space from $LZ77_{wo}(T)$,*

(3a) *in $O(nf_{\mathcal{A}} \log N \log^* N_{max})$ time and $O(w)$ working space from $\mathcal{S}$, and*

(3b) *in $O(n \log\log(n \log^* N_{max}) \log N \log^* N_{max})$ time and $O(n \log^* N_{max} + w)$ working space from $\mathcal{S}$.*

Note that $N_{max} = N$ holds if we handle T of $\mathcal{G}$ as a static text. This is the first construction algorithms from LZ77 factorization and SLPs, however, proofs of the results (2) and (3a) are straightforward from the previous work [4, 3]. (1a) and (1b) are also trivial results but to our knowledge, nobody explicitly showed these. (3b) is clearly a new result and this is more efficient than (3a) when $n = O(z \log N)$. Hence this result is also significant.

Furthermore, we give a new application of signature encodings. See Section 3.3. These results were published in [51]

Dynamic LCE Problem

Our main contribution is the following theorem.

Theorem 3 (LCE queries) *Let $\mathcal{G}$ denote the signature encoding for a dynamic string T of current length N and maximal length N_{max}. Then $\mathcal{G}$ supports LCE queries on T in $O(\log N + \log \ell \log^* N_{max})$ time, where ℓ is the answer to the query.*

Theorem 2 shows that the signature encoding $\mathcal{G}$ can be updated efficiently, and also that the size of $\mathcal{G}$ of T is $O(z \log N \log^* N_{max})$ space [55] where $z = |LZ77_{wo}(T)|$. Hence a

signature encoding of T works in compressed space as a dynamic LCE data structures for T.

The remarks on our contributions are listed in the following:

- We present the first fully dynamic LCE data structure working in compressed space. The size of our data structure does not depend on the number of update operations. This is different from dynamic LCE data structures proposed by [4, 3] and [25].
- This is the first compressed LCE data structure which can be constructed efficiently by a given SLP. This result is important for applications in compressed string processing, where the task is to process a given compressed representation of string(s) without explicit decompression. In particular, we use the result (3b) of Theorem 2 to show several applications which improve previous best known results. See Section 3.3. Note that the static LCE data structure proposed by I can be constructed faster than our data structure by a given SLP and his data structure is more space efficient. Hence his result can improve all theorems in Section 3.3.

These results were originally published in [51]

Dynamic Index Problem

We propose a new dynamic index in compressed space, as follows:

Theorem 4 *For Problem 4, there exists a dynamic index of size* $w = O(\min(z \log N \log^* N_{max}, N))$ *supports* $INSERT(Y, i)$ *and* $DELETE(j, y)$ *in amortized* $O((|Y| + \log N \log^* N_{max}) \log w \log N \log^* N_{max})$ *time,* $INSERT'(j, y, i)$ *in amortized* $O(\log w(\log N \log^* N_{max})^2)$ *time, and* $FIND(P)$ *in* $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* N_{max}(\log N + \log |P| \log^* N_{max}) + occ \log N)$ *time, where* $z = |LZ77_{wo}(T)|$ *and* $occ = |FIND(P)|$.

Since $z \geq \log N$, $\log w = \max\{\log z, \log(\log^* N_{max})\}$. Hence, our index is able to find pattern occurrences faster than the index of Hon et al. [30], and also, our index allows faster substring insertion/deletion on the text.

LZ77 Factorization Problem

We present a new LZ77 factorization algorithm working in compressed space for a given signature encoding.

Theorem 5 *Given a signature encoding $\mathcal{G}$ of a string T, we can compute $LZ77_{wo}(T)$ in $O(z \log w \log^3 N (\log^* N)^2)$ time and $O(w)$ working space where $z = |LZ77_{wo}(T)|$, $N = |T|$, and w is the size of $\mathcal{G}$.*

Theorem 2 shows that the signature encoding $\mathcal{G}$ can be constructed efficiently from various types of inputs, in particular, in $O(Nf_{\mathcal{A}})$ time and $O(w)$ working space from uncompressed string T. Therefore we can compute LZ77 factorization of a given T of length N in $O(Nf_{\mathcal{A}} + z \log w \log^3 N (\log^* N)^2)$ time and $O(w)$ working space.

These results were originally published in [50].

Grammar Compressed Dictionary Matching Problem

Our main contribution is the following theorem.

Theorem 6 *Given any DSLP $\langle \mathcal{S}, m \rangle$ of size n that represents dictionary $\Pi_{\langle \mathcal{S}, m \rangle}$ of total length N, it is possible to build, in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space, an $O(n^2 \log N)$-size compressed automaton that recognizes all occurrences of patterns in $\Pi_{\langle \mathcal{S}, m \rangle}$ within an arbitrary string with $O(h + m)$ amortized running time per character, where h is the height of the derivation tree of $\mathcal{S}$.*

Note that our proposed data structure emulate AC automata in compressed space. To the best of our knowledge, our data structure is the first which uses grammar-based string compression to reduce space requirement of AC automata.

We also present a more space-efficient solution to the case of a single pattern, namely, a compressed representation of Morris-Pratt (MP) automaton [46] as follows.

Theorem 7 *For an SLP $\mathcal{S}$ of size n representing string T of length N, it is possible to build, in $O(n^3 \log N \log n)$ time and $O(n^2 \log N)$ space, an $O(n \log N)$-size compressed MP automaton that recognizes all occurrences of T within an arbitrary string with $O(h)$ amortized running time per character, where h is the height of the derivation tree of $\mathcal{S}$.*

Theorem 7 is the first grammar-compressed MP-automaton. Note that since MP automaton is identical to an uncompressed AC-automaton for a single pattern, we can construct a compressed MP automaton of $O(n^2 \log N)$ space by Theorem 6. Hence the compressed MP automaton is more space efficient than Theorem 6.

This result was originally published in [35, 34].

1.7 Organization

The rest of this thesis is organized as follows. In Chapter 2, we set our model, define some notations, and introduce some known properties on strings and useful known data structures. In Chapter 3, we show Theorems 1 and 2. In Chapter 4, we show Theorem 3 and describe some applications of our LCE data structures. In Chapter 5, we describe our dynamic index in compressed space and LZ77 factorization algorithm working space. Namely, we show Theorems 4 and 5. In Chapter 6, we show Theorems 6 and 7

Chapter 2

Preliminaries

2.1 Strings

Let Σ be an ordered alphabet. An element of Σ^* is called a string. A *dictionary* is a non-empty, finite subset of Σ^+. For string $s = xyz$, x, y and z are called a prefix, substring, and suffix of s, respectively. The length of string s is denoted by $|s|$. The empty string ε is a string of length 0. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \leq i \leq |s|$, $s[i]$ denotes the i-th character of s. For any $1 \leq i \leq j \leq |s|$, $s[i..j]$ denotes the substring of s that begins at position i and ends at position j. Let $s[i..] = s[i..|s|]$ and $s[..i] = s[1..i]$ for any $1 \leq i \leq |s|$. For any string s, let s^R denote the reversed string of s, that is, $s^R = s[|w|] \cdots s[2]s[1]$. For any strings s_1 and s_2, let $\mathsf{LCP}(s_1, s_2)$ (resp. $\mathsf{LCS}(s_1, s_2)$) denote the length of the longest common prefix (resp. suffix) of s_1 and s_2, and also, for two integers i, j, let $\mathsf{LCE}(s_1, s_2, i, j) = \mathsf{LCP}(s_1[i..|s_1|], s_2[j..|s_2|])$. For any strings p and s, let $Occ(p, s)$ denote all occurrence positions of p in s, namely, $Occ(p, s) = \{i \mid p = s[i..i + |p| - 1], 1 \leq i \leq |s| - |p| + 1\}$. A *dynamic string* s of maximal length N_{max} is a string allowing to be updated and $|s| \leq N_{max}$ always holds.

2.1.1 Our Model

Our model of computation is the unit-cost word RAM with machine word size of B bits, and space complexities will be evaluated by the number of machine words. Bit-oriented evaluation of space complexities can be obtained with a $\log_2 B$ multiplicative factor. The value B is set in each problem. When the input of a problem represents a static text of length N, we set $B = N$. When the represented text is dynamic and the maximal length

is N_{max}, we set $B = N_{max}$.

2.1.2 Periods and Runs of Strings

A *period* of a string s is a positive integer p such that $s[i] = s[i+p]$ for every $i \in [1..|s|-p]$. A *run* in a string s is an interval $[i..j]$ with $1 \leq i \leq j \leq |s|$ such that:

- the smallest period p of $s[i..j]$ satisfies $2p \leq j - i + 1$.
- the interval can be extended neither to the left nor the right, without violating the above condition, that is, $s[i-1] \neq s[i+p-1]$ and $s[j-p+1] \neq s[j+1]$, provided that respective symbols exist.

Lemma 1 (Periodicity Lemma (see [17])) *Let p and q be two periods of a string x. If $p + q - \gcd(p, q) \leq |x|$, then $\gcd(p, q)$ is also a period of x.*

Lemma 2 ([17]) *The periods of any $x \in \Sigma^+$ are partitioned into $O(\log |x|)$-arithmetic progressions.*

2.1.3 Factorization

A string sequence $Fac(s) = f_1, \ldots, f_k$ is called a *factorization* of a string s if $f_1 \cdots f_k = s$ holds. Each string of a factorization is called a *factor*. Let $|Fac(s)|$ be the size of the factorization of s and $Fac(s)[i] = f_i$ for $1 \leq i \leq k$.

LZ77 Factorization

We already described the definition of the LZ77 Factorization without self-reference in Definition 1. We also define the *LZ77 with self-reference* as follows [65].

Definition 3 *For a string s, $LZ77_w(s)$ represents the factorization $f_1, \ldots, f_z$ of s such that for $1 \leq i \leq z$, if $s[|f_1 \cdots f_{i-1}| + 1]$ is a character not occurring $f_0 \cdots f_{i-1}$ then $f_i = s[|f_1 \cdots f_{i-1}| + 1]$, otherwise f_i is the longest prefix of $f_i \cdots f_z$ such that f_i is a substring of $f_1 \cdots f_i$, where $f_0 = \epsilon$.*

Example 2 (LZ77 Factorization) *Let $s = abababcababababcababababcd$. Then $LZ77_{wo}(s) = a, b, ab, ab, c, abababc, abababc, d$ and $LZ77_w(s) = a, b, abab, c, abababcababababc, d$.*

Run Length Encoding

For a string s, let $RLE(s)$ denote the factorization of s such that each factor is a maximal run of same characters a as a^k, where k is the length of the run. $RLE(s)$ can be computed in $O(|s|)$ time.

For a string p, let $\hat{L}(p)$ (resp. $\hat{R}(p)$) denote the first (resp. last) maximal run of p. Then following observation holds.

Observation 1 *Let s and p denote two strings. For every occurrence p in s, the factorization of $p[|\hat{L}(p)|+1..|p|-|\hat{R}(p)|]$ is same in $RLE(s)$, and also, $p[|\hat{L}(p)|+1..|p|-|\hat{R}(p)|] \neq \varepsilon$ when $|RLE(p)| \geq 3$.*

See also Example 3.

Example 3 ($RLE(s)$) *Let $s = aabbbbbabb$. Then $RLE(s) = a^2, b^5, a^1, b^2$, $|RLE(s)| = 4$, $RLE(s)[2] = b^5$ $\hat{L}(s) = a^2$ and $\hat{R}(s) = b^2$*

Locally Consistent Factorization (Parsing)

For integer sequence p of length at least 2, we say that p is an *M-colored sequence* if (1)$p[i] \neq p[i+1]$ holds for any $1 \leq i < |p|$, (2)$0 \leq p[j] \leq M$ holds for any $1 \leq j \leq |p|$, and (3)let $p[i] = 0$ for $i < 1$ or $i > n$. Let $\Gamma : [0..M]^{\Delta_L+\Delta_R+1} \to \{0,1\}$ be a function, where M is a positive integer, $\Delta_L = \log^* M + 6$ and $\Delta_R = 4$. *Locally consistent factorization* $LC_\Gamma(p)$ is a factorization of p using an M-function Γ, which is defined as follows.

Definition 4 ([44]) *We say that Γ is an M-function if $d_\Gamma(p)[1] = 1$, $d_\Gamma(p)[|p|] = 0$ and $Occ(11, d_\Gamma(p)) = Occ(0000, d_\Gamma(p)) = \phi$ hold for any M-colored sequence p, where $d_\Gamma(p) = \Gamma(p[1-\Delta_L], \ldots, p[1+\Delta_R]), \ldots, \Gamma(p[|p|-\Delta_L], \ldots, p[|p|+\Delta_R])$.*

Lemma 3 ([3]) *By computing a table of $o(\log M)$ space in $o(\log M)$ preprocessing time for an M-function Γ, we can compute $d_\Gamma(p)$ in $O(|p|)$ time using the table for a given M-colored sequence p.*

Proof. Here we give only an intuitive description of a proof of Lemma 3. More detailed proofs can be found at [44] and [3].

Mehlhorn et al. [44] showed that there exists a function Γ' which returns a $(\log M)$-colored sequence p' for a given M-colored sequence p in $O(|p|)$ time, where $p'[i]$ is determined only by $p[i-1]$ and $p[i]$ for $1 \leq i \leq |p|$. Let $p^{\langle k \rangle}$ denote the outputs after applying

Γ' to p by k times. They also showed that there exists a function Γ'' which returns a bit sequence d satisfying the conditions of Lemma 3 for a 6-colored sequence p in $O(|p|)$ time, where $d_\Gamma(p)[i]$ is determined only by $p[i-3..i+3]$ for $1 \leq i \leq |p|$. Hence we can compute $d_\Gamma(p)$ for an M-colored sequence p in $O(|p| \log^* M)$ time by applying Γ'' to $p^{\langle \log^* M+2 \rangle}$ after computing $p^{\langle \log^* M+2 \rangle}$. Furthermore, Alstrup et al. [3] showed that $d_\Gamma(p)$ can be computed in $O(|p|)$ time using a precomputed table of size $o(\log M)$. The idea is that $p^{\langle 3 \rangle}$ is a $\log\log\log M$-colored sequence and the number of all combinations of a $\log\log\log M$-colored sequence of length $\log^* M + 11$ is $2^{(\log^* M+11)\log\log\log M} = o(\log M)$. Hence we can compute $d_\Gamma(p)$ for an M-colored sequence in linear time using a precomputed table of size $o(\log M)$. □

For an M-colored sequence p and M-function Γ, we define $LC_\Gamma(p) = f_1, \ldots, f_k$ such that $f_x = p[1_i..1_{i+1} - 1]$ for $1 \leq x \leq k$, where 1_i is i-th occurrence position of 1 in $d_\Gamma(p)$ for $1 \leq i \leq k$ and $1_{k+1} = |d_\Gamma(p)| + 1$, and k is the number of 1 in $d_\Gamma(p)$. See also Example 4.

Each of $f_1, \ldots, f_k$ is called block (or factor) of p. Note that the length of each block is from two to four by the property of $d_\Gamma(p)$, i.e., $2 \leq |f_x| \leq 4$ for any $1 \leq x \leq k$. In this thesis, we omit Γ and write $LC(p)$ when it is clear from the context.

Next, we describe the locally consistent factorization version of Observation 1. For an M-colored sequence p and M-function Γ, Let $C_\Gamma(p)$ is the longest substring $f_i \cdots f_j$ of p such that factors $f_i, \ldots, f_j$ are determined only by a substring of p, where $LC_\Gamma(p) = f_1, \ldots, f_k$. Namely, $C_\Gamma(p)$ does not depend on $p[x]$ for any integer $x < 1$ or $x > |p|$, and also, $L_\Gamma(p) = f_1 \cdots f_{i-1}$ and $R_\Gamma(p) = f_{j+1} \cdots f_k$. Then following observation holds.

Observation 2 *Let s and p denote two M-colored sequences, and let Γ denote an M-function. For every occurrence p in s, the factorization of $C_\Gamma(p)$ is same in $LC_\Gamma(s)$, and also $C_\Gamma(p) \neq \varepsilon$ when $|p| \geq \delta_C$, where $\delta_C = \Delta_L + \Delta_R + 8$.*

Proof. We show that $C_\Gamma(p) \neq \varepsilon$ when $|p| \geq \delta_C$. Consider a case such that $C_\Gamma(p) \neq \varepsilon$. Then we can represent $C_\Gamma(p)$ by $p[1_{i'}..1_{j'} - 1]$, where i' is the minimal integer such that $1_{i'} - \Delta_L \geq 1$ and j' is the maximal integer such that $1_{j'} + \Delta_R \leq |p|$. Hence $|L_\Gamma(p)| \geq \Delta_L$ and $|R_\Gamma(p)| \geq \Delta_R + 1$ hold. Next, $|L_\Gamma(p)| \leq \Delta_L + |000|$ and $|R_\Gamma(p)| \leq \Delta_R + 1 + |000|$ holds by $Occ(0000, d_\Gamma(p)) = \phi$. Hence $C_\Gamma(p) = \varepsilon$ may hold when $|p| \leq \Delta_L + \Delta_R + 7$. Therefore $C_\Gamma(p) \neq \varepsilon$ always holds when $|p| \geq \Delta_L + \Delta_R + 8$. See also Figure 2.1. □

Example 4 (LC) *Let* $\log^* M = 2$, $p = 1,2,3,2,5,7,6,4,3,4,3,4,1,2,3,4,5$ *and* $d_\Gamma(p) = 1,0,0,1,0,1,0,0,1,0,0,0,1,0,1,0,0$. *Then* $LC_\Gamma(p) = (1,2,3),(2,5),(7,6,4),(3,4,3,4),(1,2),(3,4,5)$, $|LC_\Gamma(p)| = 6$, $LC_\Gamma(p)[2] = (2,5)$, $L_\Gamma(p) = 1,2,3,2,5,7,6,4$, $C_\Gamma(p) = 3,4,3,4$ *and* $R_\Gamma(p) = 1,2,3,4,5$. *Note that* $\Delta_L = 8, \Delta_R = 4$.

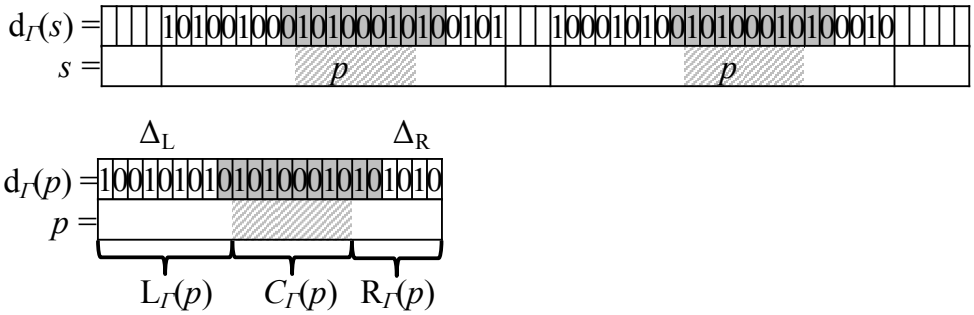

Figure 2.1: Let $\Delta_L = 8$ and $\Delta_R = 4$. This illustrates $d_\Gamma(p), d_\Gamma(s), s$ and p. $Occ(p,s) = \{5,31\}$. $d_\Gamma(p)[1+\Delta_L..|p|-\Delta_R] = d_\Gamma(s)[5+\Delta_L..5+|p|-\Delta_R] = d_\Gamma(s)[23+\Delta_L..23+|p|-\Delta_R]$ by Γ because $d_\Gamma(p)[1+\Delta_L..|p|-\Delta_R]$ is determined by $p[1..|p|]$.

2.2 Order Sets

2.2.1 Predecessor/Successor

Let $\mathcal{S}$ be an integer set whose integer has an integer as value. We consider the following operations for $\mathcal{S}$:

- $pred(\mathcal{S}, a)$: return $a_{pred} = \max\{x \in \mathcal{S} \mid x < v\}$ and a_{pred}'s value.
- $succ(\mathcal{S}, a)$: return $a_{succ} = \min\{x \in \mathcal{S} \mid v < x\}$ and a_{succ}'s value.
- $insert(\mathcal{S}, a, b)$: set $\mathcal{S} \leftarrow \mathcal{S} \bigcup \{a\}$ and b as a's value.
- $delete(\mathcal{S}, a)$: set $\mathcal{S} \leftarrow \mathcal{S} \setminus \{a\}$.
- $member(\mathcal{S}, a)$: return a's value if $a \in \mathcal{S}$.

We say that a data structure is a *dynamic predecessor/successor data structure* if the data structure supports all above operations. If a data structure supports *insert*, *delete*

and *member*, we say that the data structure is a *dynamic membership data structure*. Let $f(n, N)$ be the time for computing all operations for $\mathcal{S}$, where $n = |\mathcal{S}|$ and $\mathcal{S} \subseteq [0..N-1]$. Similarly, $\mu(n, N)$ be the time for computing *insert*, *delete* and *member* operations for $\mathcal{S}$. Beame and Fichs dynamic predecessor/successor data structure [6] of $O(n)$ space for $\mathcal{S}$ can support all operations in $f(n, N) = O(\min\{\frac{\log\log N \log\log n}{\log\log\log N}, \sqrt{\frac{\log n}{\log\log n}}\})$ time. This is the best known dynamic predecessor/successor data structure.

2.2.2 Order Maintenance

We consider the following operations for a list L.

- $insert(x, y)$: insert an element y after x in L.
- $delete(x)$: delete x from L.
- $order(x, y)$: check if x is before y in L.

We say that a data structure is an *order maintenance data structure* if the data structure supports all above operations. Diez and Sleators order maintenance data structure [18] can support every operation in constant time and use $O(|L|)$ space.

2.2.3 Tools on Grids

Range Reporting Query

Let $\mathcal{X}$ and $\mathcal{Y}$ denote subsets of two ordered sets, and let $p \in \mathcal{X} \times \mathcal{Y}$ be a point on the two-dimensional plane. Sometimes we abbreviate it as 2D point p. We consider following operations for a set of 2D points $\mathcal{R} \in \mathcal{X} \times \mathcal{Y}$, where $|\mathcal{X}|, |\mathcal{Y}| \in O(|\mathcal{R}|)$.

- $insert_{\mathcal{R}}(p, x_{pred}, y_{pred})$: given a point $p = (x, y)$, $x_{pred} = \max\{x' \in \mathcal{X} \mid x' \leq x\}$ and $y_{pred} = \max\{y' \in \mathcal{Y} \mid y' \leq y\}$, insert p to $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.
- $delete_{\mathcal{R}}(p)$: given a point $p = (x, y) \in \mathcal{R}$, delete p from $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.
- $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$: given a rectangle (x_1, x_2, y_1, y_2) with $x_1, x_2 \in \mathcal{X}$ and $y_1, y_2 \in \mathcal{Y}$, returns $\{(x, y) \in \mathcal{R} \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$.

Then, we say that a data structure is a *static 2D range reporting data structure* if the data structure supports $report_{\mathcal{R}}$ query for $\mathcal{R}$, and also, we say that the data structure is a *dynamic 2D range reporting data structure* if the data structure also supports $insert_{\mathcal{R}}$ and $delete_{\mathcal{R}}$. Static or dynamic 2D range reporting data structures are widely studied in computational geometry. In this thesis, we use the following dynamic 2D range reporting data structure.

Lemma 4 ([13]) *There exists a dynamic 2D range reporting data structure which supports* $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$ *in* $O(\log|\mathcal{R}| + occ(\log|\mathcal{R}|/\log\log|\mathcal{R}|))$ *time, and* $insert_{\mathcal{R}}(p, i, j)$, $delete_{\mathcal{R}}(p)$ *in amortized* $O(\log|\mathcal{R}|)$ *time, where occ is the number of the elements to output. This structure uses* $O(|\mathcal{R}|)$ *space.* [1]

Range Minimum Query

We say that a 2D point p is a weighed 2D point if p is labeled for an integer, namely $p \in \mathcal{X} \times \mathcal{Y} \times \mathcal{N}$. Let $\mathcal{R} \in \mathcal{X} \times \mathcal{Y} \times \mathcal{N}$ be a set of weighted 2D points, where $|\mathcal{X}|, |\mathcal{Y}| \in O(|\mathcal{R}|)$. A $rmq_{\mathcal{R}}(x_1, x_2, y_1, y_2)$ query returns $\min\{d \mid (x, y, d) \in report_{\mathcal{R}}(x_1, x_2, y_1, y_2)\}$. The following lemma supports *rmq* queries.

Lemma 5 ([1]) *Consider n weighted 2D points* $\mathcal{R}$ *on a two-dimensional plane. There exists a data structure which supports a* $rmq_{\mathcal{R}}$ *queries in* $O(\log^2 n)$ *time, occupies* $O(n)$ *space, and requires* $O(n \log n)$ *time to construct.*

2.3 Automata

2.3.1 Aho-Corasick(AC) Automata

The Aho-Corasick automaton (AC automaton for short) [2] is a finite state machine which simultaneously recognizes all occurrences of multiple patterns in a single pass through a text. The AC automaton for a dictionary Π consists of three functions: *goto*, *failure*, and *output*. Figure 2.2 displays an example of the AC automata.

[1] The original problem considers a real plane in the paper [13], however, his solution only need to compare any two elements in $\mathcal{R}$ in constant time. Hence his solution can apply to our range reporting problem by maintains $\mathcal{X}$ and $\mathcal{Y}$ using the data structure of order maintenance problem proposed by Dietz and Sleator [18], which enables us to compare any two elements in a list L and insert/delete an element to/from L in constant time.

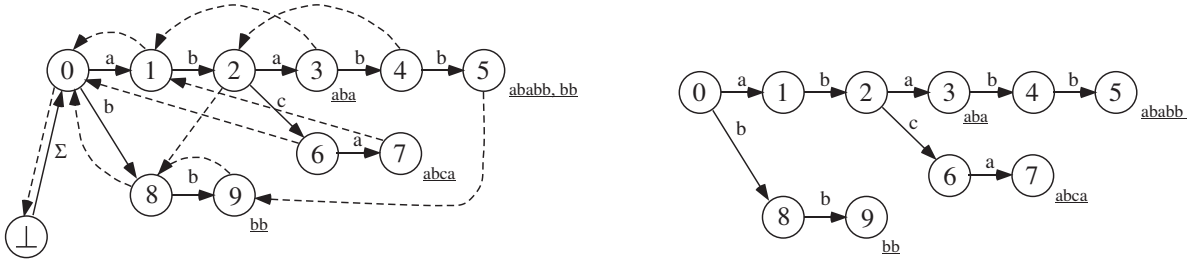

Figure 2.2: On the left the Aho-Corasick automaton for $\Pi = \{\texttt{aba}, \texttt{ababb}, \texttt{abca}, \texttt{bb}\}$ is displayed, where the circles denote states, the solid and the broken arrows represent the goto and the failure functions, respectively, and the underlined strings adjacent to states mean the outputs from them. On the right the g-trie for Π is shown.

The *g-trie* for a dictionary Π is a trie representing Π. There is a natural one-to-one correspondence between the states (nodes) of the g-trie and the pattern prefixes. State q is said to *represent* string u if the path from the initial state 0 to q spells out u. For example, the initial state 0 represents the empty string ε and the state 4 represents the string `abab` in Figure 2.2. Let Q denote the set of states of the g-trie, and let $\perp$ be an auxiliary state not in Q. The g-trie defines the goto function g so that every edge q to r labeled c implies $g(q, c) = r$. In addition, we set $g(\perp, a) = 0$ for all $a \in \Sigma$.

The output function λ and the failure function f are defined as follows.

Definition 5 *Let q be any state. Suppose q represents string u. Then $\lambda(q)$ is the set of patterns in Π that are suffixes of u.*

Definition 6 *Let q be any state with $q \neq 0$. Suppose q represents string u. Then state $f(q)$ represents the longest proper suffix of u that is also a prefix of some pattern.*

Let $\delta : Q \times \Sigma \to Q$ be the state-transition function defined by:

$$\delta(q, a) = \begin{cases} g(q, a), & \text{if } g(q, a) \text{ is defined;} \\ \delta(f(q), a), & \text{otherwise.} \end{cases}$$

We extend δ to the domain $Q \times \Sigma^*$ in the standard way. Then we have:

Lemma 6 ([2]) *For any string $w \in \Sigma^*$, $\delta(0, w)$ is the state that represents the longest suffix of w that is also a prefix of some pattern. The number of goto and failure transitions required in computing $\delta(0, w)$ is at most $2|w|$.*

We say that a state is *branching* if it is of out-degree ≥ 2, and *terminating* if it represents some pattern. We say that a state is *explicit* if it is branching or terminating, and *implicit* otherwise.

Lemma 7 *The number of explicit states is at most* $2|\Pi|$.

2.3.2 Morris-Pratt(MP) Automata

The Morris-Pratt Automaton(MP automaton for short) [46] for a string T is a finite state machine which recognizes all occurrences of T in a single pass through a text.

The set of states is $Q = \{0, 1, \ldots, N\}$ and N is the (unique) accepting state. The MP automaton for T consists of two functions: *goto*, *failure*.

The goto function $g : Q \times \Sigma \to Q \cup \{fail\}$ is defined by:

$$g(q, a) = \begin{cases} q + 1, & \text{if } q \neq N \text{ and } T[q + 1] = a; \\ fail, & \text{otherwise.} \end{cases}$$

For the sake of convenience, an auxiliary state $\perp$ is also introduced such that $g(\perp, a) = 0$ for any $a \in \Sigma$. The failure function $f : Q \to Q \cup \{\perp\}$ is then defined by:

> $f(0) = \perp$ and for any $q \in Q$ with $q \neq 0$, $T[1..f(q)]$ is the longest prefix of T that is also a proper suffix of $T[1..q]$.

Note that the MP Automaton for a pattern $T \in \Sigma^*$ is identical to the AC automaton for a dictionary $\Pi = \{T\}$.

2.4 Context Free Grammars

2.4.1 Straight-Line Programs

A *straight-line program* (*SLP*) is a context free grammar in the Chomsky normal form that generates a single string. Formally, an SLP that generates T is a quadruple $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, such that Σ is an ordered alphabet of terminal characters; $\mathcal{V} = \{X_1, \ldots, X_n\}$ is a set of positive integers, called *variables*; $\mathcal{D} = \{X_i \to expr_i\}_{i=1}^{n}$ is a set of *deterministic productions* (or *assignments*) with each $expr_i$ being either of form $X_\ell X_r$ $(1 \leq \ell, r < i)$, or a single character $a \in \Sigma$; and $S = X_n \in \mathcal{V}$ is the start symbol which derives the string T. We also assume that the grammar neither contains *redundant* variables (i.e., there is

at most one assignment whose righthand side is *expr*) nor *useless* variables (i.e., every variable appears at least once in the derivation tree of $\mathcal{S}$). The *size* of $\mathcal{S}$ is the number n of productions in $\mathcal{D}$. In the extreme cases the length N of the string T can be as large as 2^{n-1}, however, it is always the case that $n \geq \log_2 N$. See also Example 5.

Notation for SLPs

For an $X \in \mathcal{V}$, $val(X)$ represents the derived string by X, $height(X)$ represents the height of the derivation tree of X, $|X|$ represents $|val(X)|$, and $X[i] = val(X)[i]$ for $1 \leq i \leq |X|$. For any variable sequence $y \in \mathcal{V}^+$, let $val^+(y) = val(y[1]) \cdots val(y[|y|])$. For any variable X_i with $X_i \to X_\ell X_r \in \mathcal{D}$, let $X_i.\text{left} = val(X_\ell)$ and $X_i.\text{right} = val(X_r)$, which are called the *left string* and the *right string* of X_i, respectively. For two variables $X_i, X_j \in \mathcal{V}$, we say that X_i occurs at position c in X_j if there is a node labeled with X_i in the derivation tree of X_j and the leftmost leaf of the subtree rooted at that node labeled with X_i is the c-th leaf in the derivation tree of X_j. We define the function $vOcc(X_i, X_j)$ which returns all positions of X_i in the derivation tree of X_j.

Let $Assgn_{\mathcal{S}}$ be the function such that $Assgn_{\mathcal{S}}(expr_i) = X_i$ if $X_i \to expr_i \in \mathcal{D}$, otherwise it returns NIL. When clear from the context, we write $Assgn_{\mathcal{S}}$ as $Assgn$. For any variable sequence $x \in \mathcal{V}^+$, let $Pair(x) = x$ if $|x| = 1$, otherwise $Pair(x) = Pair(Assgn(x[1]x[2])x[3..])$. Let $Assgn^+(f_1, \ldots, f_k) = Assgn(f_1), \ldots, Assgn(f_k)$, and $Pair^+(f_1, \ldots, f_k) = Pair(f_1), \ldots, Pair(f_k)$ for a sequence $f_1, \ldots, f_k$.

Example 5 (SLP) *Let $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be the SLP such that $\Sigma = \{A, B, C\}$, $\mathcal{V} = \{X_1, \cdots, X_{11}\}$, $\mathcal{D} = \{X_1 \to A, X_2 \to B, X_3 \to C, X_4 \to X_3X_1, X_5 \to X_4X_2, X_6 \to X_5X_5, X_7 \to X_2X_3, X_8 \to X_1X_2, X_9 \to X_7X_8, X_{10} \to X_6X_9, X_{11} \to X_{10}X_6\}$, $S = X_{11}$, the derivation tree of S represents $CABCABBCABCABCAB$.*

Properties and Tools

Lemma 8 ([45]) *We can pre-process an SLP $\mathcal{S}$ of size n in $O(n^3)$ time and $O(n^2)$ space to answer the following query in $O(n^2)$ time: given two variables X_i and X_j ($1 \leq i, j \leq n$), compute the length of the longest common prefix of $val(X_i)$ and $val(X_j)$.*

For each variable X_i we store the length $|X_i|$ of the string derived by X_i, which can be computed in a total of $O(n)$ time using $O(n)$ space by a simple dynamic programming algorithm.

The *sorted index* of an SLP $\mathcal{S}$ of size n is the permutation σ of $[1..n]$ such that the strings $val(X_{\sigma(1)}), \ldots, val(X_{\sigma(n)})$ are arranged in the lexicographical order.

Lemma 9 *The sorted index σ of an SLP of size n can be computed in $O(n^3 \log n)$ time using $O(n^2)$ space.*

Proof. We compute the length ℓ of the longest common prefix of two variables X_i and X_j using Lemma 8. Then, comparing $val(X_i)$ and $val(X_j)$ reduces to comparing the $(\ell+1)$-th leaves of the derivation trees of X_i and X_j, which can be done in $O(n)$ time using the length of the string that each variable derives (note that the case where $\ell = min\{|X_i|, |X_j|\}$ is easier). Hence the sorted index σ can be computed in $O(n^3 + n^3 \log n) = O(n^3 \log n)$ time using any $O(n \log n)$-time comparison sort. □

Lemma 10 ([12]) *Given an SLP $\mathcal{S}$ of size n that represents a string T of length N, it is possible to pre-process $\mathcal{S}$ in $O(n)$ time using $O(n)$ space, so that any substring $T[i..i+m-1]$ of length m of T can be computed in $O(\log N + m)$ time.*

Stabbing Variables

An interval pair $([x..y], [y+1..z])$ is said to *stab* an interval $[b..e] \subseteq [x..z]$ if $b \in [x..y]$ and $e \in [y+1..z]$. A variable $X \in \mathcal{V}$ is said to stab an interval $[b..e] \subseteq [1..|X|]$ if $([1..|X.\text{left}|], [|X.\text{left}|+1..|X|])$ stabs $[b..e]$. For any string $P \in \Sigma^+$, let $Occ(P, X)$ denote $Occ(P, val(X))$, and let $Occ^{\xi}(P, X)$ be the set of positions $\alpha \in Occ(P, X)$ such that the interval $[\alpha..\alpha + |P| - 1]$ is stabbed by X.

Lemma 11 ([45]) *$Occ^{\xi}(P, X)$ forms an arithmetic progression.*

We say that X is j-stabbing variable of P if $|X.\text{left}| - j + 1 \in Occ^{\xi}(P, X)$ holds. Let $pOcc_{\mathcal{S}}(P, j)$ be the set of j-stabbing variables of P in $\mathcal{S}$. Since an occurrence of P (of length at least 2) is stabbed by a variable in the derivation tree of $\mathcal{S}$, the following observation holds.

Observation 3 (e.g. [14]) *For an SLP $\mathcal{S}$ of size n which represents a string T and a string P of length at least 2, $Occ(P, T) \Leftrightarrow \bigcup_{1 \le j < |P|} \{j + k - 1 \mid X \in pOcc_{\mathcal{S}}(P, j), k \in vOcc(X, S)\}$ holds, where S is the start variable of $\mathcal{S}$.*

See also Example 6.

Example 6 (Stabbing Variables) *Let $\mathcal{S}$ be the SLP of Example 5. Given a pattern $P = BCAB$, then P occurs at 3, 7, 10 and 13 in the string T represented by SLP $\mathcal{S}$. Hence $Occ(P,T) = \{3,7,10,13\}$. On the other hand, P occurs at 3 in $val(X_6)$ and P is divided by X_5 and X_5, where $X_6 \to X_5X_5$. Similarly, divided P occurs at 1 in $val(X_9)$, at 10 in $val(X_{11})$. Hence $pOcc_{\mathcal{S}}(P) = \{(X_6,3),(X_{11},10),(X_9,1)\}$. Specifically $pOcc_{\mathcal{S}}(P,1) = \{(X_6,3),(X_{11},10)\}$, $pOcc_{\mathcal{S}}(P,2) = \{(X_9,1)\}$ and $pOcc_{\mathcal{S}}(P,3) = \phi$. Hence we can also compute $Occ(P,T) = \{3,7,10,13\}$ by $vOcc(X_6,S) = \{1,11\}$, $vOcc(X_9,S) = \{7\}$, and $vOcc(X_{11},S) = \{1\}$. See also Fig. 2.3.*

Range Reporting and Stabbing Variables

For an SLP $\mathcal{S}$, let $\mathcal{R}_{\mathcal{S}} = \{(X.\text{left}^R, X.\text{right}) \mid X \in \mathcal{V}\}$. Then, we can regard each variable of $\mathcal{S}$ as a 2D point on the two-dimensional plane defined by $\mathcal{X}_{\mathcal{S}} = \{X.\text{left}^R \mid X \in \mathcal{V}\}$ and $\mathcal{Y}_{\mathcal{S}} = \{X.\text{right} \mid X \in \mathcal{V}\}$, where elements in $\mathcal{X}_{\mathcal{S}}$ and $\mathcal{Y}_{\mathcal{S}}$ are sorted by lexicographic order.

For a string P, let y^P_{prec} (resp. y^P_{succ}) denote the lexicographically smallest (resp. largest) element in $\mathcal{Y}_{\mathcal{S}}$ that has P as a prefix. Similarly, let x^P_{prev} (resp. x^P_{succ}) denote the lexicographically smallest (resp. largest) element in $\mathcal{X}_{\mathcal{S}}$ that has P^R as a prefix. Then the following observation holds.

Observation 4 (e.g. [14]) *For any string $P \in \Sigma^*$ of length at least 2 and an integer $1 \le i < |P|$, the following equation holds.*

$$pOcc_{\mathcal{S}}(P,i) \Leftrightarrow report_{\mathcal{R}_{\mathcal{S}}}(x^{P[..i]}_{prec}, x^{P[..i]}_{succ}, y^{P[i+1..]}_{prec}, y^{P[i+1..]}_{succ})$$

See also Example 7.

Example 7 (SLP) *Let $\mathcal{S}$ be the SLP of Example 5. Then,*

$$\begin{aligned} \mathcal{X}_{\mathcal{S}} &= \{x_1, x_4, x_2, x_8, x_5, x_9, x_6, x_{10}, x_{11}, x_3, x_7\}, \\ \mathcal{Y}_{\mathcal{S}} &= \{y_1, y_8, y_2, y_7, y_9, y_3, y_4, y_5, y_6, y_{10}, y_{11}\}, \end{aligned}$$

$x_i = val(X_i)^R$, $y_i = val(X_i)$ for any $X_i \in \mathcal{V}$. See also Fig. 2.3.

2.4.2 Dictionary SLP(DSLP)

We already defined the DSLP in Definition 2.

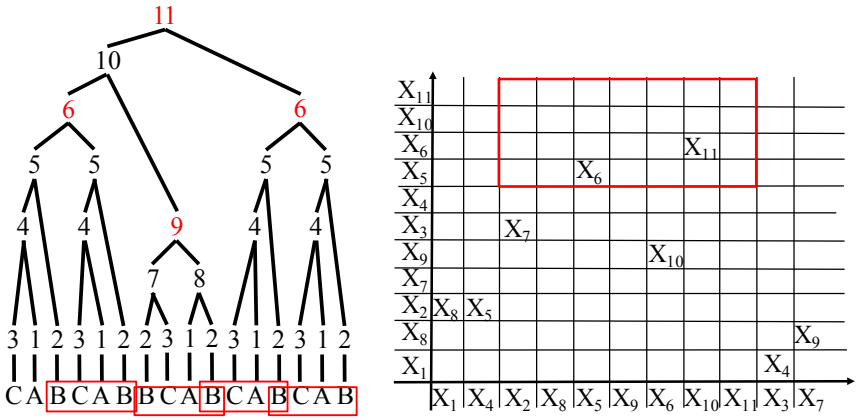

Figure 2.3: The left figure is the derivation tree of SLP $\mathcal{S}$ of Example 5, which derives the string T. The red rectangles on T represent all occurrences of $P = BCAB$ in T. The right grid represents the relation between $\mathcal{X}_\mathcal{S}$, $\mathcal{Y}_\mathcal{S}$ and $\mathcal{R}_\mathcal{S}$ of Example 7. The red rectangle on the grid is a query rectangle $(x_{prec}^{P[..i]}, x_{succ}^{P[..i]}, y_{prec}^{P[i+1..]}, y_{succ}^{P[i+1..]})$, where $i = 1$, $x_{prec}^{P[..i]} = x_2$, $x_{succ}^{P[..i]} = x_{11}$, $y_{prec}^{P[i+1..]} = y_5$ and $y_{succ}^{P[i+1..]} = y_{11}$. Therefore, $report_{\mathcal{R}_\mathcal{S}}(x_{prec}^{P[..i]}, x_{succ}^{P[..i]}, y_{prec}^{P[i+1..]}, y_{succ}^{P[i+1..]}) = \{X_6, X_{11}\}$.

2.4.3 Repetitive Straight-Line Programs

We define *repetitive SLPs* (*RSLPs*), as an extension to SLPs, which allow *run-length encodings* in the righthand sides of productions, i.e., $\mathcal{D}$ might contain a production $X \to \hat{X}^k \in \mathcal{V} \times \mathcal{N}$. The *size* of the RSLP is still the number of productions in $\mathcal{D}$ as each production can be encoded in constant space.

We define the left and right strings for any variable $X_i \to X_\ell X_r \in \mathcal{D}$ in a similar way to SLPs. Furthermore, for any $X \to \hat{X}^k \in \mathcal{D}$, let $X.\mathrm{left} = val(\hat{X})$ and $X.\mathrm{right} = val(\hat{X})^{k-1}$, and also, X is (j, x)-stabbing variable of P if $p \in Occ(P, X)$ holds and $[p..p + |P| - 1]$ is stabbed by $([1..|\hat{X}^x|], [\hat{X}^x + 1..|\hat{X}^k|])$, where $p = |\hat{X}^x| - j - 1$. Note that X is j-stabbing variable of P if and only if X is $(j, 1)$-stabbing variable of P. Since $X \to \hat{X}^k$ is a run of $\hat{X}$, the following observation holds.

Observation 5 *For a production $X \to \hat{X}^k$ in $\mathcal{D}$, if there exists a string P and two integers j, x such that X is a (j, x)-stabbing variable of P and $x > 1$, then X is a $(j, x-1)$-stabbing variable of P.*

We can show the RSLP version of Observation 3 by Observations 5 and 6.

Observation 6 *For a string P of length at least 2, an occurrence P is j or (j, x)-stabbed by a variable in the derivation tree of $\mathcal{S}$, where j and $x > 1$ are positive integers.*

Example 8 (RSLP) *Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be an RSLP, where $\Sigma = \{A, B\}$, $\mathcal{V} = \{1, \ldots, 24\}$, $\mathcal{D} = \{1 \to A, 2 \to B, 3 \to 1^1, 4 \to 2^1, 5 \to 2^2, 6 \to 1^2, 7 \to (3, 4), 8 \to (3, 5), 9 \to (8, 3), 10 \to (4, 3), 11 \to (10, 4), 12 \to (11, 6), 13 \to 7^3, 14 \to 9^1, 15 \to 10^1, 16 \to 12^1, 17 \to 10^3, 18 \to (13, 14), 19 \to (18, 15), 20 \to (16, 17), 21 \to 19^1, 22 \to 20^1, 23 \to (21, 22), 24 \to 23^1\}$, and $S = 24$. The derivation tree of the start symbol S represents a single string $T = ABABABABBABABABAABABABA$.*

Representation of RSLPs

For an RSLP $\mathcal{G}$ of size w, we can consider a DAG of size w as a compact representation of the derivation trees of variables in $\mathcal{G}$. Each node represents a variable X in $\mathcal{V}$ and store $|val(X)|$ and out-going edges represent the assignments in $\mathcal{D}$: For an assignment $X_i \to X_\ell X_r \in \mathcal{D}$, there exist two out-going edges from X_i to its ordered children X_ℓ and X_r; and for $X \to \hat{X}^k \in \mathcal{D}$, there is a single edge from X to $\hat{X}$ with the multiplicative factor k.

2.5 Signature Encoding

A signature encoding [44] of a string T of length N is a RSLP determined by recursively applying LC and RLE to T until a start symbol S is obtained. Formally, we say that a RSLP $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ representing T is a signature encoding of T if $S = id(T)$ holds, where $id(T) = Pow_h^T$, h is the minimum integer satisfying $|Pow_h^T| = 1$, Γ is an M-function, and

$$\begin{aligned} Shrink_t^T &= \begin{cases} Assgn^+(T) & \text{for } t = 0, \\ Pair^+(LC_\Gamma(Pow_{t-1}^T)) & \text{for } 0 < t \leq h, \end{cases} \\ Pow_t^T &= Assgn^+(RLE(Shrink_t^T)) \text{ for } 0 \leq t < h. \end{aligned}$$

We call each variable of the signature encoding a *signature*, and use e (for example, $e_i \to e_\ell e_r \in \mathcal{D}$) instead of X to distinguish from general RSLPs. We say that a node is in *level* t in the derivation tree of S if the node is produced by $Shrink_t^T$ or Pow_t^T. In this thesis, we implement signature encodings by the DAG of RSLP introduced in Chapter 2,

and also, a signature encoding for a dynamic string is called *dynamic signature encoding*. See also Example 9.

Example 9 (Signature encoding) *Let* $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ *be an RSLP of Example 8. Assuming* $LC(Pow_0^T) = (3,4)^3, (3,5,3), (4,3), (4,3,4,6), (4,3)^3$, $LC(Pow_2^T) = (13, 14, 15)$, $(16, 17)$ *and* $LC(Pow_3^T) = (21, 22)$ *hold,* $\mathcal{G}$ *is the signature encoding of* T *and* $id(T) = 24$. *Here,* $Pair((13, 14)) = 18$, $Pair((4, 3, 4, 6)) = 12$, $Pair((4, 5)) = \text{NIL}$. *See Fig. 2.4 for an illustration of the derivation tree of* $\mathcal{G}$ *and the corresponding DAG.*

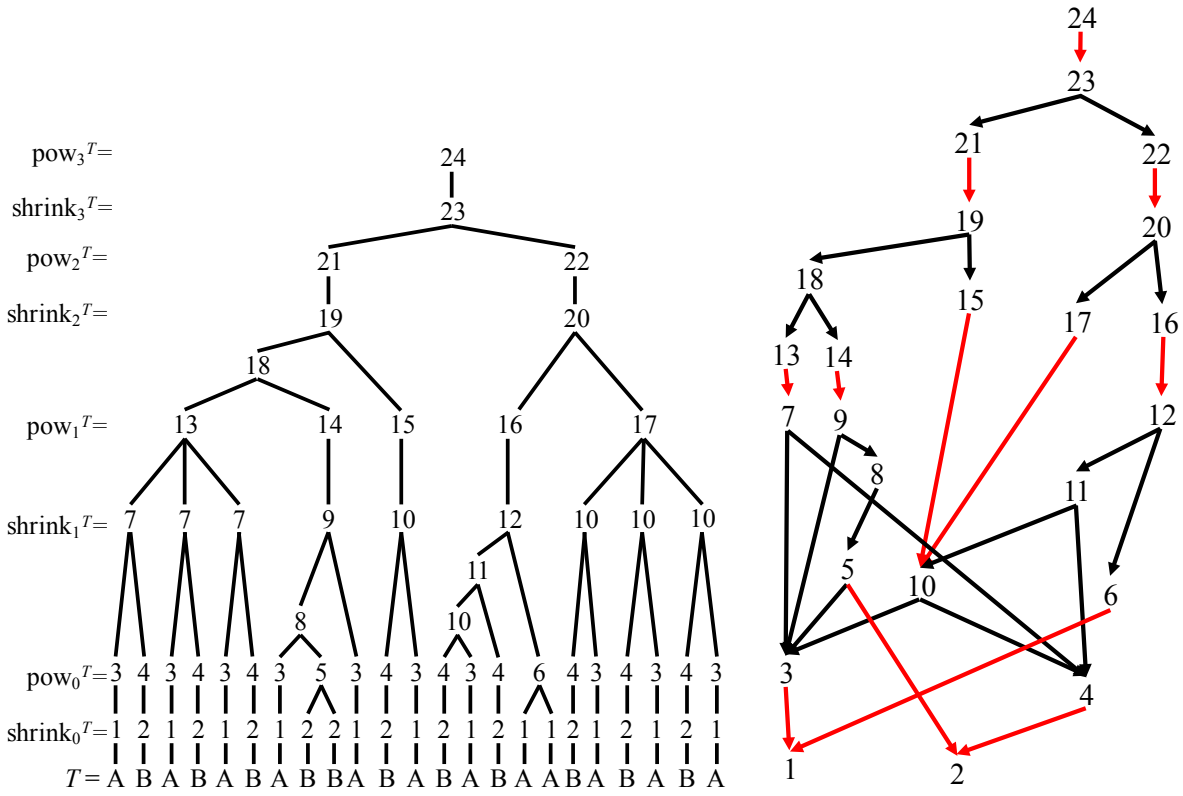

Figure 2.4: The derivation tree of S (left) and the DAG for $\mathcal{G}$ (right) of Example 8. In the DAG, the black and red arrows represent $e \to e_\ell e_r$ and $e \to \hat{e}^k$ respectively. In Example 9, T is encoded by signature encoding.

2.5.1 Properties

In this section, we describe some properties of signature encodings.

Random access under the signature encoding. We show that a signature encoding $\mathcal{G}$ for a string T supports a random access in T.

Lemma 12 *Using a signature encoding $\mathcal{G}$ for a string T, We can compute $T[i..i+\ell-1]$ for a given integers i and ℓ in $O(\ell + \log|T|)$ time.*

Proof. (1) By the property of locally consistent parsing, a signature in $Shrink_t^T$ derive signatures of at least 2 in $Shrink_{t-1}^T$ for $1 \leq t \leq h$. (2) The height of the derivation tree of the signature encoding of T is $O(\log|T|)$ by the fact (1). Hence Lemma 12 clearly holds by the facts (1) and (2). □

Space requirement of the signature encoding. It is clear from the definition of the signature encoding $\mathcal{G}$ of T that the size of $\mathcal{G}$ is less than $4N$, all signatures are in $[0..M]$, Pow_t^T is M-colored sequence for all $0 \leq t < h$, and $\Delta_L = \log^* M + 6$ and $\Delta_R = 4$ in Γ.

Moreover, the next lemma shows that $\mathcal{G}$ requires only *compressed space*:

Lemma 13 ([55]) *The size w of the signature encoding of T of length N is $O(\min(z \log N \log^* M, N))$, where $z = |LZ77_{wo}(T)|$.*

Proof. See the proof of Theorem 1(2).

Common sequences of signatures to all occurrences of same substrings. Here, we explain the most important property of the signature encoding, which ensures the existence of common signatures to all occurrences of same substrings(Lemma 14).

Consider a pattern P which occurs in T. Since each $Shrink_t^T$ and Pow_t^T is factorized by RLE and LC respectively, we can apply Observations 1 and 2 to P recursively. This means that, for every occurrence of P in T, an internal substring of P is represented by a common signature sequence in $Shrink_t^T / Pow_t^T$. See also Figure 2.5(1). Formally, we define such a sequence by $XShr_t^P$ and $XPow_t^P$, which are defined as follows :

Definition 7 *For a string P, let*

$$XShr_t^P = \begin{cases} Assgn^+(P) & \text{for } t = 0, \\ Pair^+(LC_\Gamma(XPow_{t-1}^P)[|L_t^P|..|XPow_{t-1}^P| - |R_t^P|]) & \text{for } 0 < t \leq h^P, \end{cases}$$

$$XPow_t^P = Assgn^+(RLE(XShr_t^P[|\hat{L}_t^P| + 1..|XShr_t^P| - |\hat{R}_t^P|])|) \quad \text{for } 0 \leq t < h^P, \text{ where}$$

$L_t^P = L_\Gamma(XPow_{t-1}^P)$, $R_t^P = R_\Gamma(XPow_{t-1}^P)$, $\hat{L}_t^P = \hat{L}(XShr_t^P)$, $\hat{R}_t^P = \hat{R}(XShr_t^P)$, and h^P is the minimum integer such that $|RLE(XShr_{h^P}^P)| \leq \delta_C$.

Then Observation 1 clearly holds when $p = XShr_t^P$ and $s = Shrink_t^T$. This means that $f_i, \ldots, f_j$ are encoded to $XPow_t^P$. Similarly Observation 2 clearly holds when $p = XPow_t^P$ and $s = Pow_t^T$. This means that $f_i, \ldots, f_j$ are encoded to $XShr_{t+1}^P$. By applying these observations to P recursively, we can represent every occurrence of P in T by $Uniq(P) = \hat{L}_0^P L_0^P \cdots \hat{L}_{h^P}^P L_{h^P}^P XShr_{h^P}^P R_{h^P}^P \hat{R}_{h^P}^P \cdots R_0^P \hat{R}_0^P$. Namely, $val^+(Uniq(P)) = P$. See also Figure 2.5(2). Hence the following lemma holds by $|L_t^P|, |R_t^P|, |RLE(\hat{L}_t^P)|, |RLE(\hat{R}_t^P)|$ and $|RLE(XShr_{h^P}^P)| = O(\log^* M)$.

Lemma 14 (common sequences [55]) *For a string P, every substring P in T is represented by a signature sequence $Uniq(P)$ of run-length $O(\log |P| \log^* M)$ in the derivation tree of $id(T)$* [2].

Hence we call $Uniq(P)$ the *common sequence* of P.

The number of ancestors of nodes corresponding to $Uniq(P)$ is upper bounded by:

Lemma 15 *Let P be a string and $\mathcal{T}$ be the derivation tree of a signature $e \in \mathcal{V}$. Consider an occurrence of P in $val(e)$, and the induced subtree X of $\mathcal{T}$ whose root is the root of $\mathcal{T}$ and whose leaves are the parents of the nodes representing $Uniq(P)$. Then X contains $O(\log^* M)$ nodes for every level and $O(\log |e| + \log |P| \log^* M)$ nodes in total.*

Proof. By Definition 7, for every level, X contains $O(\log^* M)$ nodes that are parents of the nodes representing $Uniq(P)$. Lemma 15 holds because the number of nodes at some level is halved when *Shrink* is applied. More precisely, considering the x nodes of X at some level to which *Shrink* is applied, the number of their parents is at most $(x + 2)/2$. Here the '+2' term reflects the fact that both ends of x nodes may be coupled

[2] The common sequences are conceptually equivalent to the *cores* [41] which are defined for the *edit sensitive parsing* of a text, a kind of locally consistent parsing of the text.

with nodes outside X, and also, since $|RLE(\hat{L}_t^P)| = |RLE(\hat{R}_t^P)| = 1$ for $0 \leq t < h^P$ and $|RLE(XShr_{h^P}^P)| = O(|\log^* M|)$, each nodes representing $\hat{L}_t^P$ and $\hat{R}_t^P$ has a common parent for every level, and the number of parents of nodes representing $XShr_{h^P}^P$ is $O(\log^* M)$. Note that $h = O(\log |e|)$ holds for $e \in \mathcal{V}$ by the signature encoding, where h is the height of derivation tree of e. □

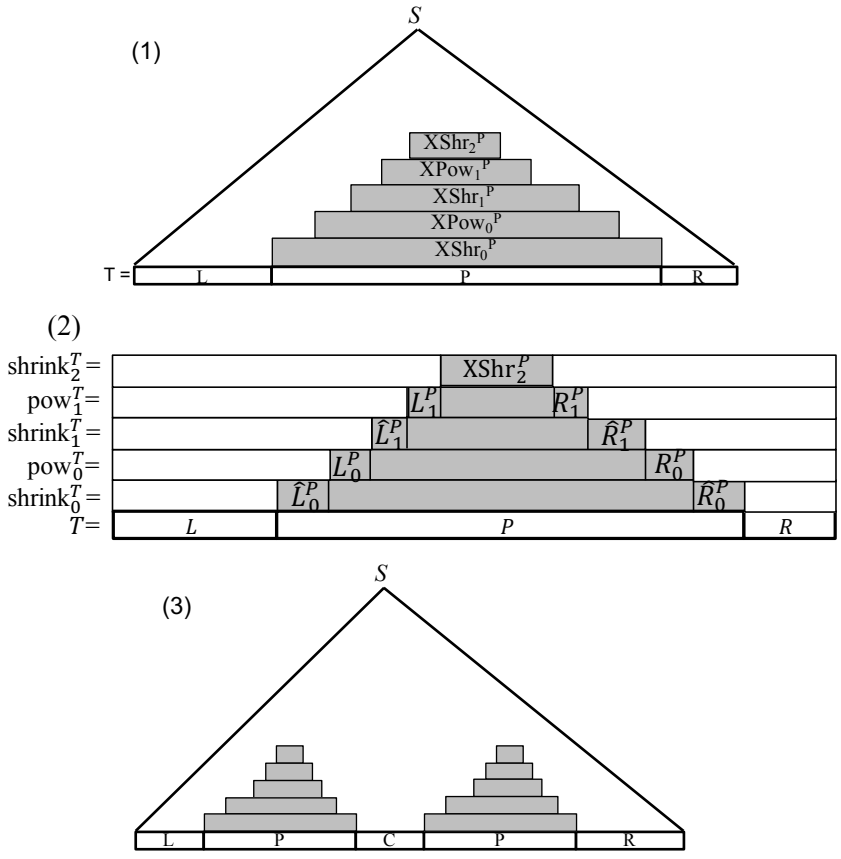

Figure 2.5: Abstract images of consistent signatures of substring P of text T, on the derivation trees of the signature encoding of T. Gray rectangles in Figures (1)-(3) represent common signatures for occurrences of P. (1) Each $XShr_t^P$ and $XPow_t^P$ occur on substring P in $Shrink_t^T$ and Pow_t^T, respectively, where $T = LPR$. (2) The substring P can be represented by $\hat{L}_0^P L_0^P \hat{L}_1^P L_1^P XShrink_2^P R_1^P \hat{R}_1^P R_0^P \hat{R}_0^P$. (3) There exist common signatures on every substring P in the derivation tree.

Lemma 16 shows that we can efficiently compute $Uniq(P)$ for a substring P of T.

Lemma 16 *Using the DAG of $\mathcal{G}$, given a signature $e \in \mathcal{V}$ (and its corresponding node in the DAG) and two integers j and y, we can compute $RLE(Uniq(e[j..j+y-1]))$ in $O(\log |e| + \log y \log^* M)$ time.*

Proof. Let $\mathcal{T}$ be the derivation tree of e and consider the induced subtree X of $\mathcal{T}$ whose root is the root of $\mathcal{T}$ and whose leaves are the parents of the nodes representing $Uniq(e[j..j+y-1])$. Then the size of X is $O(\log |e| + \log y \log^* M)$ by Lemma 15. Starting at the given node in the DAG which corresponds to e, we compute X using Definition 7 and the properties described in the proof of Lemma 15 in $O(\log |e| + \log y \log^* M)$ time. Hence Lemma 16 holds. □

Lemma 17 ([3]) *Let $\mathcal{V}_s$ be the set of signatures which occur in the derivation tree of $id(s) \in \mathcal{V}$ for a string s. Then $|(\mathcal{V}_{s_1} \cup \mathcal{V}_{s_2}) \setminus \mathcal{V}_{s_1 s_2}|, |\mathcal{V}_{s_1 s_2} \setminus (\mathcal{V}_{s_1} \cup \mathcal{V}_{s_2})| = O(\log N' \log^* M)$ holds for two strings s_1 and s_2 such that $id(s_1), id(s_2), id(s_1 s_2) \in \mathcal{V}$, where $N' = |s_1 s_2|$.*

Proof. By Lemma 14, the derivation tree of $id(s_1 s_2)$ is created over $Uniq(s_1)$ $Uniq(s_2)$, and hence, $O(\log N' \log^* M)$ signatures can be added by Lemma 15. Similarly, $O(\log N' \log^* M)$ signatures, which were created over $Uniq(T[..i-1])\,Uniq(T[i..])$, are removed. □

Chapter 3

Construction and Update of Signature Encoding

As described in Chapter 1, signature encodings are used by many applications. In this chapter, we show Theorems 1 and 2, and present a new application of signature encodings.

3.1 Updates

In this section, we show Theorem 1.

Consider the update of a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of a text T. During updates we recompute Shrink_t^T and Pow_t^T for some part of new T. When we need a signature for expr, we look up the signature assigned to expr (i.e., compute $\mathit{Assgn}(\mathit{expr})$) and use it if such exists. If $\mathit{Assgn}(\mathit{expr})$ is NIL, we create a new signature e_{new}, which is an integer that is currently not used as signatures, and add $e_{new} \to \mathit{expr}$ to $\mathcal{D}$. Similarly, updates produce an useless signature, which does not occur in the derivation tree representing new T, i.e., the parents in the DAG are all removed. We remove all useless signatures from $\mathcal{D}$ during updates.

3.1.1 Update Algorithms and Data Structures

Note that we do not need to recompute $\mathit{id}(T)$ from scratch for updating $\mathcal{G}$. Lemma 17 implies that *INSERT* and *DELETE* need to add/remove only $O(\log N \log^* M + y)$ signatures to/from $\mathcal{G}$, because $T' \leftarrow T[..i-1]YT[i..]$ is a concatenation of $T[..i-1]$, Y and $T[i..]$. We can show that these update operations can be computed efficiently using this fact.

We support $DELETE(j, y)$ as follows: (1) Compute the new start variable $S' = id(T[..j-1]T[j+y..])$ by recomputing the new signature encoding from $Uniq(T[..j-1])$ and $Uniq(T[j+y..])$. Although we need a part of $d_\Gamma(Pow_t^{T[..j-1]T[j+y..]})$ to recompute $LC_\Gamma(Pow_t^{T[..j-1]T[j+y..]})$ for every level t, the input size to compute the part of $d_\Gamma(Pow_t^{T[..j-1]T[j+y..]})$ is $O(\log^* M)$ by Lemma 3. Hence these can be done in $O((q_{Assgn} + q_{Add/Remove} + q_{new}) \log N \log^* M)$ time by Lemmas 16 and 17, where q_{Assgn}, $q_{Add/Remove}$ and q_{new} are the times for computing $Assgn(expr)$ for a given $expr$, for adding/removing an assignment to/from $\mathcal{D}$ and for computing e_{new}. (2) Remove all useless signatures Z from $\mathcal{G}$, where $|Z| = O(y + \log N \log^* M)$ by Lemma 15. If a signature is useless, then all the signatures along the path from S to it are also useless. Hence, we can remove all useless signatures efficiently by depth-first search starting from S, which takes $O(q_{Add/Remove}|Z|)$ time.

Similarly, we can support $INSERT(Y, i)$ in $O((q_{Assgn} + q_{Add/Remove} + q_{new})(y + \log N \log^* M))$ time. Note that we can naively compute $Uniq(Y)$ in $O(y(q_{Assgn} + q_{Add/Remove} + q_{new}))$ time. For $INSERT'(j, y, i)$, we can avoid $O(y(q_{Assgn} + q_{Add/Remove} + q_{new}))$ time by computing $Uniq(T[j..j+y-1])$ using Lemma 16.

Next, we describe data structures for updating $\mathcal{G}$. In addition to the DAG for $\mathcal{G}$, we need additional data structures which supports $Assgn(\cdot)$ and e_{new} for dynamic $\mathcal{G}$. For computing $Assgn(\cdot)$, we use a dynamic membership data structure using linear space. For computing e_{new}, we have an integer $i = \max \mathcal{V} + 1$. When we need e_{new}, this data structure returns i as e_{new} and update $i \leftarrow i + 1$. Hence we get $q_{Assgn}, q_{Add/Remove} = O(\mu(w, M))$ and $q_{new} = O(1)$.

Therefore, there exists a data structure of $O(w)$ space which supports $INSERT$ and $DELETE$ in $O(\mu(w, M)(y + \log N \log^* M))$ time and $INSERT'$ in $O(\mu(w, M)y)$ time.

3.1.2 The Data Structure for Bounding New Signatures

In the above update algorithm, the value of e_{new} depends on the number of update operations. This implies that e_{new} will be larger than M. To prevent this, we use the following lemma.

Lemma 18 *Let $\mathcal{G}$ be the signature encoding of size w for a dynamic string of maximal length N_{max}. Using additional $O(w)$ space, we can compute e_{new} such that $e_{new} \leq 4N_{max}$ holds in $O(f(w, 4N_{max}))$ time.*

Proof. Note that $w \leq 4N_{max}$ holds by the definition of signature encodings. Let $\overline{\mathcal{V}}$ be the set of unused signatures in $[1..4N_{max}]$ and we choose $\min \overline{\mathcal{V}}$ as e_{new}. Since $\overline{\mathcal{V}}$ can be represented by $O(w)$ intervals, we can maintain $\overline{\mathcal{V}}$ by a dynamic predecessor/successor data structures of $O(w)$ size. Hence we can compute $\min \overline{\mathcal{V}}$ in $O(f_{\mathcal{A}})$ time. □

Hence $q_{new} = O(f(w, M))$ by Lemma 18 and set $M = 4N_{max}$ if we want to always bound $e_{new} \leq M$. Therefore Theorem 1 holds. Note that $Expr(e)$ can be computed in constant time using the DAG of $\mathcal{G}$.

3.2 Construction

In this section, we give proofs of Theorem 2. Note that we set $M = 4N$ in following proofs, and also, we can replace $f_{\mathcal{A}}$ with $\mu(w, cM)$ in following proofs because the output of Theorem 2 is the static signature encoding of T, where c is a constant positive value such that $c \geq 1$.

3.2.1 Proof of Theorem 2 (2)

Proof. Consider a dynamic signature encoding $\mathcal{G}$ for an empty string. Then Theorem 2 (2) immediately holds by computing $INSERT'(c_i, |f_i|, |f_1 \cdots f_{i-1}|+1)$ for all $1 \leq i \leq z$ incrementally, where $c_i \leq |f_1 \cdots f_{i-1}| - |f_i|$ is a position such that $T[c_i..c_i + |f_i| - 1] = f_i$ holds. Note that when f_i is a character which does not occur in $f_1, \ldots f_{i-1}$ for $1 \leq i \leq z$, we compute $INSERT(f_i, |f_1 \cdots f_{i-1}| + 1)$ in $O(f(w, M) \log N \log^* M)$ time instead of the above $INSERT'$ operation. □

Note that we can directly show Lemma 13 from the above proof because the size of $\mathcal{G}$ increases $O(\log N \log^* M)$ by Lemma 15, every time we do $INSERT'(c_i, |f_i|, |f_1 \cdots f_{i-1}| + 1)$ for $1 \leq i \leq z$.

3.2.2 Proof of Theorem 2 (3a)

Proof. We use the *G-factorization* proposed in [54]. By the G-factorization of T with respect to $\mathcal{S}$, T is partitioned into $O(n)$ strings, each of which, corresponding to $T[i..j]$, is derived by a variable X of $\mathcal{S}$ such that X appears in the derivation tree of $\mathcal{S}$ to derive a substring of $T[1..i-1]$, or otherwise X derives a single character that does not appear

in $T[1..i-1]$. Note that we can compute a sequence of variables of $\mathcal{S}$ corresponding to the G-factorization of T with respect to $\mathcal{S}$ in $O(n)$ time by the depth-first traversal of the DAG of S. Since the G-factorization resembles the LZ77 factorization, we can construct the dynamic signature encoding $\mathcal{G}$ for T by $O(n)$ *INSERT'* and *INSERT* operations as the proof of Theorem 2 (2). □

3.2.3 Proof of Theorem 2 (1a)

Proof. Note that we can naively compute $id(T)$ for a given string T in $O(Nf_{\mathcal{A}})$ time and $O(N)$ working space. In order to reduce the working space, we consider factorizing T into blocks of size b and processing them incrementally: Starting with the empty signature encoding $\mathcal{G}$, we can compute $id(T)$ in $O(\frac{N}{b} f_{\mathcal{A}}(\log N \log^* M + b))$ time and $O(w+b)$ working space by using $INSERT(T[(i-1)b+1..ib], (i-1)b+1)$ for $i = 1, \ldots, \frac{N}{b}$ in increasing order. Hence our proof is finished by choosing $b = \log N \log^* M$. □

3.2.4 Proof of Theorem 2 (1b)

We compute signatures level by level, i.e., construct $Shrink_0^T, Pow_0^T, \ldots, Shrink_h^T, Pow_h^T$ incrementally. For each level, we create signatures by sorting signature blocks (or run-length encoded signatures) to which we give signatures, as shown by the next two lemmas.

Lemma 19 *Given $LC(Pow_{t-1}^T)$ for $0 < t \leq h$, we can compute $Shrink_t^T$ in $O((b-a) + |Pow_{t-1}^T|)$ time and space, where b is the maximum integer in Pow_{t-1}^T and a is the minimum integer in Pow_{t-1}^T.*

Proof. Since we assign signatures to signature blocks and run-length signatures in the derivation tree of S in the order they appear in the signature encoding, $Pow_{t-1}^T[i] - a$ fits in an entry of a bucket of size $b - a$ for each element of $Pow_{t-1}^T[i]$ of Pow_{t-1}^T. Recall that the length of each block is at most four. Hence we can sort all the blocks of $LC(Pow_{t-1}^T)$ by bucket sort in $O((b-a) + |Pow_{t-1}^T|)$ time and space. Since *Assgn* is an injection and since we process the levels in increasing order, for any two different levels $0 \leq t' < t \leq h$, no elements of $Shrink_{t-1}^T$ appear in $Shrink_{t'-1}^T$, and hence no elements of Pow_{t-1}^T appear in $Pow_{t'-1}^T$. Thus, we can determine a new signature for each block in $LC(Pow_{t-1}^T)$, *without* searching existing signatures in the lower levels. This completes the proof. □

Lemma 20 *Given $RLE(Shrink_t^T)$, we can compute Pow_t^T in $O(x+(b-a)+|RLE(Shrink_t^T)|)$ time and space, where x is the maximum length of runs in $RLE(Shrink_t^T)$, b is the maximum integer in Pow_{t-1}^T, and a is the minimum integer in Pow_{t-1}^T.*

Proof. We first sort all the elements of $RLE(Shrink_t^T)$ by bucket sort in $O(b-a+|RLE(Shrink_t^T)|)$ time and space, ignoring the powers of runs. Then, for each integer r appearing in $Shrink_t^T$, we sort the runs of r's by bucket sort with a bucket of size x. This takes a total of $O(x+|RLE(Shrink_t^T)|)$ time and space for all integers appearing in $Shrink_t^T$. The rest is the same as the proof of Lemma 19. □

Proof. [Proof of Theorem 2 (1b)] Since the size of the derivation tree of $id(T)$ is $O(N)$, by Lemmas 3, 19, and 20, we can compute a DAG of $\mathcal{G}$ for T in $O(N)$ time and space. □

3.2.5 Proof of Theorem 2 (3b)

In this section, we sometimes abbreviate $val(X)$ as X for $X \in \mathcal{S}$. For example, $Shrink_t^X$ and Pow_t^X represents $Shrink_t^{val(X)}$ and $Pow_t^{val(X)}$ respectively.

Our algorithm computes signatures level by level, i.e., constructs incrementally $Shrink_0^{X_n}, Pow_0^{X_n}, \ldots, Shrink_h^{X_n}, Pow_h^{X_n}$. Like the algorithm described in Subsection 3.2.4, we can create signatures by sorting blocks of signatures or run-length encoded signatures in the same level. The main difference is that we now utilize the structure of the SLP, which allows us to do the task efficiently in $O(n \log^* M + w)$ working space. In particular, although $|Shrink_t^{X_n}|, |Pow_t^{X_n}| = O(N)$ for $0 \le t \le h$, they can be represented in $O(n \log^* M)$ space.

In so doing, we introduce some additional notations relating to $XShr_t^P$ and $XPow_t^P$ in Definition 7. By Lemma 14, there exist $\hat{z}_t^{(P_1,P_2)}$ and $z_t^{(P_1,P_2)}$ for any string $P = P_1P_2$ such that the following equation holds: $XShr_t^P = \hat{y}_t^{P_1}\hat{z}_t^{(P_1,P_2)}\hat{y}_t^{P_2}$ for $0 < t \le h^P$, and $XPow_t^P = y_t^{P_1}z_t^{(P_1,P_2)}y_t^{P_2}$ for $0 \le t < h^P$, where we define $\hat{y}_t^P$ and y_t^P for a string P as:

$$\hat{y}_t^P = \begin{cases} XShr_t^P & \text{for } 0 < t \le h^P, \\ \varepsilon & \text{for } t > h^P, \end{cases} \quad y_t^P = \begin{cases} XPow_t^P & \text{for } 0 \le t < h^P, \\ \varepsilon & \text{for } t \ge h^P. \end{cases}$$

For any variable $X_i \to X_\ell X_r$, we denote $\hat{z}_t^{X_i} = \hat{z}_t^{(val(X_\ell),val(X_r))}$ (for $0 < t \le h^{val(X_i)}$) and $z_t^{X_i} = z_t^{(val(X_\ell),val(X_r))}$ (for $0 \le t < h^{val(X_i)}$). Note that $|z_t^{X_i}|, |\hat{z}_t^{X_i}| = O(\log^* M)$ because $z_t^{X_i}$ is created on $\hat{R}_t^{X_\ell}\hat{z}_t^{X_i}\hat{L}_t^{X_r}$, similarly, $\hat{z}_t^{X_i}$ is created on $R_{t-1}^{X_\ell}z_{t-1}^{X_i}L_{t-1}^{X_r}$. We can

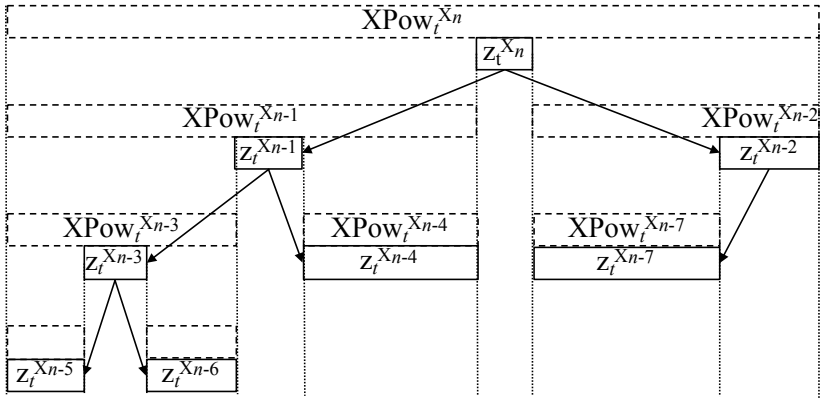

Figure 3.1: $XPow_t^{X_n}$ can be represented by $z_t^{X_1}, \ldots, z_t^{X_n}$. In this example, $XPow_t^{X_n} = z_t^{X_{n-5}} z_t^{X_{n-3}} z_t^{X_{n-6}} z_t^{X_{n-1}} z_t^{X_{n-4}} z_t^{X_n} z_t^{X_{n-7}} z_t^{X_{n-2}}$.

use $\hat{z}_t^{X_1}, \ldots, \hat{z}_t^{X_n}$ (resp. $z_t^{X_1}, \ldots, z_t^{X_n}$) as a compressed representation of $XShr_t^{X_n}$ (resp. $XPow_t^{X_n}$) based on the SLP: Intuitively, $\hat{z}_t^{X_n}$ (resp. $z_t^{X_n}$) covers the middle part of $XShr_t^{X_n}$ (resp. $XPow_t^{X_n}$) and the remaining part is recovered by investigating the left/right child recursively (see also Fig. 3.1). Hence, with the DAG structure of the SLP, $XShr_t^{X_n}$ and $XPow_t^{X_n}$ can be represented in $O(n \log^* M)$ space.

In addition, we define $\hat{A}_t^P$, $\hat{B}_t^P$, A_t^P and B_t^P as follows: For $0 < t \leq h^P$, $\hat{A}_t^P$ (resp. $\hat{B}_t^P$) is a prefix (resp. suffix) of $Shrink_t^P$ which consists of signatures of $A_{t-1}^P L_{t-1}^P$ (resp. $R_{t-1}^P B_{t-1}^P$); and for $0 \leq t < h^P$, A_t^P (resp. B_t^P) is a prefix (resp. suffix) of Pow_t^P which consists of signatures of $\hat{A}_t^P \hat{L}_t^P$ (resp. $\hat{R}_t^P \hat{B}_t^P$). By the definition, $Shrink_t^P = \hat{A}_t^P XShr_t^P \hat{B}_t^P$ for $0 \leq t \leq h^P$, and $Pow_t^P = A_t^P XPow_t^P B_t^P$ for $0 \leq t < h^P$. See Fig. 3.2 for the illustration.

Since $Shrink_t^{X_n} = \hat{A}_t^{X_n} XShr_t^{X_n} \hat{B}_t^{X_n}$ for $0 < t \leq h^{X_n}$, we use $\hat{\Lambda}_t = (\hat{z}_t^{X_1}, \ldots, \hat{z}_t^{X_n}, \hat{A}_t^{X_n}, \hat{B}_t^{X_n})$ as a compressed representation of $Shrink_t^{X_n}$ of size $O(n \log^* M)$. Similarly, for $0 \leq t < h^{X_n}$, we use $\Lambda_t = (z_t^{X_1}, \ldots, z_t^{X_n}, A_t^{X_n}, B_t^{X_n})$ as a compressed representation of $Pow_t^{X_n}$ of size $O(n \log^* M)$.

Our algorithm computes incrementally $\Lambda_0, \hat{\Lambda}_1, \ldots, \hat{\Lambda}_{h^{X_n}}$. Given $\hat{\Lambda}_{h^{X_n}}$, we can easily get $Pow_{h^{X_n}}^{X_n}$ of size $O(\log^* M)$ in $O(n \log^* M)$ time, and then $id(val(X_n))$ in $O(\log^* M)$ time from $Pow_{h^{X_n}}^{X_n}$. Hence, in the following three lemmas, we show how to compute $\Lambda_0, \hat{\Lambda}_1, \ldots, \hat{\Lambda}_{h^{X_n}}$.

Lemma 21 *Given an SLP of size n, we can compute Λ_0 in $O(n \log\log(n \log^* M) \log^* M)$*

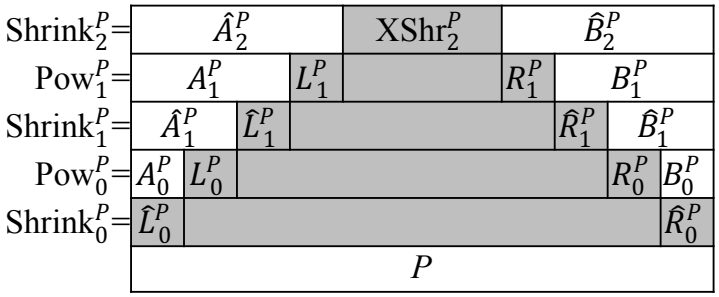

Figure 3.2: An abstract image of $Shrink_t^P$ and Pow_t^P for a string P. For $0 \leq t < h^P$, $A_t^P L_t^P$ (resp. $R_t^P B_t^P$) is encoded into $\hat{A}_{t+1}^P$ (resp. $\hat{B}_{t+1}^P$). Similarly, for $0 < t < h^P$, $\hat{A}_t^P \hat{L}_t^P$ (resp. $\hat{R}_t^P \hat{B}_t^P$) is encoded into A_t^P (resp. B_t^P).

time and $O(n \log^* M)$ *space.*

Proof. We first compute, for all variables X_i, $RLE(XShr_0^{X_i})$ if $|RLE(XShr_0^{X_i})| \leq \delta_C$, otherwise $RLE(\hat{L}_0^{X_i})$ and $RLE(\hat{R}_0^{X_i})$. The information can be computed in $O(n \log^* M)$ time and space in a bottom-up manner, i.e., by processing variables in increasing order. For $X_i \to X_\ell X_r$, if both $|RLE(XShr_0^{X_\ell})|$ and $|RLE(XShr_0^{X_r})|$ are no greater than δ_C, we can compute $RLE(XShr_0^{X_i})$ in $O(\log^* M)$ time by naively concatenating $RLE(XShr_0^{X_\ell})$ and $RLE(XShr_0^{X_r})$. Otherwise $|RLE(XShr_0^{X_i})| > \delta_C$ must hold, and $RLE(\hat{L}_0^{X_i})$ and $RLE(\hat{R}_0^{X_i})$ can be computed in $O(1)$ time from the information for X_ℓ and X_r.

The run-length encoded signatures represented by $z_0^{X_i}$ can be obtained by using the above information for X_ℓ and X_r in $O(\log^* M)$ time: $z_0^{X_i}$ is created over run-length encoded signatures $RLE(XShr_0^{X_\ell})$ (or $RLE(\hat{R}_0^{X_\ell})$) followed by $RLE(XShr_0^{X_r})$ (or $RLE(\hat{R}_0^{X_r})$). Also, by definition $A_0^{X_n}$ and $B_0^{X_n}$ represents $RLE(\hat{L}_0^{X_n})$ and $RLE(\hat{R}_0^{X_n})$, respectively.

Hence, we can compute in $O(n \log^* M)$ time $O(n \log^* M)$ run-length encoded signatures to which we give signatures. We determine signatures by sorting the run-length encoded signatures as Lemma 20. However, in contrast to Lemma 20, we do not use bucket sort for sorting the powers of runs because the maximum length of runs could be as large as N and we cannot afford $O(N)$ space for buckets. Instead, we use the sorting algorithm of Han [29] which sorts x integers in $O(x \log\log x)$ time and $O(x)$ space. Hence, we can compute Λ_0 in $O(n \log\log(n \log^* M) \log^* M)$ time and $O(n \log^* M)$ space. □

Lemma 22 *Given $\hat{\Lambda}_t$, we can compute Λ_t in $O(n \log\log(n \log^* M) \log^* M)$ time and $O(n \log^* M)$ space.*

Proof. The computation is similar to that of Lemma 21 except that we also use $\hat{\Lambda}_t$.

We first compute, for all variables X_i, $RLE(XShr_t^{X_i})$ if $|RLE(XShr_t^{X_i})| \leq \delta_C$, otherwise $RLE(\hat{L}_t^{X_i})$ and $RLE(\hat{R}_t^{X_i})$. The information can be computed in $O(n \log^* M)$ time and space in a bottom-up manner, i.e., by processing variables in increasing order. For $X_i \to X_\ell X_r$, if both $|RLE(XShr_t^{X_\ell})|$ and $|RLE(XShr_t^{X_r})|$ are no greater than δ_C, we can compute $RLE(XShr_t^{X_i})$ in $O(\log^* M)$ time by naively concatenating $RLE(XShr_t^{X_\ell})$, $RLE(\hat{z}_t^{X_i})$ and $RLE(XShr_t^{X_r})$. Otherwise $|RLE(XShr_t^{X_i})| > \delta_C$ must hold, and $RLE(\hat{L}_0^{X_i})$ and $RLE(\hat{R}_0^{X_i})$ can be computed in $O(1)$ time from $RLE(\hat{z}_t^{X_i})$ and the information for X_ℓ and X_r.

The run-length encoded signatures represented by $z_t^{X_i}$ can be obtained in $O(\log^* M)$ time by using $\hat{z}_t^{X_i}$ and the above information for X_ℓ and X_r: $z_t^{X_i}$ is created over run-length encoded signatures that are obtained by concatenating $RLE(XShr_0^{X_\ell})$ (or $RLE(\hat{R}_0^{X_\ell})$), $z_t^{X_i}$ and $RLE(XShr_0^{X_r})$ (or $RLE(\hat{R}_0^{X_r})$). Also, $A_t^{X_n}$ and $B_t^{X_n}$ represents $\hat{A}_t^{X_n}\hat{L}_t^{X_n}$ and $\hat{R}_t^{X_n}\hat{B}_t^{X_n}$, respectively.

Hence, we can compute in $O(n \log^* M)$ time $O(n \log^* M)$ run-length encoded signatures to which we give signatures. We determine signatures in $O(n \log\log(n \log^* M) \log^* M)$ time by sorting the run-length encoded signatures as Lemma 22. □

Lemma 23 *Given Λ_t, we can compute $\hat{\Lambda}_{t+1}$ in $O(n \log^* M)$ time and $O(n \log^* M)$ space.*

Proof. In order to compute $\hat{z}_{t+1}^{X_i}$ for a variable $X_i \to X_\ell X_r$, we need a signature sequence on which $\hat{z}_{t+1}^{X_i}$ is created, as well as its context, i.e., Δ_L signatures to the left and Δ_R to the right. To be precise, the needed signature sequence is $v_t^{X_\ell} z_t^{X_i} u_t^{X_r}$, where $u_t^{X_j}$ (resp. $v_t^{X_j}$) denotes a prefix (resp. suffix) of $y_t^{X_j}$ of length $\Delta_L + \Delta_R + 4$ for any variable X_j (see also Figure 3.3). Also, we need $A_t u_t^{X_n}$ and $v_t^{X_n} B_t$ to create $\hat{A}_{t+1}^{X_n}$ and $\hat{B}_{t+1}^{X_n}$, respectively.

Note that by Definition 7, $|z_t^X| > \delta_C$ if $z_t^X \neq \varepsilon$. Then, we can compute $u_t^{X_i}$ for all variables X_i in $O(n \log^* M)$ time and space by processing variables in increasing order on the basis of the following fact: $u_t^{X_i} = u_t^{X_\ell}$ if $z_t^{X_\ell} \neq \varepsilon$, otherwise $u_t^{X_i}$ is the prefix of $z_t^{X_i}$ of length $\Delta_L + \Delta_R + 4$. Similarly $v_t^{X_i}$ for all variables X_i can be computed in $O(n \log^* M)$ time and space.

Using $u_t^{X_i}$ and $v_t^{X_i}$ for all variables X_i, we can obtain $O(n \log^* M)$ blocks of signatures to which we give signatures. We determine signatures by sorting the blocks by bucket

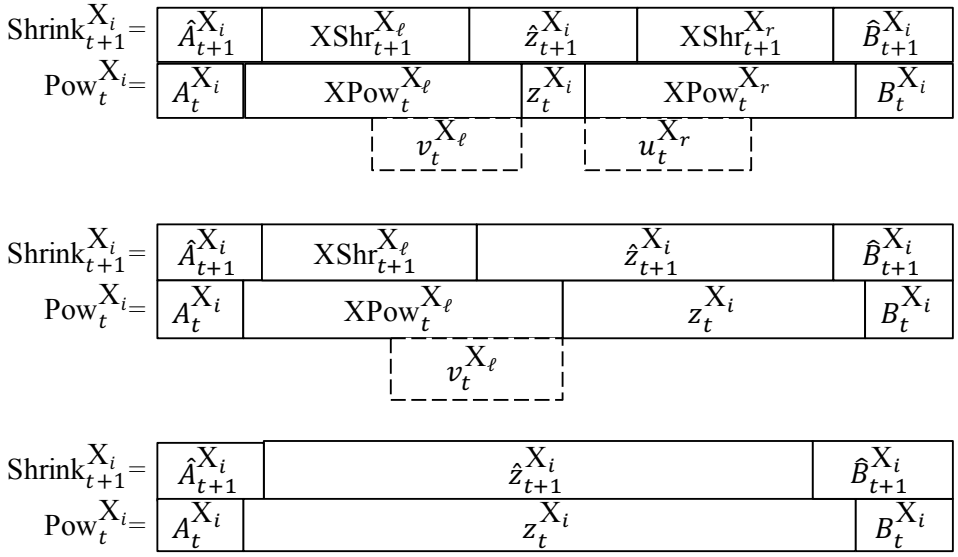

Figure 3.3: Abstract images of the needed signature sequence $v_t^{X_\ell} z_t^{X_i} u_t^{X_r}$ ($v_t^{X_\ell}$ and $u_t^{X_r}$ are not shown when they are empty) for computing $\hat{z}_{t+1}^{X_i}$ in three situations: Top for $0 \leq t < h^{X_\ell}, h^{X_r}$; middle for $h^{X_r} \leq t < h^{X_\ell}$; and bottom for $h^{X_\ell}, h^{X_r} \leq t < h^{X_i}$.

sort as in Lemma 19 in $O(n \log^* M)$ time. Hence, we can get $\hat{\Lambda}_{t+1}$ in $O(n \log^* M)$ time and space. □

Proof. [Proof of Theorem 2 (3b)] Using Lemmas 21, 22 and 23, we can get $\hat{\Lambda}_{h^{X_n}}$ in $O(n \log\log(n \log^* M) \log N \log^* M)$ time by computing $\Lambda_0, \hat{\Lambda}_1, \ldots, \hat{\Lambda}_{h^{X_n}}$ incrementally. Note that during the computation we only have to keep Λ_t (or $\hat{\Lambda}_t$) for the current t and the assignments of $\mathcal{G}$. Hence the working space is $O(n \log^* M + w)$. By processing $\hat{\Lambda}_{h^{X_n}}$ in $O(n \log^* M)$ time, we can get the DAG of $\mathcal{G}$ of size $O(w)$. □

3.3 Application

Theorem 8 is an application to text compression.

Theorem 8 *(1) Given a dynamic signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w which generates T, we can compute an SLP $\mathcal{S}$ of size $O(w \log |T|)$ generating T in $O(w \log |T|)$ time. (2) Let us conduct a single INSERT or DELETE operation on the string T generated by the SLP of (1). Let y be the length of the substring to be inserted or deleted,*

and let T' be the resulting string. During the above operation on the string, we can update, in $O((y + \log |T'| \log^* M)(f_{\mathcal{A}} + \log |T'|))$ time, the SLP of (1) to an SLP $\mathcal{S}'$ of size $O(w' \log |T'|)$ which generates T', where w' is the size of updated $\mathcal{G}$ which generates T'.

Proof. (1)For any signature $e \in \mathcal{V}$ such that $e \to e_\ell e_r$, we can easily translate e to a production of SLPs because the assignment is a pair of signatures, like the right-hand side of the production rules of SLPs. For any signature $e \in \mathcal{V}$ such that $e \to \hat{e}^k$, we can translate e to at most $2 \log k$ production rules of SLPs: We create $t = \lfloor \log k \rfloor$ variables which represent $\hat{e}^{2^1}, \hat{e}^{2^2}, \ldots, \hat{e}^{2^t}$ and concatenating them according to the binary representation of k to make up k $\hat{e}$'s. Therefore we can compute $\mathcal{S}$ in $O(w \log |T|)$ time.

(2)Note that the number of created or removed signatures in $\mathcal{V}$ is bounded by $O(y + \log |T'| \log^* M)$ by Lemma 15. For each of the removed signatures, we remove the corresponding production from $\mathcal{S}$. For each of created signatures, we create the corresponding production and add it to $\mathcal{S}$ as in the proof of (1). Therefore Theorem 8 holds. □

3.4 Conclusions and Future Work

In this chapter, we showed Theorems 1 and 2, and present a new application of signature encodings. We think that the algorithm of Theorem 2(3b) can speed up because the bottle neck is the range of value of a run in signature encodings. Since the range is $O(N)$, we used Han's sorting algorithm, however the case seem to be rare. If the range is $O(w)$, then we can use the bucket sorting in $O(w)$ space. Otherwise, there are a few runs whose the value range is $O(N)$ and many runs whose the value range is $O(w)$. Hence we may remove the bottle neck by applying some technique to the few runs.

Chapter 4

Dynamic Longest Common Extension

In this chapter, we present a compressed data structure which can solve the Dynamic LCE problem(Problem 3). Technically speaking, we show that signature encodings of a string T of length N and maximal length N_{max} can support LCE queries in $O(\log N + \log \ell \log^* N_{max})$ time. Since the signature encoding of T is in compressed space and supports update operations of T, signature encodings can solve the Dynamic LCE problem.

4.1 LCE Algorithm

In this section we show the following lemma. Note that Theorem 3 immediately follows from Lemma 24 by setting $M = 4N_{max}$.

Lemma 24 *Using a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ for a string T with an M-function, we can support queries $\mathsf{LCE}(s_1, s_2, i, j)$ and $\mathsf{LCE}(s_1^R, s_2^R, i, j)$ in $O(\log |s_1| + \log |s_2| + \log \ell \log^* M)$ time for given two signatures $e_1, e_2 \in \mathcal{V}$ and two integers $1 \leq i \leq |s_1|$, $1 \leq j \leq |s_2|$, where $s_1 = val(e_1)$, $s_2 = val(e_2)$ and ℓ is the answer to the LCE query.*

Proof. We focus on $\mathsf{LCE}(s_1, s_2, i, j)$ as $\mathsf{LCE}(s_1^R, s_2^R, i, j)$ is supported similarly.

Let P denote the longest common prefix of $s_1[i..]$ and $s_2[j..]$. Our algorithm simultaneously traverses two derivation trees rooted at e_1 and e_2 and computes P by matching the common signatures greedily from left to right. Recall that s_1 and s_2 are substrings of T. Since the both substrings P occurring at position i in $val(e_1)$ and at position j in $val(e_2)$ are represented by $Uniq(P)$ in the signature encoding by Lemma 14, we can

compute P by at least finding the common sequence of nodes which represents $Uniq(P)$, and hence, we only have to traverse ancestors of such nodes. By Lemma 15, the number of nodes we traverse, which dominates the time complexity, is upper bounded by $O(\log |s_1| + \log |s_2| + RLE(Uniq(P))) = O(\log |s_1| + \log |s_2| + \log \ell \log^* M)$. $\square$

4.2 Applications

In this section, we describe applications of our dynamic LCE data structures. Theorems 9-13 are applications to compressed string processing, where the task is to process a given compressed representation of string(s) without explicit decompression. We believe that only a few applications are listed here, considering the importance of LCE queries. As one example of unlisted applications, there is a paper [33] in which our LCE data structure was used to improve an algorithm of computing the Lyndon factorization of a string represented by a given SLP. Note that the time complexity of Theorem 2(3b) in Chapter 3 can be written as $O(n \log \log n \log N \log^* M)$ when $\log^* M = O(n)$ which in many cases is true, and always true in static case because $\log^* M = O(\log^* N) = O(\log N) = O(n)$.

We can get the next lemma using Theorem 2 (3b) and Theorem 1:

Lemma 25 *Given an SLP of size n representing a string of length N, we can sort the variables of the SLP in lexicographical order in $O(n \log n \log N \log^* N)$ time and $O(n \log^* N + w)$ working space.*

Lemma 25 has an application to an SLP-based index of Claude and Navarro [14]. In the paper, they showed how to construct their index in $O(n \log n)$ time if the lexicographic order of variables of a given SLP is already computed. However, in order to sort variables they almost decompressed the string, and hence, needs $\Omega(N)$ time and $\Omega(N \log |\Sigma|)$ bits of working space. Now, Lemma 25 improves the sorting part yielding the next theorem.

Theorem 9 *Given an SLP of size n representing a string of length N, we can construct the SLP-based index of [14] in $O(n \log n \log N \log^* N)$ time and $O(n \log^* N + w)$ working space.*

Theorem 10 *Given an SLP $\mathcal{S}$ of size n generating a string T of length N, we can construct, in $O(n \log \log n \log N \log^* N)$ time, a data structure which occupies $O(n \log N \log^*$*

N) space and supports $\mathsf{LCP}(val(X_i), val(X_j))$ *and* $\mathsf{LCS}(val(X_i), val(X_j))$ *queries for variables X_i, X_j in $O(\log N)$ time. The* $\mathsf{LCP}(val(X_i), val(X_j))$ *and* $\mathsf{LCS}(val(X_i), val(X_j))$ *query times can be improved to $O(1)$ using $O(n \log n \log N \log^* N)$ preprocessing time.*

We use the following known result to show Theorem 10.

Lemma 26 ([3]) *Let $\mathcal{T} = \{T_1, \ldots, T_k\}$. Using a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ such that $id(T_1), \ldots, id(T_k) \in \mathcal{V}$, we can support following operations for two strings $T_i, T_j \in \mathcal{T}$:*

- *$LCP(T_i, T_j)$ in $O(\log |T_i| + \log |T_j|)$ time,*
- *$LCS(T_i, T_j)$ in $O((\log |T_i| + \log |T_j|) \log^* M)$ time*

Proof. We compute $LCP(T_i, T_j)$ by $LCE(T_i, T_j, 1, 1)$, namely, we use the algorithm of Lemma 24. Let P denote the longest common prefix of T_i and T_j. We use the notation $\hat{A}^P$ defined in Subsection 3.2.5. Then the both substrings P occurring at position 1 in T_i and at position 1 in T_j are represented as $v = \hat{A}^P_{h^P} XShr^P_{h^P} R^P_{h^P-1} \hat{R}^P_{h^P-1} \cdots R^P_0 \hat{R}^P_0$ in the signature encoding by a similar argument of Lemma 14. Since $|RLE(v)| = O(\log |P| + \log^* M)$, we can compute $LCP(T_i, T_j)$ in $O(\log |T_i| + \log |T_j|)$ time. Similarly, we can compute $LCS(T_i, T_j)$ in $O((\log |T_i| + \log |T_j|) \log^* M)$ time. More detailed proofs can be found in [3]. □

To use Lemma 26 for $id(val(X_1)), \ldots, id(val(X_n))$, we show the following lemma.

Lemma 27 *Given an SLP $\mathcal{S}$, we can compute $id(val(X_1)), \ldots, id(val(X_n))$ in $O(n \log \log n \log N \log^* M)$ time and $O(n \log N \log^* M)$ space.*

Proof. Recall that the algorithm of Theorem 2 (3) computes $id(val(X_n))$ in $O(n \log \log n \log N \log^* M)$ time. We can modify the algorithm to compute $id(val(X_1)), \ldots, id(val(X_n))$ without changing the time complexity: We just compute A^X_t, $\hat{A}^X_t$, B^X_t and $\hat{B}^X_t$ for "all" $X \in \mathcal{S}$, not only for X_n. Since the total size is $O(n \log N \log^* M)$, Lemma 27 holds. □

We are ready to prove Theorem 10.

Proof. The first result immediately follows from Lemma 26 and 27. To speed-up query times for LCP and LCS, we sort variables in lexicographical order in $O(n \log n \log N)$

time by LCP query and a standard comparison-based sorting. Constant-time LCP queries are then possible by using a constant-time RMQ data structure [8] on the sequence of the lcp values. Next we show that LCS queries can be supported similarly. Let SLP $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ and $Y_i \to expr_i$ for $1 \leq i \leq n$, where $expr_i = Y_r Y_\ell$ for $X_i \to X_\ell X_r \in \mathcal{D}$ and $expr_i = a$ for $(X_i \to a \in \Sigma) \in \mathcal{D}$. Then consider an SLP $\mathcal{S}' = (\Sigma, \mathcal{V}', \mathcal{D}, S')$ of size n, where $\mathcal{V}' = \{Y_1, \ldots, Y_n\}$, $\mathcal{D}' = \{Y_1 \to expr_i, \ldots, Y_n \to expr_n\}$ and $S' = Y_n$. Namely $\mathcal{S}'$ represents T^R. By supporting LCP queries on $\mathcal{S}'$, LCS queries on $\mathcal{S}$ can be supported. Hence Theorem 10 holds. □

Theorem 11 *Given an SLP $\mathcal{S}$ of size n generating a string T of length N, there is a data structure of $O(w+n)$ space which supports queries $\mathsf{LCE}(val(X_i), val(X_j), a, b)$ for variables X_i, X_j, $1 \leq a \leq |X_i|$ and $1 \leq b \leq |X_j|$ in $O(\log N + \log \ell \log^* N)$ time, where $w = O(z \log N \log^* N)$, $z = |LZ77_{wo}(T)|$ and ℓ is the answer of LCE query. The data structure can be constructed in $O(n \log\log n \log N \log^* N)$ preprocessing time and $O(n \log^* N + w)$ working space.*

Proof. We can compute a static signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w representing T in $O(n \log\log n \log N \log^* M)$ time and $O(n \log^* M + w)$ working space using Theorem 2, where $w = O(z \log N \log^* M)$. Notice that each variable of the SLP appears at least once in the derivation tree of T_n of the last variable X_n representing the string T. Hence, if we store an occurrence of each variable X_i in $\mathcal{T}_n$ and $|val(X_i)|$, we can reduce any LCE query on two variables to an LCE query on two positions of $val(X_n) = T$. In so doing, for all $1 \leq i \leq n$ we first compute $|val(X_i)|$ and then compute the leftmost occurrence ℓ_i of X_i in $\mathcal{T}_n$, spending $O(n)$ total time and space. By Lemma 24, each LCE query can be supported in $O(\log N + \log \ell \log^* M)$ time. Since $z \leq n$ [54], the total preprocessing time is $O(n \log\log n \log N \log^* M)$ and working space is $O(n \log^* M + w)$. □

Let h be the height of the derivation tree of a given SLP $\mathcal{S}$. Note that $h \geq \log N$. Matsubara et al. [42] showed an $O(nh(n + h \log N))$-time $O(n(n + \log N))$-space algorithm to compute an $O(n \log N)$-size representation of all palindromes in the string. Their algorithm uses a data structure which supports in $O(h^2)$ time, LCE queries of a special form $\mathsf{LCE}(val(X_i), val(X_j), 1, p_j)$ [42]. This data structure takes $O(n^2)$ space and can be constructed in $O(n^2 h)$ time [39]. Using Theorem 11, we obtain a faster algorithm, as follows:

Theorem 12 *Given an SLP of size n generating a string of length N, we can compute an $O(n \log N)$-size representation of all palindromes in the string in $O(n \log^2 N \log^* N)$ time and $O(n \log^* N + w)$ space.*

Proof. For a given SLP of size n representing a string of length N, let $P(n, N)$, $S(n, N)$, and $E(n, N)$ be the preprocessing time and space requirement for an LCE data structure on SLP variables, and each LCE query time, respectively.

Matsubara et al. [42] showed that we can compute an $O(n \log N)$-size representation of all palindromes in the string in $O(P(n, N) + E(n, N) \cdot n \log N)$ time and $O(n \log N + S(n, N))$ space. Hence, using Theorem 11, we can find all palindromes in the string in $O(n \log \log n \log N \log^* M + n \log^2 N \log^* M) = O(n \log^2 N \log^* M)$ time and $O(n \log^* M + w)$ space. □

Our data structures also solve the grammar compressed dictionary matching problem.

Theorem 13 *Given a DSLP $\langle \mathcal{S}, m \rangle$ of size n that represents a dictionary $\Pi_{\langle \mathcal{S}, m \rangle}$ for m patterns of total length N, we can preprocess the DSLP in $O((n \log \log n + m \log m) \log N \log^* N)$ time and $O(n \log N \log^* N)$ space so that, given any text T in a streaming fashion, we can detect all occ occurrences of the patterns in T in $O(|T| \log m \log N \log^* N + occ)$ time.*

Proof. In the preprocessing phase, we construct a static signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w' such that $id(val(X_1)), \ldots, id(val(X_n)) \in \mathcal{V}$ using Lemma 27, spending $O(n \log \log n \log N \log^* M)$ time, where $w' = O(n \log N \log^* M)$. Next we construct a compacted trie of size $O(m)$ that represents the m patterns, which we denote by *PTree (pattern tree)*. Formally, each non-root node of *PTree* represents either a pattern or the longest common prefix of some pair of patterns. *PTree* can be built by using LCP of Theorem 10 in $O(m \log m \log N)$ time. We let each node have its string depth, and the pointer to its deepest ancestor node that represents a pattern if such exists. Further, we augment *PTree* with a data structure for level ancestor queries so that we can locate any prefix of any pattern, designated by a pattern and length, in *PTree* in $O(\log m)$ time by locating the string depth by binary search on the path from the root to the node representing the pattern. Supposing that we know the longest prefix of $T[i..|T|]$ that is also a prefix of one of the patterns, which we call the *max-prefix for i*, *PTree* allows us to output occ_i

patterns occurring at position i in $O(\log m + occ_i)$ time. Hence, the pattern matching problem reduces to computing the max-prefix for every position.

In the pattern matching phase, our algorithm processes T in a streaming fashion, i.e., each character is processed in increasing order and discarded before taking the next character. Just before processing $T[j+1]$, the algorithm maintains a pair of signature p and integer l such that $val(p)[1..l]$ is the longest suffix of $T[1..j]$ that is also a prefix of one of the patterns. When $T[j+1]$ comes, we search for the smallest position $i \in \{j-l+1, \ldots, j+1\}$ such that there is a pattern whose prefix is $T[i..j+1]$. For each $i \in \{j-l+1, \ldots, j+1\}$ in increasing order, we check if there exists a pattern whose prefix is $T[i..j+1]$ by binary search on a sorted list of m patterns. Since $T[i..j] = val(p)[i-j+l..l]$, LCE with p can be used for comparing a pattern prefix and $T[i..j+1]$ (except for the last character $T[j+1]$), and hence, the binary search is conducted in $O(\log m \log N \log^* M)$ time. For each i, if there is no pattern whose prefix is $T[i..j+1]$, we actually have computed the max-prefix for i, and then we output the occurrences of patterns at i. The time complexity is dominated by the binary search, which takes place $O(|T|)$ times in total. Therefore, the algorithm runs in $O(|T| \log m \log N \log^* M + occ)$ time.

By the way, one might want to know occurrences of patterns as soon as they appear as Aho-Corasick automata do it by reporting the occurrences of the patterns by their ending positions. Our algorithm described above can be modified to support it without changing the time and space complexities. In the preprocessing phase, we additionally compute *RPTree (reversed pattern tree)*, which is analogue to *PTree* but defined on the reversed strings of the patterns, i.e., *RPTree* is the compacted trie of size $O(m)$ that represents the reversed strings of the m patterns. Let $T[i..j]$ be the longest suffix of $T[1..j]$ that is also a prefix of one of the patterns. A suffix $T[i'..j]$ of $T[i..j]$ is called the *max-suffix for* j iff it is the longest suffix of $T[i..j]$ that is also a suffix of one of the patterns. Supposing that we know the max-suffix for j, *RPTree* allows us to output $eocc_j$ patterns occurring with ending position j in $O(\log m + eocc_j)$ time. Given a pair of signature p and integer l such that $T[i..j] = val(p)[1..l]$, the max-suffix for j can be computed in $O(\log m \log N \log^* M)$ time by binary search on a list of m patterns sorted by their "reversed" strings since each comparison can be done by "leftward" LCE with p. Except that we compute the max-suffix for every position and output the patterns ending at each position, everything else is the same as the previous algorithm, and hence, the time and space complexities are not changed. □

Note that Theorem 6 also the grammar compressed dictionary matching problem. Our data structure of Theorem 13 is always smaller, and runs faster when $h = \omega(\log m \log N \log^* N)$.

4.3 Conclusions and Future Work

In this chapter, we presented a dynamic compressed LCE data structures and its applications. As a future work, we are planning to present a dynamic compressed LCE data structures supporting LCE queries in $O(\log N)$ time. This realization is hard but we believe that it is not impossible.

Chapter 5

Dynamic Compressed Index and LZ77 Factorization

In this chapter, we handle the dynamic index problem and LZ77 factorization problem. Especially, we show Theorems 4 and 5.

5.1 Different Points with Previous Techniques

To emphasize the difference between our dynamic compressed index and previously proposed indexes, we explain that difference points.

To achieve Theorem 4, technically speaking, we use the signature encoding $\mathcal{G}$ of T. The signature encoding and the related ideas have been used in many applications. In particular, our dynamic compressed index has close relationship to Alstrup et al.'s index [4, 3], which is based on signature encodings, and the ESP-indices [60, 61], which are based on ESP. Note that ESP is an another version of signature encodings. Hence, we describe the difference points with their indices.

Alstrup et al.'s index is dynamic and non-compressed index which is based on the signature encoding of strings, while improving the update time of signature encodings [4] and the locally consistent parsing algorithm (details can be found in the technical report [3]).

Our data structure uses insert/delete update operations of signature encodings which are based on Alstrup et al.'s fast string concatenation/split algorithms (update algorithm) and linear-time computation of locally consistent parsing, but has little else in common than those. Especially, Alstrup et al.'s dynamic pattern matching algorithm [4, 3] requires to maintain specific locations called *anchors* over the parse trees of the signature

encodings, but our index does not use anchors, and also, it is different that we focus on the size of the dynamic index but they did not.

Our index also has close relationship to the ESP-indices [60, 61], but there are two significant differences between ours and ESP-indices: The first difference is that the ESP-index [60] is static and its online variant [61] allows only for appending new characters to the end of the text, while our index is fully dynamic allowing for insertion and deletion of arbitrary substrings at arbitrary positions. The second difference is that the pattern search time of the ESP-index is proportional to the number occ_c of occurrences of the so-called "core" of a query pattern P, which corresponds to a maximal subtree of the ESP derivation tree of a query pattern P. If occ is the number of occurrences of P in the text, then it always holds that $occ_c \geq occ$, and in general occ_c cannot be upper bounded by any function of occ. In contrast, as can be seen in Theorem 4, the pattern search time of our index is proportional to the number occ of occurrences of a query pattern P. This became possible due to our discovery of a new property of the signature encoding [3] (stated in Lemma 29).

5.2 The Idea of Our Searching Algorithm

As already mentioned in the introduction, our strategy for pattern matching is different from that of Alstrup et al. [3]. It is rather similar to the one taken in the static index for SLPs of Claude and Navarro [14]. Besides applying their idea to RSLPs, we show how to speed up pattern matching by utilizing the properties of signature encodings.

Index for SLPs. For given an SLP $\mathcal{S}$, we consider the static index of the SLP using Observations 3 and 4. Let prec/succ query for a given string P denote a query which asks $x^P_{prec}, x^P_{succ}, y^P_{prec}$ and y^P_{succ} in $\mathcal{S}$. Let $q_{prec/succ}, q_{report}, q_{vOcc}$ denote query times for computing prec/succ queries for a given string P, a reporting queries in $\mathcal{R}_{\mathcal{S}}$, and $vOcc(X, S)$ for a given variable $X \in \mathcal{V}$. Then the following lemma holds by Observations 3 and 4.

Lemma 28 (e.g. [14]) *For an SLP $\mathcal{S}$, Given a pattern P of length at least 2, we can compute $Occ(P, T)$ in $O(|P|(q_{prec/succ} + q_{report}) + |pOcc_{\mathcal{S}}(P)| \times q_{vOcc})$ time.*

Index for RSLPs. In Chapter 2, we already explained that Lemma 4 can be extended to the RSLP version by Observations 5 and 6. This means that we can easily extend

Lemma 28 to the RSLPs version. Specifically, the added task is to compute the maximal integer x such that X is (j, x)-stabbed variable of P for a given variable $X \to \hat{X}^k$ such that X is j-stabbed variable of P. Fortunately, we can compute this x in constant time using the fact that X is the run of $\hat{X}$. Hence the RSLPs version of Lemma 28 achieve the same bounds in Lemma 28.

Index for signature encodings. Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be a signature encoding of size w of a dynamic string T of current length N with an M-function. Since signature encodings are RSLPs, we can use the RSLPs version of Lemma 28 for $\mathcal{G}$. However, we can reduce the number of range reporting queries in Lemma 28. The following lemma is stronger than Observation 3 and this is a new property of signature encodings.

Lemma 29 *Let P be a string with $|P| > 1$. If $|Pow_0^P| = 1$, then $pOcc_{\mathcal{G}}(P, i) = \phi$ for $1 < i < |P|$. Otherwise, $pOcc_{\mathcal{G}}(P, i) = \phi$ holds for $i \in [1..|P| - 1] \setminus \mathcal{P}$, where $\mathcal{P} = \{|val^+(u[1..i])| \mid 1 \leq i < |u|, u[i] \neq u[i+1]\}$ with $u = Uniq(P)$.*

Proof. If $|Pow_0^P| = 1$, then $P = a^{|P|}$ for some character $a \in \Sigma$. In this case, P must be contained in a node labeled with a signature $e \to \hat{e}^d$ such that $\hat{e} \to a$ and $d \geq |P|$. Hence, all stabbing variables of P can be found by $pOcc_{\mathcal{G}}(P, 1)$.

If $|Pow_0^P| > 1$, we consider the common sequence u of P. Recall that substring P occurring at j in $val(e)$ is represented by u for any $e \in pOcc(P, j)$ by Lemma 14. Hence at least $pOcc_{\mathcal{G}}(P, i) = \phi$ holds for $i \in [1..|P|-1] \setminus \mathcal{P}'$, where $\mathcal{P}' = \{|val^+(u[1])|, \ldots, |val^+(u[..|u|-1])|\}$. Moreover, we show that $pOcc_{\mathcal{G}}(P, i) = \emptyset$ for any $i \in \mathcal{P}'$ with $u[i] = u[i+1]$. Note that $u[i]$ and $u[i+1]$ are encoded into the same signature in the derivation tree of e, and that the parent of two nodes corresponding to $u[i]$ and $u[i+1]$ has a signature e' in the form $e' \to u[i]^d$. Now assume for the sake of contradiction that $e = e'$. By the definition of the stabbing variables, $i = 1$ must hold, and hence, $Shrink_0^P[1] = u[1] \in \Sigma$. This means that $P = u[1]^{|P|}$, which contradicts $|Pow_0^P| > 1$. Therefore the statement holds. □

Lemma 29 means that all we have to do is to compute $pOcc(P, i)$ for $i \in \mathcal{P}$. Hence the following lemma holds by Lemmas 28 and 29. Note that for computing $\mathcal{P}$, we need to compute the common sequence of P using Lemma 16 in $O(\log |P| \log^* M)$ time, and also, need to compute $id(P)$ in $O(|P| f_{\mathcal{A}})$ time.

Lemma 30 *Let $\mathcal{G}$ be a dynamic signature encoding of T using Theorem 1. Given a pattern P, we can compute $Occ(P,T)$ in $O(|P|f_{\mathcal{A}}+\log|P|\log^* M+|\mathcal{P}|(q_{prec/succ}+q_{report})+|pOcc_{\mathcal{S}}(P)|\times q_{vOcc})$ time using $\mathcal{G}$.*

5.3 Dynamic Compressed Index for Signature Encodings

In this section, we propose a new dynamic compressed index using Lemma 30. Since $\mathcal{G}$ supports computing $vOcc(e,S)$ for a given signature e and $id(P)$ for a given pattern P, our remain tasks are considering dynamic data structures supports prec/succ queries for a given string P and reporting queries in $\mathcal{R}_{\mathcal{G}}$. We can construct the dynamic data structure supporting prec/succ queries for a given string P using the following lemma.

Lemma 31 *For a signature encoding $\mathcal{G}$ of size w, there exists a data structure of size $O(w)$ which can compute $x^P_{prec}, x^P_{succ}, y^P_{prec}$ and y^P_{succ} for $e\in\mathcal{V}$ in $O(\log w(\log N'+\log|val(e)|\log^* M))$, where P is a substring of $val(e)$ and $N'=\max\{|val(e')|\mid e'\in\mathcal{V}\}$. For a given signature e added into/removed from $\mathcal{G}$, x^e_{pred} and y^e_{pred}, this data structure can be updated in $O(\log w)$ time, where $x^e_{pred}=\max\{x'\in\mathcal{X}_{\mathcal{G}}\mid x'\le e.\text{left}^R\}$ and $y^e_{pred}=\max\{y'\in\mathcal{Y}_{\mathcal{G}}\mid y'\le e.\text{right}\}$.*

Proof. Consider two self-balancing search trees for $\mathcal{X}_{\mathcal{G}}$ and $\mathcal{Y}_{\mathcal{G}}$. We can efficiently compute $x^P_{prec}, x^P_{succ}, y^P_{prec}$ and y^P_{succ} using LCE queries by $\mathcal{G}$(Theorem 3) and binary search on these trees.

For a given signature e added into/removed from $\mathcal{G}$, we can update $\mathcal{X}_{\mathcal{G}}$ and $\mathcal{Y}_{\mathcal{G}}$ in $O(\log w)$ time using x^e_{pred} and y^e_{pred}. Hence Lemma 31 holds. □

Next, we can construct the dynamic data structure supporting reporting queries in $\mathcal{R}_{\mathcal{G}}$ using the following lemma.

Lemma 32 *For a signature encoding $\mathcal{G}$ of size w, there exists a data structure of size $O(w)$ which can compute $report_{\mathcal{R}_{\mathcal{G}}}(x_1,x_2,y_1,y_2)$ for given x_1,x_2,y_1 and y_2 in $O(\log w+occ(\log w/\log\log w))$ time. For a given signature e added into/removed from $\mathcal{G}$, x^e_{pred} and y^e_{pred}, this data structure can be updated in amortized $O(\log w)$ time.*

Proof. Consider a Blelloch's dynamic 2D range reporting data structure for $\mathcal{R}_{\mathcal{G}}$ and two Diez and Sleator's order maintenance data structures for $\mathcal{X}_{\mathcal{G}}$ and $\mathcal{Y}_{\mathcal{G}}$. Blelloch's data structure uses Diez and Sleator's data structures to compare any two elements in $\mathcal{R}_{\mathcal{G}}$ in constant time. Hence this range reporting data structure can support $report_{\mathcal{R}_{\mathcal{G}}}$ in $O(\log w + occ(\log w/\log\log w))$ time. Since these data structure clearly can be updated in amortized $O(\log w)$ time using x^e_{pred} and y^e_{pred}, Lemma 32 holds. □

We show Theorem 4.

Proof. We focus on the case $|Pow^P_0| > 1$ as the other case is easier to be solved. By Lemma 31, we get $q_{prec/succ} = O(\log w(\log N + \log|P|\log^* M))$ for a string $P[..i]$ and $P[i+1..]$, where $id(P) \in \mathcal{V}$ and $1 \leq i < |P|$. By Lemma 32, we get $q_{report} = O(\log w + occ(\log w/\log\log w))$. Hence, by Lemma 30, our dynamic index of $O(w)$ space supports *FIND* queries in $O(|P|f_{\mathcal{A}} + \log w \log|P|\log^* M(\log N + \log|P|\log^* M) + occ\log N)$ time.

Next, we consider the update of our dynamic index. For a given signature e added into/removed from $\mathcal{G}$, we can compute x^e_{pred} and y^e_{pred} in $O(\log w \log N \log^* M)$ time using Lemma 31. Since the number of signatures added to or removed from $\mathcal{G}$ during a single update operation is upper bounded by Lemma 17, the update part of Theorem 4 holds by Theorem 1, Lemmas 31 and 32. Therefore Theorem 4 holds. □

5.4 LZ77 Factorization Algorithm using Signature Encodings

In this section, we show Theorem 5 using the idea of our dynamic index. For integers j, k with $1 \leq j \leq j+k-1 \leq N$, let $Fst(j,k)$ be the function which returns the minimum integer i such that $i < j$ and $T[i..i+k-1] = T[j..j+k-1]$, if it exists. To compute LZ77 factorization, we use following fact and lemma.

fact 1 *Let $f_1, \ldots, f_z = LZ77_{wo}(T)$. Given $f_1, \ldots, f_{i-1}$, we can compute f_i with $O(\log|f_i|)$ calls of $Fst(j,k)$ (by doubling the value of k, followed by a binary search), where $j = |f_1 \cdots f_{i-1}| + 1$.*

Lemma 33 *Given a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w which generates T of length N with an M-function, we can construct a data structure of $O(w)$ space in*

$O(w \log w \log N \log^* M)$ time to support queries $Fst(j,k)$ in $O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time.

Proof. We explain how to support queries $Fst(j,k)$ efficiently using the signature encoding. We define $e.\min = \min vOcc(e,S) + |e.\text{left}|$ for a signature $e \in \mathcal{V}$ with $e \to e_\ell e_r$ or $e \to \hat{e}^k$. We also define $FstOcc(P,i)$ for a string P and an integer i as follows:

$$FstOcc(P,i) = \min\{e.\min \mid e \in pOcc_{\mathcal{G}}(P,i)\}$$

Then $Fst(j,k)$ can be represented by $FstOcc(P,i)$ as follows:

$$\begin{aligned} Fst(j,k) &= \min\{FstOcc(T[j..j+k-1],i) - i \mid i \in \{1,\ldots,k-1\} \\ &= \min\{FstOcc(T[j..j+k-1],i) - i \mid i \in \mathcal{P}\}, \end{aligned}$$

where $\mathcal{P}$ is the set of integers in Lemma 29 with $P = T[j..j+k-1]$. Hence we can compute $Fst(j,k)$ efficiently by computing $FstOcc(P,i)$ efficiently. We can compute $FstOcc(P,i)$ by a $rmq(x_{prec}^{P[..i]}, x_{succ}^{P[..i]}, y_{prec}^{P[i+1..]}, y_{succ}^{P[i+1..]})$ query of Lemma 5 because we can regard a signature e as a 2D weighted point by regarding $e.\min$ as its weight. (Recall that we regarded e as a 2D point in Section 5.2.) Note that we already described how to compute $x_{prec}^{P[..i]}, x_{succ}^{P[..i]}, y_{prec}^{P[i+1..]}$ and $y_{succ}^{P[i+1..]}$, and also, we can compute $RLE(Uniq(P))$ with $P = T[j..j+k-1]$ in $O(\log N + \log k \log^* M)$ time by Lemma 16. Hence we can compute $Fst(j,k)$ in $O(\log k \log^* M(\log w(\log N + \log k \log^* M) + \log^2 w)) = O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time.

For construction, we first compute $e.\min$ in $O(w)$ time for all $e \in \mathcal{V}$ using the DAG of $\mathcal{G}$. Next, given $\mathcal{G}$, $\mathcal{X}_{\mathcal{G}}$ (and $\mathcal{Y}_{\mathcal{G}}$) can be sorted in $O(w \log w \log N \log^* M)$ time by LCE algorithm (Lemma 24) and a standard comparison-based sorting. Finally we build the data structure of Lemma 5 in $O(w \log w)$ time. □

Hence Theorem 5 immediately holds by Fact 1 and Lemma 33. We remark that we can similarly compute the LZ77 factorization with self-reference of a text in the same time and same working space.

5.5 Conclusions and Future Work

In this chapter, we presented a new dynamic compressed index and LZ77 factorization algorithm in compressed space. As a future work, we are planning to present a new

dynamic compressed index which is based on signature encodings and supports faster find queries. Specifically, we are planning to realize the Alstrup's anchor technique in compressed space. If it is possible, the speeding up find queries is easy. Furthermore, we can present faster LZ77 factorization in compressed space using the anchor technique. These realizations are hard but we believe that they are not impossible.

Chapter 6

Compressed Automata for Dictionary Matching

In this chapter, we show Theorem 6. We describe the process of the proof of Theorem 6. Recall the AC automaton introduced in Chapter 2, which consists of goto, failure and output functions. We show to emulate these functions in compressed space. In Subsection 6.1.1, we show to compute goto function in $O(\log N)$ time using $O(n)$ space by preprocessing $O(n^3 \log n)$ time and $O(n^2)$ working space(Lemma 35). In Subsections 6.1.2 and 6.1.3, we show to compute failure function in $O(\log n)$ time using $O(n^2 \log N)$ space by preprocessing $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ working space(Lemma 42). In Subsection 6.1.4, we show to compute output function in $O(h + m)$ time using $O(nm)$ space by preprocessing $O(n^3 \log n)$ time and $O(n^2)$ working space(Lemma 45). In Subsection 6.1.5, we show Theorem 6 using above results. In Section 6.2, we describe a compressed MP-automaton.

6.1 Compressed AC Automata

In this section, we consider the AC automaton for $\Pi_{\mathcal{S}} = \Pi_{\langle \mathcal{S},n \rangle} = \{val(X_i) \mid i \in [1..n]\}$, not for $\Pi_{\langle \mathcal{S},m \rangle}$. Independently of $m \in [1..n]$, we use the goto and the failure functions of this automaton, and adjust the output function appropriately for $\Pi_{\langle \mathcal{S},m \rangle}$.

6.1.1 Compact Representation of G-Trie

For a compact representation of the g-trie, we can adopt the so-called path compaction technique like the suffix trees [63]. The *compact g-trie* for $\mathcal{S} = \{X_i \to expr_i\}_{i=1}^{n}$ is the

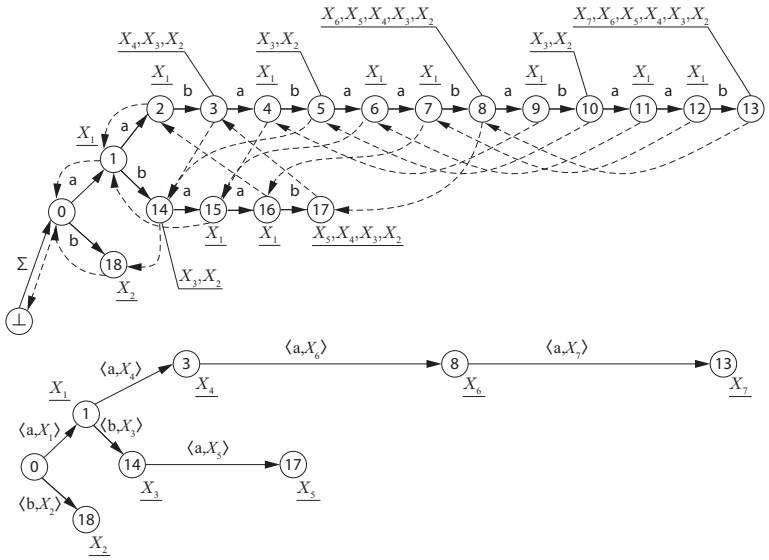

Figure 6.1: The AC automaton and the compact g-trie for $\Pi_{\mathcal{S}}$ are displayed on the upper and on the lower, respectively, where $\mathcal{S}$ is identical to the SLP of Figure 1.1.

path-compacted trie obtained from the g-trie for $\{val(X_i) \mid i \in [1..n]\}$ by removing the implicit states, where every edge e from q to r (let q and r be explicit states representing strings u and uv, respectively) is labeled by $\langle a, X_i \rangle$ such that $a = v[1]$, $X_i[1..|uv|] = uv$ and X_i stabs $[1..|uv|]$. The next lemma directly follows from Lemma 7.

Lemma 34 *There are at most $2n$ states in the compact g-trie for $\mathcal{S}$ of size n.*

Figure 6.1 displays the AC automaton and the compact g-trie for $\Pi_{\mathcal{S}}$ where $\mathcal{S}$ is identical to the example SLP of Figure 1.1.

An implicit state q' on edge $e = (q, r)$ can be specified by an integer $h \geq 1$ such that q' represents the string $X_i[1..|u| + h]$ and X_i stabs $[1..|u| + h]$, where q represents string u and e is labeled by $\langle a, X_i \rangle$.

Lemma 35 *An $O(n)$-space compact g-trie can be constructed in $O(n^3 \log n)$ time and $O(n^2)$ space so that for any state q and any character c, $g(q, c)$ can be determined in $O(\log N)$ time.*

Proof. We can compute in $O(n^3 \log n)$ time the sorted index σ of $\mathcal{S}$ and an array storing the longest common prefix length of $val(X_{\sigma(i)})$ and $val(X_{\sigma(i+1)})$ for all $i \in [1..n-1]$. Thus the compact g-trie can be constructed in $O(n^3 \log n)$ time.

When q is an explicit state, we can find the edge $e = (q, r)$ labeled by $\langle c, X_i \rangle$ for some variable X_i in $O(\log |\Sigma|)$ time, if such e exists, and we thus determine $g(q, c)$ in $O(\log |\Sigma|)$ time. When q is an implicit state on edge e specified by integer h, we can compute the $(h+1)$-th character in the string spelled out by e in $O(\log N)$ time by using the technique of Lemma 10, and then compare it with c to determine $g(q, c)$. □

Thus, we can represent the goto function compactly. A naive implementation of the failure function, however, requires exponential space. In the following two subsections, we describe how to represent the failure and the output functions in polynomial space with respect to n. By combining those results, we will finally show our main theorem in Subsection 6.1.5.

6.1.2 Compact Representation of Failure Function

As stated in the previous subsection we can represent any implicit state of the compact g-trie as a pair of an edge $e = (q, r)$ and an integer h. Here, we show another representation of states in the compact g-trie: A *reference-pair* of explicit/implicit state q is defined to be $\langle X_i, h \rangle$ such that q represents string $X_i[1..h]$ and X_i stabs $[1..h]$.

Lemma 36 *A mutual conversion between the two state representations can be performed in $O(\log n)$ time using some data structure of size $O(n^2)$.*

Proof. Let q be any state that represents string u. Suppose q is an explicit state. If q is terminating, let X_i be the variable corresponding to q, and otherwise, let X_i be the variable such that some out-going edge e from q is labeled by $\langle a, X_i \rangle$. Then, $\langle X_i, |u| \rangle$ gives a reference-pairs of q. Suppose q' is an implicit state on edge $e = (q, r)$ specified by integer h, and e is labeled by $\langle a, X_i \rangle$. Then, $\langle X_i, |u| + h \rangle$ gives a reference-pairs of q.

Conversely, suppose we are given a reference-pair $\langle X_i, h \rangle$ of some state q'. Then, it is possible to determine in $O(\log n)$ time the explicit state q that is the nearest ancestor of q', by using a simple binary search over the lengths of strings represented by the explicit states on the path from the initial state to the terminating state for X_i. □

Let $\mathit{Prefix}(\mathcal{S})$ denote the set of prefixes of $\mathit{val}(X_i)$ for all variables X_i in $\mathcal{S}$. For any variable $X_i \to X_l X_r \in \mathcal{S}$, an *f-interval* of X_i is a maximal element in the set $\{[b..e] \mid 1 < b \leq |X_l| < e \leq |X_i|, X_i[b..e] \in \mathit{Prefix}(\mathcal{S})\}$ with respect to the set inclusion relation $\subseteq$.

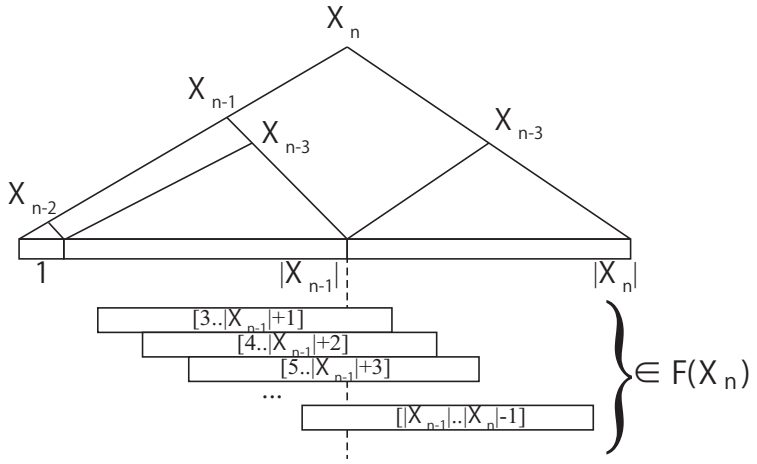

Figure 6.2: The f-interval sequence $\mathcal{F}(X_n)$ of length $2^{n-4}-1$ in Example 10 is illustrated.

The *f-interval sequence* of X_i, denoted $\mathcal{F}(X_i)$, is defined to be the sequence $\{[b_k..e_k]\}_{k=1}^{s}$ of all f-intervals of X_i arranged in the increasing order of b_k. By definition $e_1, \ldots, e_s$ are also arranged in the increasing order of e_k.

The set of f-interval sequences represents the failure function f as follows:

Lemma 37 *Let q be any state. Suppose q represents string $X_i[1..h]$. If $h = 1$, then $f(q)$ is the initial state. Suppose $h \geq 2$. Choose X_i so that X_i stabs $[1..h]$. Let $\{[b_k..e_k]\}_{k=1}^{s}$ be the f-interval sequence of X_i, and let $k' \in [1..s]$ be the smallest integer such that $h \in [b_{k'}..e_{k'}]$. Then, the state $f(s)$ represents the string $X_i[b_{k'}..h]$. If no such k' exist, then $f(q)$ represents the string $X_r[1..h - |X_l|]$ where $X_i \to X_l X_r \in \mathcal{S}$.*

A naive way of encoding the f-interval sequence $\{[b_k..e_k]\}_{k=1}^{s}$ of a variable X_i is to have a linear-list of triples of $\langle b_k, e_k, X_j \rangle$ such that $X_i[b_k..e_k] = X_j[1..e_k - b_k + 1]$ and X_j stabs $[1..e_k - b_k + 1]$. The list length s can, however, be exponential with respect to n.

Example 10 *Consider the SLP $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ and $\mathcal{D} = \{X_1 \to \mathtt{a}\} \cup \{X_i \to X_{i-1}X_{i-1}\}_{i=2}^{n-3} \cup \{X_{n-2} \to \mathtt{b}, X_{n-1} \to X_{n-2}X_{n-3}, X_n \to X_{n-1}X_{n-3}\}$. Then there are $2^{n-4}-1$ f-intervals of X_n. See Figure 6.2.*

Fortunately, we can prove Lemma 42 by making use of cyclic structures on f-intervals.

For any variable $X_i \to X_l X_r \in \mathcal{S}$ and any f-interval $[b..e] \in \mathcal{F}(X_i)$, if there is a run $[\alpha..\beta]$ with period p such that $\alpha \leq b < e \leq \beta$ and $e - b + 1 \geq 2p$, we say that the run $[\alpha..\beta]$ *subsumes* the f-interval $[b..e]$. Note that if such run exists, p is the smallest period of

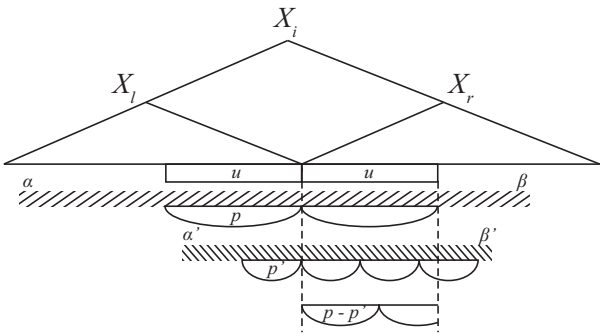

Figure 6.3: When $|X_l| + p \leq e < \beta'$ holds, p' and $p - p'$ are periods of u. From the periodicity lemma, $\gcd(p - p', p')$ is also a period of u. This contradicts that $p > \gcd(p - p', p')$ is the smallest period of $X_i[\alpha..\beta]$.

$X_i[b..e]$ and the run is unique with respect to $[b..e]$. If a run $[\alpha..\beta]$ subsumes two distinct f-intervals $[b..e]$ and $[b'..e']$ such that $X_i[b..e] = X_i[b'..e']$ and $b < b' \leq |X_l| - p$, $[\alpha..\beta]$ is said to be *f-rich*.

Lemma 38 *For any variable $X_i \to X_l X_r$ in $\mathcal{S}$, there is at most one f-rich run.*

Proof. The existence of an f-rich run $[\alpha..\beta]$ with period p implies that $u = X_i[|X_l| - p + 1..|X_l|] = X_i[|X_l| + 1..|X_l| + p]$. Also, from the definition of f-rich run, there must exist an f-interval $[b..e]$ such that $[b..e] \supseteq [|X_l| - p + 1..|X_l| + p]$.

Assume on the contrary that there is another f-rich run $[\alpha'..\beta']$ with period p' (w.l.o.g. assume $p' < p$). Since $X_i[|X_l| - p' + 1..|X_l|] = X_i[|X_l| + 1..|X_l| + p']$, $p - p'$ is a period of u. Since any interval contained in $[b..e]$ cannot be an f-interval, at least one of $\alpha' < b \leq |X_l| - p + 1$ or $|X_l| + p \leq e < \beta'$ must hold. In either case, we can see that u has a period p' (Figure 6.3 depicts the situation when $|X_l| + p \leq e < \beta'$ is assumed). It follows from the periodicity lemma that $\gcd(p - p', p')$ is a period of u, which means that p is not the smallest period of $X_i[\alpha..\beta]$, a contradiction. □

Lemma 39 *Let $X_i \to X_l X_r$ be any variable in $\mathcal{S}$. Let $[b..e]$ and $[b'..e']$ be the first and the last f-intervals subsumed by a run $[\alpha..\beta]$ with period p, respectively. For any d with $p \leq d < d + p < b' - b$, $[b + d..e''] \in \mathcal{F}(X_i) \iff [b + d + p..e'' + p] \in \mathcal{F}(X_i)$.*

Proof. We remark that $X_i[b..e']$ has period p.

Firstly, we show that $[b + d..e''] \in \mathcal{F}(X_i) \implies [b + d + p..e'' + p] \in \mathcal{F}(X_i)$. It is clear that $X_i[b + d..e''] \in \mathit{Prefix}(\mathcal{S})$. Note that $e'' < e'' + p < e'$ holds, since otherwise

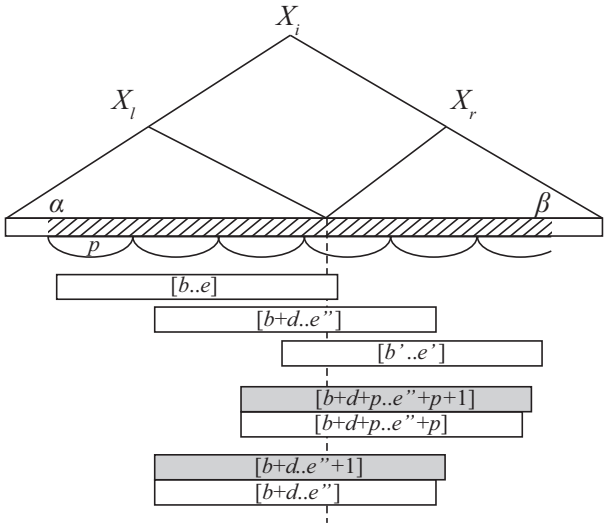

Figure 6.4: If $[b+d+p..e''+p] \notin \mathcal{F}(X_i)$ due to $X_i[b+d+p..e''+p+1] \in \mathit{Prefix}(\mathcal{S})$, $X_i[b+d+p..e''+p+1] = X_i[b+d..e''+1] \in \mathit{Prefix}(\mathcal{S})$ holds. This contradicts $[b+d..e''] \in \mathcal{F}(X_i)$.

$X_i[b+d+p..e']$ is a prefix of $X_i[b+d..e'']$ and in $\mathit{Prefix}(\mathcal{S})$, which implies that $[b'..e']$ is not an f-interval. Assume on the contrary that $[b+d+p..e''+p] \notin \mathcal{F}(X_i)$, i.e., at least one of $X_i[b+d+p..e''+p+1] \in \mathit{Prefix}(\mathcal{S})$ or $X_i[b+d+p-c..e''+p] \in \mathit{Prefix}(\mathcal{S})$ with some $c > 0$ holds. If $X_i[b+d+p..e''+p+1] \in \mathit{Prefix}(\mathcal{S})$, we get $X_i[b+d+p..e''+p+1] = X_i[b+d..e''+1] \in \mathit{Prefix}(\mathcal{S})$, which contradicts that $[b+d..e'']$ is an f-interval (see also Figure 6.4). If $X_i[b+d+p-c..e''+p] \in \mathit{Prefix}(\mathcal{S})$, we consider two cases: When $c > p$, we get $[b+d..e''] \subset [b+d+p-c..e''+p]$, which contradicts that $[b+d..e''] \in \mathcal{F}(X_i)$. When $c \leq p$, we get $X_i[b+d+p-c..e''+p] = X_i[b+d-c..e''] \in \mathit{Prefix}(\mathcal{S})$, a contradiction. Therefore $[b+d..e''] \in \mathcal{F}(X_i) \Longrightarrow [b+d+p..e''+p] \in \mathcal{F}(X_i)$ holds.

Next we show that $[b+d..e''] \in \mathcal{F}(X_i) \Longleftarrow [b+d+p..e''+p] \in \mathcal{F}(X_i)$. Note that $e < e''$, since otherwise $X_i[b..e''] = X_i[b+p..e''+p]$ is in $\mathit{Prefix}(\mathcal{S})$, which implies that $[b+d+p..e''+p]$ is not in $\mathcal{F}(X_i)$. Assume on the contrary that $[b+d..e''] \notin \mathcal{F}(X_i)$, i.e., at least one of $X_i[b+d..e''+1] \in \mathit{Prefix}(\mathcal{S})$ or $X_i[b+d-c..e''] \in \mathit{Prefix}(\mathcal{S})$ with some $c > 0$ holds. If $X_i[b+d..e''+1] \in \mathit{Prefix}(\mathcal{S})$, we get $X_i[b+d..e''+1] = X_i[b+d+p..e''+p+1] \in \mathit{Prefix}(\mathcal{S})$, which contradicts that $[b+d+p..e''+p]$ is an f-interval. If $X_i[b+d-c..e''] \in \mathit{Prefix}(\mathcal{S})$, we consider two cases: When $c > d$, we get $[b..e] \subset [b+d-c..e'']$, which contradicts that $[b..e] \in \mathcal{F}(X_i)$. When $c \leq d$, we get $X_i[b+d-c..e''] = X_i[b+d+p-c..e''+p] \in \mathit{Prefix}(\mathcal{S})$, a contradiction. Therefore $[b+d..e''] \in \mathcal{F}(X_i) \Longleftarrow [b+d+p..e''+p] \in \mathcal{F}(X_i)$ holds. $\square$

Lemma 39 implies that f-intervals subsumed by the f-rich run are cyclic except for the ones starting from the first period and the last f-interval in the run. Moreover, the next lemma shows that the number of f-intervals subsumed by the f-rich run and starting with an arbitrary period is at most n.

Lemma 40 *Let $X_i \to X_l X_r$ be any variable in $\mathcal{S}$. Let $[b..e]$ and $[b'..e']$ be the first and the last f-intervals subsumed by a run $[\alpha..\beta]$ with period p, respectively. For any d with $0 \leq d < d + p < b' - b$, $|\{[b''..e''] \mid [b''..e''] \in \mathcal{F}(X_i), b + d \leq b'' < b + d + p\}| \leq n$.*

Proof. Assume on the contrary that $|\{[b''..e''] \mid [b''..e''] \in \mathcal{F}(X_i), b+d \leq b'' < b+d+p\}| > n$. By the pigeon hole principle, there exists X_j such that $X_i[b_1..e_1]$ and $X_i[b_2..e_2]$ are prefixes of X_j, where $[b_1..e_1], [b_2..e_2] \in \mathcal{F}(X_i)$ with $b+d \leq b_1 < b_2 < b+d+p$. Also, recall that from the definition of f-rich run, $[|X_l| - p + 1..|X_l| + p]$ is covered by an f-interval, and the length of any f-interval subsumed by the f-rich run is longer than p. Consider $u = X_i[b_1..b_2 + p - 1]$ and observe that p and $b_2 - b_1 (< p)$ are both periods of u. Then, it follows from the periodicity lemma that $\gcd(b_2 - b_1, p)$ is a period of u which contradicts that p is the smallest period of the f-rich run. □

In light of Lemma 39 we consider storing cyclic f-intervals in a different way from the naive list of $\mathcal{F}(X_i)$. Since information of f-intervals for one period is enough to compute failure function for any state within the cyclic part, it can be stored in an $O(n)$-size list $L_c(X_i)$ by Lemma 40. Let $L(X_i)$ denote the list storing $\mathcal{F}(X_i)$ other than cyclic f-intervals. Note that $L(X_i)$ includes $O(n)$ f-intervals subsumed by the f-rich run but not in the cyclic part.

Lemma 41 *For any $X_i \to X_l X_r \in \mathcal{S}$, the size of $L(X_i)$ is bounded by $O(n \log N)$.*

Proof. Let X_j be any variable and let $c_0, \ldots, c_s$ ($c_0 < \cdots < c_s$) be the positions of $val(X_l)$ at which a suffix of $val(X_l)$ overlaps with a prefix of $val(X_j)$. We note that each c_k is a candidate for the beginning position of an f-interval of X_i. It follows from Lemma 2 that $c_0, \ldots, c_s$ can be partitioned into at most $O(\log |X_l|)$ disjoint segments such that each segment forms an arithmetic progression.

Let $0 \leq k < k' \leq s$ be integers such that $C = c_k, \ldots, c_{k'}$ is represented by one arithmetic progression. Let d be the step of C, i.e., $c_{k'} = c_{k'-1} + d = \cdots = c_k + (k' - k)d$.

We show that if more than two elements of C are related to the beginning positions of f-intervals of X_i, the f-rich run subsumes all those f-intervals but the last one.

Suppose that for some $k \leq h_1 < h_2 < h_3 \leq k'$, $c_{h_1}, c_{h_2}, c_{h_3} \in C$ are corresponding to f-intervals, namely, $[c_{h_1}..e], [c_{h_2}..e'], [c_{h_3}..e''] \in \mathcal{F}(X_i)$ with $e - c_{h_1} + 1 = \mathsf{LCP}(X_i[c_{h_1}..|X_i|], X_j)$, $e' - c_{h_2} + 1 = \mathsf{LCP}(X_i[c_{h_2}..|X_i|], X_j)$ and $e'' - c_{h_3} + 1 = \mathsf{LCP}(X_i[c_{h_3}..|X_i|], X_j)$. It is clear that d is the smallest period of $X_i[c_{h_1}..|X_l|]$ and $|X_l| - c_{h_1} + 1 > 2d$. Let β be the largest position of $val(X_i)$ such that $X_i[c_{h_1}..\beta]$ has period d, i.e., there is a run $[\alpha, \beta]$ with $\alpha \leq c_{h_1} < |X_l| < \beta$. Let β' be the largest position of $val(X_j)$ such that $X_j[1..\beta']$ has period d.

- If $\beta < e''$. Note that this happens only when $\beta - c_{h_3} + 1 = \beta'$. Consequently, $\mathsf{LCP}(X_i[c_{h_1}..|X_i|], X_j) = \mathsf{LCP}(X_i[c_{h_2}..|X_i|], X_j) = \beta'$.
- If $\beta \geq e''$. It is clear that $\beta' < e'' - c_{h_2} + 1$, since otherwise $[c_{h_3}..e'']$ would be contained in $[c_{h_2}..e']$. Then, $\mathsf{LCP}(X_i[c_{h_1}..|X_i|], X_j) = \mathsf{LCP}(X_i[c_{h_2}..|X_i|], X_j) = \beta'$.

In either case $X_i[c_{h_1}..e] = X_i[c_{h_2}..e'] = X_j[1..\beta']$ holds, which means that except for at most one f-interval $[c..e]$ satisfying $\beta < e$ the others are all subsumed by the f-rich run $[\alpha..\beta]$.

Since in each segment there are at most two f-intervals which are not subsumed by the f-rich run, the number of such f-intervals can be bounded by $O(\log N)$. Considering every variable X_j, we can bound the size of $L(X_i)$ by $O(n \log N)$. □

In light of Lemmas 39 and 41 we get the next lemma.

Lemma 42 *An $O(n^2 \log N)$-size representation of the failure function f can be constructed in $O(n^3 \log n \log N)$ time using $O(n^2 \log N)$ space so that given reference-pair of any state q, a reference-pair of the state $f(q)$ can be computed in $O(\log n)$ time.*

Proof. Karpinski et al. considered in [37] a compressed overlap table OV for an SLP of size n such that for any pair of variables X and Y, $OV(X, Y)$ contains $O(\log N)$-size representation of overlaps between suffixes of $val(X)$ and prefixes of $val(Y)$. They showed how to compute OV in $O(n^3 \log n \log N)$ time. Actually, their algorithm can be extended to compute $L(X_i)$ and $L_c(X_i)$ for all variable $X_i \in \mathcal{S}$ in $O(n^3 \log n \log N)$ time (see Subsection 6.1.3 for the details). From Lemma 39 and Lemma 41, the total size for $L(X_i)$ and $L_c(X_i)$ for all variable $X_i \in \mathcal{S}$ is bounded by $O(n^2 \log N)$.

Using $L(X_i)$ and $L_c(X_i)$, we can compute $f(q)$ for any state $q = \langle X_i, h\rangle$ in $O(\log n)$ time. If q is not in cyclic part of f-intervals, we conduct binary search on $L(X_i)$, otherwise on $L_c(X_i)$ with proper offset. It takes $O(\log(n \log N)) = O(\log n)$ time. □

6.1.3 Efficient Construction of Failure Function

Here we show that $L_c(X_i)$ and $L(X_i)$ for all $1 \leq i \leq n$ can be computed in $O(n^3 \log n \log N)$ time. Let $X_i \to X_l X_r$. Let an f-interval of X_i w.r.t. X_j is a maximal element in the set $\{[b..e] \mid 1 < b \leq |X_l| < e \leq |X_i|, X_i[b..e] \in \mathit{Prefix}(X_j)\}$ with respect to the set inclusion relation $\subseteq$. The f-interval sequence of X_i w.r.t. X_j, denoted $\mathcal{F}(X_i, X_j)$, is defined to be the sequence $\{[b_k..e_k]\}_{k=1}^{s}$ of all f-intervals of X_i w.r.t. X_j, arranged in the increasing order of b_k. For any $1 \leq i \leq n$, we compute $\mathcal{F}(X_i)$ by first computing $\mathcal{F}(X_i, X_j)$ for all $1 \leq j \leq n$, and then we build $L_c(X_i)$ and $L(X_i)$.

We use the compressed overlap table and *FirstMismatch*. The compressed overlap table OV for SLP of size n, introduced by Karpinski [37], is $n \times n$ table, where for any pair of variables X and Y, $OV(X, Y)$ contains $O(\log N)$ arithmetic progressions which represent the overlaps between suffixes of $val(X)$ and prefixes of $val(Y)$. Thus the space requirement of OV is $O(n^2 \log N)$. For variables X, Y and integer k, let $\mathit{FirstMismatch}(X, Y, k) = \min\{i > 0 : X[|X| - k + i] \neq Y[i]\}$.

The following results are known:

Lemma 43 ([37]) *The compressed overlap table OV for a given SLP of size n can be computed in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space.*

Lemma 44 ([37]) *Given the overlap table OV for an SLP $\mathcal{S}$ of size n, for any pair of variables X and Y of $\mathcal{S}$ and integer k, $\mathit{FirstMismatch}(X, Y, k)$ can be computed in $O(n \log n)$ time.*

Let $X_i \to X_l X_r$. Since an f-interval of X_i w.r.t. X_j accompanies an overlap between a suffix of X_l and a prefix of X_j, $\mathcal{F}(X_i, X_j)$ can be computed from $OV(X_l, X_j)$, i.e., we consider extending prefix matches of X_j in X_l presented in $OV(X_l, X_j)$ to X_r and computing prefix matches of X_j in X_i that cross the boundary of X_l and X_r. Although there could be exponential number of overlaps to consider which are presented in arithmetic progressions, similar techniques to compute the overlap table can be used and each arithmetic progression can be processed by constant number of *FirstMismatch* queries.

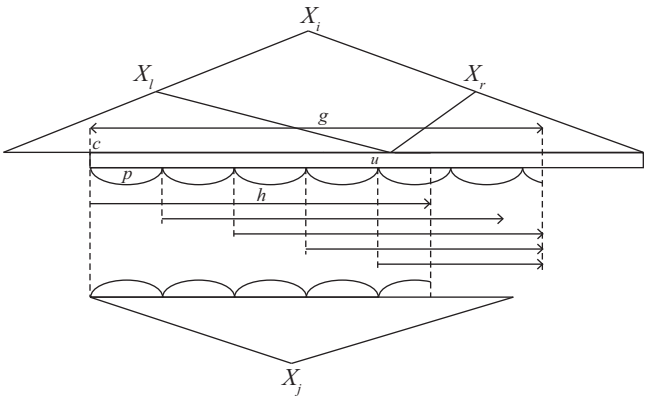

Figure 6.5: The right arrows illustrate the values of $\mathsf{LCP}(X_i[c + d'p..|X_i|], X_j)$. In this picture $d = 2$.

Take one arithmetic progression from $OV(X_l, X_j)$ and consider suffixes of X_i that start from $c, c + p, \ldots, c + kp$, i.e., $X_l[c + k'p..|X_l|]$ is a prefix of X_j for any $0 \leq k' \leq k$. Let u be the suffix of X_i starting from c. Let g be the length of the longest prefix of u having period p, and let h be the length of the longest prefix of u that is also a prefix X_j. We can compute g and h in $O(n \log n)$ time by asking $\mathit{FirstMismatch}(X_i, X_r, |X_r| + p)$ and $\mathit{FirstMismatch}(X_i, X_j, |X_i| - c + 1)$, respectively. From the periodicity that can be seen in the prefixes of u and X_j, we are able to know that

$$\mathsf{LCP}(X_i[c + d'p..|X_i|], X_j) = \begin{cases} h & \text{if } 0 \leq d' < d, \\ g - d'p & \text{if } d < d' \leq k, \end{cases}$$

where d is the minimum integer such that $h + dp \geq g$ with $0 \leq d \leq k$ if such exists, or otherwise $d = k + 1$ (see also Figure 6.5). Also $\mathsf{LCP}(X_i[c + dp..|X_i|], X_j) \geq g - dp$.

Since $\mathsf{LCP}(X_i[c + dp..|X_i|], X_j) \geq g - dp > g - (d + 1)p \geq g - d'p = \mathsf{LCP}(X_i[c + d'p..|X_i|], X_j)$ for any $d < d' \leq k$, $\mathcal{F}(X_i, X_j)$ cannot contain an f-interval starting from $c + d'p$ with $d < d' \leq k$. Then, it suffices to consider f-intervals starting from $c + d'p$ with $0 \leq d' \leq d$. Note that when $d \geq 2$ the f-intervals w.r.t. X_j corresponding to $c + d'p$ with $0 \leq d' < d$ should be subsumed by the f-rich run, and are represented in $O(1)$ space. When $d \geq 2$, we treat the intervals corresponding to $c + d'p$ with $0 \leq d' < d$ all together as a group-candidate. Let us call an interval not belonging to a group-candidate as a solo-candidate.

We now consider building $\mathcal{F}(X_i, X_j)$. We process arithmetic progressions of $OV(X_l, X_j)$ in increasing order of positions while maintaining a tentative list of $\mathcal{F}(X_i, X_j)$. Note

that the list can be maintained in $O(\log N)$ space since for each arithmetic progression we discover at most two solo-candidates and one group-candidate. When solo-candidates/group-candidates are discovered, we add them to the list and update so that the list contains maximal elements w.r.t. the set inclusion relation $\subseteq$. For a solo-candidate the maintenance can be easily done in $O(\log N)$ time. For a group-candidate, it follows from the proof of Lemma 38 that group-candidate with a small period is completely included by an f-interval in group-candidate with a bigger period, and hence, it suffices to consider the group-candidate with the biggest period. Therefore, by processing $O(\log N)$ arithmetic progressions of $OV(X_l, X_j)$, we can build $\mathcal{F}(X_i, X_j)$ in $O(n \log n \log N + \log^2 N) = O(n \log n \log N)$ time.

Finally, we can build $\mathcal{F}(X_i)$ of size $O(n \log N)$ by merging $\mathcal{F}(X_i, X_j)$ for all $1 \leq j \leq n$ in $O(n^2 \log n \log N + n \log^2 N) = O(n^2 \log n \log N)$ time, and obtain $L_c(X_i)$ and $L(X_i)$. Hence, $L_c(X_i)$ and $L(X_i)$ for all $1 \leq i \leq n$ can be computed in $O(n^3 \log n \log N)$ time.

6.1.4 Compact Representation of Output Function

Lemma 45 *An $O(nm)$-size representation of the output function λ can be computed in $O(n^3 \log n)$ time and $O(n^2)$ space so that given any state $q = \langle X_i, h \rangle$ we can compute $\lambda(q)$ in $O(height(X_i) + m)$ time.*

Proof. First we construct a tree with nodes $\Pi \cup \{\varepsilon\}$ such that for any $p \in \Pi_{\langle \mathcal{S}, m \rangle}$ the parent of p is the longest element of $\Pi_{\langle \mathcal{S}, m \rangle} \cup \{\varepsilon\}$ which is also a suffix of p. The tree can be constructed in $O(n^3 \log n)$ time in a similar way to the construction of the compact g-trie. Note that $\lambda(q)$ can be computed by detecting the longest member p of $\Pi_{\langle \mathcal{S}, m \rangle}$ which is also a suffix of $X_i[1..h]$, and outputting all patterns on the path from p to the root of the tree. In addition, we compute in $O(n^3)$ time a table of size $O(nm)$ such that for any pair of $p \in \Pi_{\langle \mathcal{S}, m \rangle}$ and variable X_j the table has $Occ^{\xi}(p, X_j)$ in a form of one arithmetic progression.

Now we show how to compute the longest member of $\Pi_{\langle \mathcal{S}, m \rangle}$ which is also a suffix of $X_i[1..h]$. We search for it in descending order of pattern length. We use three variables p', i' and h', which are initially set to the longest pattern in $\Pi_{\langle \mathcal{S}, m \rangle}$, i and h, respectively. We omit the case when $|p'| = 1$ or $|p'| > h$ since it is trivial. If the end position of $X_i[1..h]$ is contained in $X_{r(i')}$ and $|p'| > h' - |X_{\ell(i')}|$, using arithmetic progression of $Occ^{\xi}(p', X_{i'})$, we can check if p' is a suffix of $X_i[1..h]$ or not in constant time by simple arithmetic

operations. If the above condition does not hold, we traverse the derivation tree of $X_{i'}$ toward the end position of $X_i[1..h]$ updating i' and h' properly until meeting the above situation, where h' is updated to be the length of the overlapped string between $X_{i'}$ and $X_i[1..h]$.

It is not difficult to see that the total time is $O(height(X_i) + m)$. □

6.1.5 Main Result on Compressed AC Automata

We show Theorem 6.

Proof. By Lemma 35, an $O(n)$-size representation of the g-trie can be obtained in $O(n^3 \log n)$ time and $O(n^2)$ space. By Lemma 42, an $O(n^2 \log N)$-size representation of the failure function can be obtained in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space. By Lemma 45, an $O(nm)$-size representation of the output function can be obtained in $O(n^3 \log n)$ time and $O(n^2)$ space. We also build an $O(n^2)$-size data structure to conduct the bidirectional conversion between a state on the g-trie and its reference-pair (see Lemma 36). Thus, the space occupancy of our compressed automaton is $O(n^2 \log N)$ which is dominated by the representation of the failure function. While pattern matching, the computations on the compact g-trie, the failure function and the output function require $O(\log N)$, $O(\log n)$ and $O(height(X_i) + m)$ amortized time per character, respectively. Note that the failure function is called $O(1)$ times per character. Therefore we get the statement. □

We note that when $m = 1$, the output function of Lemma 45 is not needed since it is enough to report the occurrence of X_n when we reach there. Hence, the following Corollary holds.

Corollary 1 *For an SLP $\mathcal{S}$ of size n representing string T of length N, it is possible to build, in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space, an $O(n^2 \log N)$-size compressed automaton that recognizes all occurrences of T within an arbitrary string with $O(\log N)$ amortized running time per character.*

6.2 Compact Representation of MP Automata for SLPs

In this section, we show Theorem 7. Note that MP automaton is identical to an uncompressed AC-automaton for a single pattern. See the definition in Chapter 2. Since the MP automaton consists of goto and failure functions, we consider the compact representations by the following lemmas.

Lemma 46 (Goto function) *Given an SLP $\mathcal{S}$ of size n that represents a string T of length N, it is possible to build a data structure in $O(n)$ time using $O(n)$ space, so that the value $g(q, a)$ can be determined in $O(\log N)$ time for any state q and any symbol a.*

Proof. Directly from Lemma 10. □

Lemma 47 (Failure function) *Given an SLP $\mathcal{S}$ of size n that represents a string T of length N, it is possible to build a data structure of size $O(n \log N)$ in $O(n^3 \log n \log N)$ time using $O(n^2 \log N)$ space, so that the value $f(q)$ can be determined in $O(height(\mathcal{S}))$ time for any state $q \in Q$ with $q \neq 0$.*

Proof. Similar to Subsection 6.1.2, we realize the failure transition by considering f-intervals for each variable X_i, but we slightly modify the definition and how to use it since every transition state of failure function represents a prefix of T. An f-interval of X_i ($X_i \to X_l X_r$) is a maximal element in the set $\{[b..e] \mid 1 < b \leq |X_l| < e \leq |X_i|, X_i[b..e] \in \mathit{Prefix}(T)\}$ in the set inclusion relation $\subseteq$. Since such f-intervals hold Lemmas 38 and 39, they can be split into non-cyclic part $L(X_i)$ and cyclic part $L_c(X_i)$. Note that the space requirements for $L(X_i)$ and $L_c(X_i)$ are $O(\log N)$ and $O(1)$, respectively, since f-intervals are defined on $\mathit{Prefix}(T)$ instead of $\mathit{Prefix}(\mathcal{S})$ in Subsection 6.1.2. In a similar way to the construction of f-intervals for compressed AC automata, we can build $L(X_i)$ and $L_c(X_i)$ for all variables in $\mathcal{S}$ in $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space. We remark that the size of the data structure is $O(n \log N)$, and we can build it in $O(n^2 \log n \log N)$ time if the compressed overlap table for $\mathcal{S}$ is computed in advance.

Now we show how to calculate $f(q)$ using f-intervals. What we want to do is to find the leftmost f-interval that covers position q. Firstly, we conduct binary search on the derivation tree of $\mathcal{S}$ starting from the root and searching for the shallowest node such that its right node contains q and its rightmost f-interval covers q. If such a node is found,

we can find, in the f-intervals of the node, the leftmost f-interval that covers q, taking $O(\log \log N)$ time. If such a node is not found, it means that $f(q) < 2$. Then, $f(q) = 1$ if $T[q] = T[1]$, $f(q) = 0$ otherwise. The computational time is dominated by the search on the derivation tree which takes $O(\mathit{height}(\mathcal{S}))$ time. □

Theorem 7 follows from Lemmas 46 and 47.

6.3 Conclutions, Discussion and Future Work

In this chapter, we presented a new grammar compressed AC automaton and MP automaton.

Our method of Section 6.1 builds the goto and the failure functions of the AC automaton for the dictionary $\Pi_{\langle \mathcal{S},n \rangle}$ independently of m. This introduces redundant states and edges into the compact g-trie, and unnecessary failure transitions. Another possible solution to Problem 5 would be to divide the input DSLP into m SLPs and then build compressed MP automata, proposed in Section 6.2, for each of them.

Both solutions need $O(n^3 \log n \log N)$ time and $O(n^2 \log N)$ space for construction. There is a space–time tradeoff: The sizes of a compressed AC automaton and m compressed MP automata are $O(n^2 \log N)$ and $O(mn \log N)$, respectively. On the other hand, their amortized running time per character are $O(\mathit{height}(\mathcal{S}) + m)$ and $O(m\, \mathit{height}(\mathcal{S}))$, respectively.

As a future work, we are planning to remove the amortization factor in running time of our AC automaton. This realization is hard but we believe that it is not impossible.

Chapter 7

Conclusions

In this thesis, we presented some compressed data structures and algorithms based on grammar compressed strings for string processings.

In Chapter 3, we introduced signature encodings, showed that we can maintain the signature encoding of a dynamic string in compressed space, and we can construct efficiently the signature encoding of a string T for a given uncompressed string T, SLP representing T, or LZ77 factorization representing T. Especially, we showed that the signature encoding of T can be constructed for a given SLP representing T in $O(n \log N \log\log n \log^* N_{max})$ time and $O(n \log^* N_{max} + w)$ working space using an $O(m \log\log m)$ sorting algorithm, and also, we presented an application of signature encodings.

In Chapter 4, we showed that signature encodings support LCE queries in $O(\log N + \log \ell \log^* N_{max})$ time. Since signature encodings support *INSERT* and *DELETE* operations in $O((y + \log N \log^* N_{max}) f_{\mathcal{A}})$ time, we can solve the dynamic LCE problem by signature encodings, and also, we show that our LCE data structures improve previous some results using LCE queries.

In Chapter 5, we presented a new dynamic compressed index which supports find queries in $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* N_{max}(\log N + \log |P| \log^* N_{max}) + occ \log N)$ time, *INSERT* and *DELETE* operations in amortized $O((|Y| + \log N \log^* N_{max}) \log w \log N \log^* N_{max})$ time, and also, we presented a new LZ77 factorization algorithm which runs in $O(z \log w \log^3 N (\log^* N)^2)$ time and $O(w)$ working space.

In Chapter 6, we showed how to construct our compressed AC automata in $O(n^3 \log n \log N)$ time using $O(n^2 \log N)$ space, and also, we showed that the grammar compressed dictionary matching problem can be solved in $O(|T|(h+m))$ time by our compressed AC automata.

Bibliography

[1] P. K. Agarwal, L. Arge, S. Govindarajan, J. Yang, and K. Yi. Efficient external memory structures for range-aggregate queries. *Comput. Geom.*, 46(3):358–370, 2013.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[3] S. Alstrup, G. S. Brodal, and T. Rauhe. Dynamic pattern matching. Technical report, Department of Computer Science, University of Copenhagen, 1998.

[4] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In D. B. Shmoys, editor, *Proc. SODA*, pages 819–828. ACM/SIAM, 2000.

[5] J. Arz and J. Fischer. LZ-compressed string dictionaries. In A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, editors, *Proc. DCC*, pages 322–331. IEEE Computer Society, 2014.

[6] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[7] D. Belazzougui. Succinct dictionary matching with no slowdown. In A. Amir and L. Parida, editors, *Proc. CPM*, volume 6129 of *LNCS*, pages 88–100. Springer, 2010.

[8] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[9] P. Bille, A. R. Christiansen, P. H. Cording, and I. L. Gørtz. Finger search, random access, and longest common extensions in grammar-compressed strings. *CoRR*, abs/1507.02853, 2015.

[10] P. Bille, P. H. Cording, I. L. Gørtz, B. Sach, H. W. Vildhøj, and S. Vind. Fingerprints in compressed strings. In F. Dehne, R. Solis-Oba, and J. Sack, editors, *Proc. WADS*, volume 8037 of *LNCS*, pages 146–157. Springer, 2013.

[11] P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In F. Cicalese, E. Porat, and U. Vaccaro, editors, *Proc. CPM*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015.

[12] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[13] G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In S.-H. Teng, editor, *Proc. SODA*, pages 894–903. ACM/SIAM, 2008.

[14] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundam. Inform.*, 111(3):313–337, 2011.

[15] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, editors, *Proc. SPIRE*, volume 7608 of *LNCS*, pages 180–192. Springer, 2012.

[16] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1), 2007.

[17] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[18] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In A. V. Aho, editor, *Proc. STOC*, pages 365–372. ACM, 1987.

[19] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S. Woo. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms*, 9(1):100–121, 2011.

[20] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In N. Bansal and I. Finocchi, editors, *Proc. ESA*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015.

[21] J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Proc. LATIN*, pages 483–496, 2016.

[22] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In A. H. Dediu and C. Martín-Vide, editors, *Proc. LATA*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012.

[23] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In A. Pardo and A. Viola, editors, *Proc. LATIN*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014.

[24] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Approximate pattern matching in LZ77-compressed texts. *J. Discrete Algorithms*, 32:64–68, 2015.

[25] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Lacki, and P. Sankowski. Optimal dynamic strings. *CoRR*, abs/1511.02612, 2015.

[26] K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda. Restructuring compressed texts without explicit decompression. *CoRR*, abs/1107.2729, 2011.

[27] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. In D. D. Sleator, editor, *Proc. SODA*, pages 697–704. ACM/SIAM, 1994.

[28] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[29] Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[30] W. Hon, T. W. Lam, K. Sadakane, W. Sung, and S. Yiu. Compressed index for dynamic text. In *Proc. DCC*, pages 102–111. IEEE Computer Society, 2004.

[31] T. I. Longest common extensions with recompression. *CoRR*, abs/1611.05359, 2016.

[32] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015.

[33] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656(Part B):215–224, 2016.

[34] T. I, T. Nishimoto, S. Inenaga, H. Bannai, and M. Takeda. Compressed automata for dictionary matching. In S. Konstantinidis, editor, *Proc. CIAA*, volume 7982 of *LNCS*, pages 319–330. Springer, 2013.

[35] T. I, T. Nishimoto, S. Inenaga, H. Bannai, and M. Takeda. Compressed automata for dictionary matching. *Theor. Comput. Sci.*, 578:30–41, 2015.

[36] C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, editors. *SPIRE 2015*, volume 9309 of *LNCS*. Springer, 2015.

[37] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

[38] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305. IEEE Computer Society, 1999.

[39] Y. Lifshits. Processing compressed texts: A tractability border. In B. Ma and K. Zhang, editors, *Proc. CPM*, volume 4580 of *LNCS*, pages 228–240. Springer, 2007.

[40] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[41] S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.

[42] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009.

[43] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[44] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[45] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In A. Apostolico and J. Hein, editors, *Proc. CPM*, volume 1264 of *LNCS*, pages 1–11. Springer, 1997.

[46] J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

[47] J. I. Munro, Y. Nekrich, and J. S. Vitter. Dynamic data structures for document collections and graphs. In T. Milo and D. Calvanese, editors, *Proc. PODS*, pages 277–289. ACM, 2015.

[48] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

[49] C. G. Nevill-Manning, I. H. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In J. A. Storer and M. Cohn, editors, *Proc. DCC*, pages 244–253. IEEE Computer Society, 1994.

[50] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. In J. Holub and J. Zdárek, editors, *Proc. PSC*, pages 158–170. Czech Technical University in Prague, 2016.

[51] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In P. Faliszewski, A. Muscholl, and R. Niedermeier, editors, *Proc. MFCS*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[52] A. Policriti and N. Prezza. Fast online Lempel-Ziv factorization in compressed space. In Iliopoulos et al. [36], pages 13–20.

[53] N. Prezza and A. Policriti. Computing LZ77 in run-compressed space. *CoRR*, abs/1510.06257, 2015.

[54] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[55] S. C. Sahinalp and U. Vishkin. Data compression using locally consistent parsing. *Technical report, University of Maryland Department of Computer Science*, 1995.

[56] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *Proc. FOCS*, pages 320–328. IEEE Computer Society, 1996.

[57] H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009.

[58] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *J. Discrete Algorithms*, 8(2):241–257, 2010.

[59] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[60] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In J. Gudmundsson and J. Katajainen, editors, *Proc. SEA*, volume 8504 of *LNCS*, pages 338–350. Springer, 2014.

[61] Y. Takabatake, Y. Tabei, and H. Sakamoto. Online self-indexed grammar compression. In Iliopoulos et al. [36], pages 258–269.

[62] Y. Tanimura, T. I, H. Bannai, S. Inenaga, S. J. Puglisi, and M. Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In R. Grossi and M. Lewenstein, editors, *Proc. CPM*, volume 54 of *LIPIcs*, pages 1:1–1:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[63] P. Weiner. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11. IEEE Computer Society, 1973.

[64] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[65] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.

[66] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978.