Longest Common Extensions with Recompression[*]

Tomohiro I

Kyushu Institute of Technology, Fukuoka, Japan
tomohiro@ai.kyutech.ac.jp

Abstract

Given two positions i and j in a string T of length N, a *longest common extension (LCE) query* asks for the length of the longest common prefix between suffixes beginning at i and j. A compressed LCE data structure stores T in a compressed form while supporting fast LCE queries. In this article we show that the *recompression* technique is a powerful tool for compressed LCE data structures. We present a new compressed LCE data structure of size $O(z \lg(N/z))$ that supports LCE queries in $O(\lg N)$ time, where z is the size of Lempel-Ziv 77 factorization without self-reference of T. Given T as an uncompressed form, we show how to build our data structure in $O(N)$ time and space. Given T as a grammar compressed form, i.e., a straight-line program of size n generating T, we show how to build our data structure in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space. Our algorithms are deterministic and always return correct answers.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Longest Common Extension (LCE) queries, compressed data structure, grammar compressed strings, Straight-Line Program (SLP)

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.18

1 Introduction

Given two positions i and j in a text T of length N, a *longest common extension (LCE) query* $\mathsf{LCE}(i, j)$ asks for the length of the longest common prefix between suffixes beginning at i and j. Since LCE queries play a central role in many string processing algorithms (see text book [6] for example), efficient LCE data structures have been extensively studied. If we are allowed to use $O(N)$ space, optimal $O(1)$ query time can be achieved by, e.g., lowest common ancestor queries [1] on the suffix tree of T. However, $O(N)$ space can be too expensive nowadays as the size of strings to be processed becomes quite large. Thus, recent studies focus on more space efficient solutions.

Roughly there are three scenarios: Several authors have studied tradeoffs among query time, construction time and data structure size [19, 5, 4, 21]; In [18], Prezza presented in-place LCE data structures showing that the memory space for storing T can be replaced with an LCE data structure while retaining optimal substring extraction time; LCE data structures working on grammar compressed representation of T were studied in [7, 3, 2, 17].

In this article we pursue the third scenario, which is advantageous when T is highly compressible. In grammar compression, T is represented by a Context Free Grammar (CFG) that generates T and only T. In particular CFGs in Chomsky normal form, called Straight Line Programs (SLPs), are often considered as any CFG can be easily transformed into an SLP without changing the order of grammar size. Let $\mathcal{S}$ be an arbitrary SLP of size n generating T. Bille et al. [2] showed a Monte Carlo randomized data structure of $O(n)$ space that supports LCE queries in $O(\lg N + \lg^2 \ell)$ time, where ℓ is the answer to the LCE query.

[*] I was supported by JSPS KAKENHI Grant Number 16K16009.

Because their algorithm is based on Karp-Rabin fingerprints, the answer is correct w.h.p (with high probability). If we always expect correct answers, we have to verify fingerprints in preprocessing phase, spending either $O(N \lg N)$ time (w.h.p.) and $O(N)$ space or $O(\frac{N^2}{n} \lg N)$ time (w.h.p.) and $O(n)$ space.

For a deterministic solution, I et al. [7] proposed an $O(n^2)$-space data structure, which can be built in $O(n^2 h)$ time and $O(n^2)$ space from $\mathcal{S}$, and supports LCE queries in $O(h \lg N)$ time, where h is the height of $\mathcal{S}$. As will be stated in Theorem 2, we outstrip this result.

Our work is most similar to that presented in [17]. They showed that the signature encoding [15] of T, a special kind of CFGs that can be stored in $O(z \lg N \lg^* N)$ space, can support LCE queries in $O(\lg N + \lg \ell \lg^* N)$ time, where z is the size of LZ77 factorization[1] of T and $\lg^*$ is the iterated logarithm. The signature encoding is based on the localy consistent parsing technique, which determines the parsing of a string by local surrounding. A key property of the signature encoding is that any occurrence of the same substring of length ℓ in T is guaranteed to be compressed in almost the same way leaving only $O(\lg \ell \lg^* N)$ discrepancies in its surrounding. As a result, an LCE query can be answered by tracing the $O(\lg \ell \lg^* N)$ surroundings created over two occurrences of the longest common extension. Since the cost $O(\lg N)$ is needed anyway to traverse the derivation tree of height $O(\lg N)$ from the root, an LCE query is supported in $O(\lg N + \lg \ell \lg^* N)$ time.

In this article we show that CFGs created by the *recompression* technique exhibit a similar property that can be used to answer LCE queries in $O(\lg N)$ time. In recent years recompression has been proved to be a powerful tool in problems related to grammar compression [8, 9, 10, 13] and word equations [11, 12]. The main component of recompression is to replace some pairs in a string with variables of the CFG. Although we use global information (like the frequencies of pairs in the string) to determine which pairs to be replaced, the pairing itself is done very locally, i.e., “all” occurrences of the pairs are replaced regardless of contexts. Then we can show that recompression compresses any occurrence of the same substring in T in almost the same way leaving only $O(\lg N)$ discrepancies in its surrounding. This leads to an $O(\lg N)$-time algorithm to answer LCE queries, improving the $O(\lg N + \lg \ell \lg^* N)$-time algorithm of [17]. We also improve the data structure size from $O(z \lg N \lg^* N)$ of [17][2] to $O(z \lg(N/z))$.

In [17], the authors proposed efficient algorithms to build their LCE data structure from various kinds of input as summarized in Table 1. We achieve a better and cleaner complexity to build our LCE data structure from SLPs. This has a great impact on compressed string processing, in which we are to solve problems on SLPs without decompressing the string explicitly. For instance, we can apply our result to the problems discussed in Section 7 of [17] and immediately improve the results (other than Theorem 17). It should be noted that the data structures in [17] also support efficient text edit operations. We are not sure if our data structures can be efficiently dynamized.

Theorems 1 and 2 show our main results. Note that our data structure is a simple CFG of height $O(\lg N)$ on which we can simulate the traversal of the derivation tree in constant time per move. Thus, it naturally supports $\mathsf{Extract}(i, \ell)$ queries, which asks for retrieving the substring $T[i..i + \ell - 1]$, in $O(\lg N + \ell)$ time.

[1] Note that there are several variants of LZ77 factorization. In this article we refer to the one that is known as the *f-factorization without self-reference* as LZ77 factorization unless otherwise noted.

[2] We believe that the space complexities of [17] can be improved to $O(z \lg(N/z) \lg^* N)$ by using the same trick we use in Lemma 13.

Table 1 Comparison of construction time and space between ours and [17], where N is the length of T, $\mathcal{S}$ is an SLP of size n generating T, z is the size of LZ77 factorization of T, and $f_{\mathcal{A}}$ is the time needed for predecessor queries on a set of $z \lg N \lg^* N$ integers from an N-element universe.

Input	Construction time	Construction space	Reference
T	$O(N f_{\mathcal{A}})$	$O(z \lg N \lg^* N)$	Theorem 3 (1a) of [17]
T	$O(N)$	$O(N)$	Theorem 3 (1b) of [17]
$\mathcal{S}$	$O(n f_{\mathcal{A}} \lg N \lg^* N)$	$O(n + z \lg N \lg^* N)$	Theorem 3 (3a) of [17]
$\mathcal{S}$	$O(n \lg\lg n \lg N \lg^* N)$	$O(n \lg^* N + z \lg N \lg^* N)$	Theorem 3 (3b) of [17]
LZ77	$O(z f_{\mathcal{A}} \lg N \lg^* N)$	$O(z \lg N \lg^* N)$	Theorem 3 (2) of [17]
T	$O(N)$	$O(N)$	this work, Theorem 1
$\mathcal{S}$	$O(n \lg(N/n))$	$O(n + z \lg(N/z))$	this work, Theorem 2
LZ77	$O(z \lg^2(N/z))$	$O(z \lg(N/z))$	this work, Corollary 3

▶ **Theorem 1.** *Given a string T of length N, we can compute in $O(N)$ time and space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

▶ **Theorem 2.** *Given an SLP of size n generating a string T of length N, we can compute in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

Suppose that we are given the LZ77-compression of size z of T as an input. Since we can convert the input into an SLP of size $O(z \lg(N/z))$ [20], we can apply Theorem 2 to the SLP and get the next corollary.

▶ **Corollary 3.** *Given the LZ77-compression of size z of a string T of length N, we can compute in $O(z \lg^2(N/z))$ time and $O(z \lg(N/z))$ space a compressed representation of T of size $O(z \lg(N/z))$ that supports* Extract(i, ℓ) *in $O(\lg N + \ell)$ time and* LCE *queries in $O(\lg N)$ time.*

Technically, this work owes very much to two papers [10, 9]. For instance, our construction algorithm of Theorem 1 is essentially the same as the grammar compression algorithm [10], which produces an SLP of size $O(g^* \lg(N/g^*))$, where g^* is the smallest grammar size to generate T. Our contribution is in discovering the above mentioned property that can be used for fast LCE queries. Also, we use the property to upper bound the size of our data structure in terms of z rather than g^*. Since it is known that $z \le g^*$ holds [20], an upper bound in terms of z is preferable. The technical issues in our construction algorithm of Theorem 2 have been tackled in [9], in which the recompression technique is used to solve the fully-compressed pattern matching problems. However, we make some contributions on top of it: We give a new observation that simplifies the implementation and analysis of a component of recompression called BComp (see Section 4.1.2). Also, we achieve a better construction time $O(n \lg(N/n))$ than what we obtain by straightforwardly applying the analysis in [9]—$O(n \lg N)$.

2 Preliminaries

An alphabet Σ is a set of characters. A string over Σ is an element in Σ^*. For any string $w \in \Sigma^*$, $|w|$ denotes the length of w. Let ε be the empty string, i.e., $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any $1 \le i \le |w|$, $w[i]$ denotes the i-th character of w. For any $1 \le i \le j \le |w|$, $w[i..j]$

denotes the substring of w beginning at i and ending at j. For convenience, let $w[i..j] = \varepsilon$ if $i > j$. For any $0 \leq i \leq |w|$, $w[1..i]$ (resp. $w[|w| - i + 1..|w|]$) is called the prefix (resp. suffix) of w of length i. We say that a string x *occurs* at position i in w iff $w[i..i + |x| - 1] = x$. A substring $w[i..j] = c^d$ ($c \in \Sigma, d \geq 1$) of w is called a *block* iff it is a maximal run of a single character, i.e., $(i = 1 \vee w[i-1] \neq c) \wedge (j = |w| \vee w[j+1] \neq c)$.

The text on which LCE queries are performed is denoted by $T \in \Sigma^*$ with $N = |T|$ throughout this paper. We assume that Σ is an integer alphabet $[1..N^{O(1)}]$ and the standard word RAM model with word size $\Omega(\lg N)$.

The size of our compressed LCE data structure is bounded by $O(z \lg(N/z))$, where z is the size of the LZ77 factorization of T defined as follows:

▶ **Definition 4** (LZ77 factorization)**.** The factorization $T = f_1 f_2 \cdots f_z$ is the LZ77 factorization of T iff the following condition holds: For any $1 \leq i \leq z$, let $p_i = |f_1 f_2 \cdots f_{i-1}| + 1$, then $f_i = T[p_i]$ if $T[p_i]$ does not appear in $T[1..p_i - 1]$, otherwise f_i is the longest prefix of $T[p_i..N]$ that occurs in $T[1..p_i - 1]$.

In this article, we deal with grammar compressed strings, in which a string is represented by a Context Free Grammar (CFG) generating the string only. In particular, we consider *Straight-Line Programs (SLPs)* that are CFGs in Chomsky normal form. Formally, an SLP that generates a string T is a triple $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D})$, where Σ is the set of characters (terminals), $\mathcal{V}$ is the set of variables (non-terminals), $\mathcal{D}$ is the set of deterministic production rules whose righthand sides are in $\mathcal{V}^2 \cup \Sigma$, and the last variable derives T.[3] Let $n = |\mathcal{V}|$. We treat variables as integers in $[1..n]$ (which should be distinguishable from Σ by having extra one bit), and $\mathcal{D}$ as an injective function that maps a variable to its righthand side. We assume that given any variable X we can access in $O(1)$ time the information on X, e.g., $\mathcal{D}(X)$. We refer to n as the size of $\mathcal{S}$ since $\mathcal{S}$ can be encoded in $O(n)$ space. Note that N can be as large as 2^{n-1}, and so, SLPs have a potential to achieve exponential compression.

We extend SLPs by allowing run-length encoded rules whose righthand sides are of the form X^d with $X \in \mathcal{V}$ and $d \geq 2$, and call such CFGs *run-length SLPs (RLSLPs)*. Since a run-length encoded rule can be stored in $O(1)$ space, we still define the size of an RLSLP by the number of variables.

Let us consider the derivation tree $\mathcal{T}$ of an RLSLP $\mathcal{S}$ that generates a string T, where we delete all the nodes labeled with terminals for simplicity. That is, every node in $\mathcal{T}$ is labeled with a variable. The height of $\mathcal{S}$ is the height of $\mathcal{T}$. We say that a sequence $C = v_1 \cdots v_m$ of nodes is a *chain* iff the nodes are all adjacent in this order, i.e., the beginning position of v_{i+1} is the ending position of v_i plus one for any $1 \leq i < m$. C is labeled with the sequence of labels of $v_1 \cdots v_m$. For any sequence $p \in \mathcal{V}^*$ of variables, let $val_{\mathcal{S}}(p)$ denote the string obtained by concatenating the strings derived from all variables in the sequence. We omit $\mathcal{S}$ when it is clear from context. We say that p generates $val(p)$. Also, we say that p *occurs* at position i iff there is a chain that is labeled with p and begins at i.

The next lemma, which is somewhat standard for SLPs, also holds for RLSLPs.

▶ **Lemma 5.** *For any RLSLP $\mathcal{S}$ of height h generating T, by storing $|val(X)|$ for every variable X, we can support* Extract(i, ℓ) *in $O(h + \ell)$ time.*

[3] We treat the last variable as the starting variable.

3 LCE data structure built from uncompressed texts

In this section, we prove Theorem 1 by showing that the RLSLP obtained by grammar compression algorithm [9] based on recompression can be used for fast LCE queries. In Subsection 3.1 we review recompression and introduce notation we use. In Subsection 3.2 we present a new characterization of recompression, which is a key to our contributions.

3.1 TtoG: Grammar compression based on recompression

In [9] Jeż proposed an algorithm TtoG to compute an RLSLP of T in $O(N)$ time.[4] Let TtoG(T) denote the RLSLP of T produced by TtoG. We use the term *letters* for variables introduced by TtoG. In particular, we often refer to an occurrence of a sequence of letters, for which the readers should recall the definition of an occurrence of a sequence of variables. Also, we use c (rather than X) to represent a letter.

TtoG consists of two different types of compression, BComp and PComp, which stand for Block Compression and Pair Compression, respectively.

- BComp: Given a string w over $\Sigma = [1..|w|]$, BComp compresses w by replacing all blocks of length ≥ 2 with fresh letters. Note that BComp eliminates all blocks of length ≥ 2 in w. We can conduct BComp in $O(|w|)$ time and space (Lemma 6).
- PComp: Given a string w over $\Sigma = [1..|w|]$ that contains no block of length ≥ 2, PComp compresses w by replacing all pairs from $\acute{\Sigma}\grave{\Sigma}$ with fresh letters, where $(\acute{\Sigma}, \grave{\Sigma})$ is a partition of Σ, i.e., $\Sigma = \acute{\Sigma} \cup \grave{\Sigma}$ and $\acute{\Sigma} \cap \grave{\Sigma} = \emptyset$. We can deterministically compute in $O(|w|)$ time and space a partition of Σ by which at least $(|w| - 1)/4$ pairs are replaced (Lemma 7), and conduct PComp in $O(|w|)$ time and space (Lemma 8).

Let T_0 be a sequence of letters obtained by replacing every character c of T with a letter generating c. TtoG compresses T_0 by applying BComp and PComp by turns until the string gets shrunk into a single letter. Since PComp compresses a given string by a constant factor $3/4$, the height of TtoG(T) is $O(\lg N)$, and the total running time is bounded by $O(N)$.

In order to give a formal description we introduce some notation below. TtoG transforms level by level T_0 into strings, $T_1, T_2, \ldots, T_{\hat{h}}$, where $|T_{\hat{h}}| = 1$. For any $0 \leq h \leq \hat{h}$, we say that h is the *level* of T_h. If h is even, the transformation from T_h to T_{h+1} is performed by BComp, and production rules of the form $c \to \ddot{c}^d$ are introduced. If h is odd, the transformation from T_h to T_{h+1} is performed by PComp, and production rules of the form $c \to \acute{c}\grave{c}$ are introduced. Let Σ_h be the set of letters appearing in T_h. For any even h $(0 \leq h < \hat{h})$, let $\ddot{\Sigma}_h$ denote the set of letters with which there is a block of length ≥ 2 in T_h. For any odd h $(0 \leq h < \hat{h})$, let $(\acute{\Sigma}_h, \grave{\Sigma}_h)$ denote the partition of Σ_h used in PComp of level h.

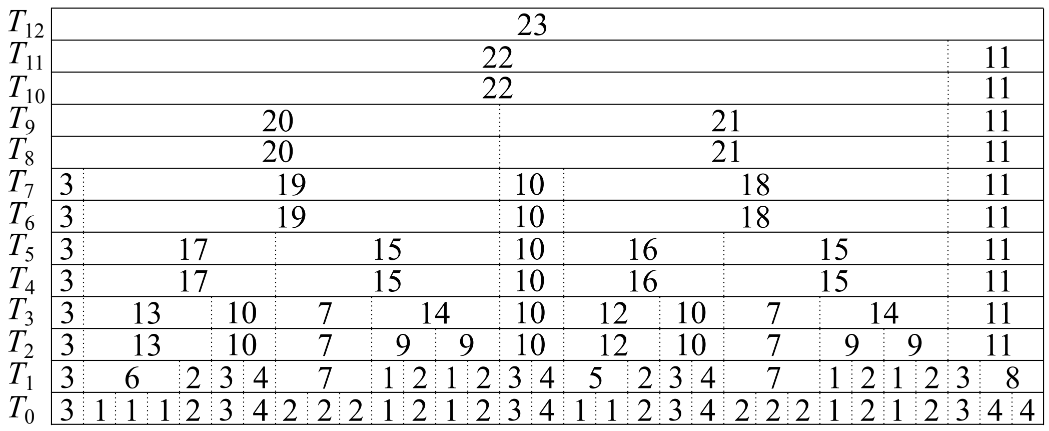
Figure 1 shows an example of how TtoG compresses T_0.

The following four lemmas show how to conduct BComp, PComp, and thus, TtoG, efficiently, which are essentially the same as respectively Lemma 2, Lemma 5, Lemma 6, and Theorem 1, stated in [9]. We give the proofs in Appendix for the sake of completeness.

▶ **Lemma 6.** *Given a string w over $\Sigma = [1..|w|]$, we can conduct* BComp *in $O(|w|)$ time and space.*

For any string $w \in \Sigma^*$ that contains no block of length ≥ 2, let $\mathsf{Freq}_w(c, \tilde{c}, 0)$ (resp. $\mathsf{Freq}_w(c, \tilde{c}, 1)$) with $c > \tilde{c} \in \Sigma$ denote the number of occurrences of $c\tilde{c}$ (resp. $\tilde{c}c$) in w. We

[4] Indeed, the paper shows how to compute an "SLP" of size $O(g^* \lg(N/g^*))$, where g^* is the smallest SLP size to generate T. In order to estimate the number of SLP's variables needed to represent run-length encoded rules, its analysis becomes much involved.



T_{12} 23
T_{11} 22 11
T_{10} 22 11
T_9 20 21 11
T_8 20 21 11
T_7 3 19 10 18 11
T_6 3 19 10 18 11
T_5 3 17 15 10 16 15 11
T_4 3 17 15 10 16 15 11
T_3 3 13 10 7 14 10 12 10 7 14 11
T_2 3 13 10 7 9 9 10 12 10 7 9 9 11
T_1 3 6 2 3 4 7 1 2 1 2 3 4 5 2 3 4 7 1 2 1 2 3 8
T_0 3 1 1 1 2 3 4 2 2 2 1 2 1 2 3 4 1 1 2 3 4 2 2 2 1 2 1 2 3 4 4

Figure 1 An example of how TtoG compresses T_0. Below we enumerate non-empty $\ddot{\Sigma}_h, \acute{\Sigma}_h, \grave{\Sigma}_h$ and production rules introduced in each level. From T_0 to T_1: $\ddot{\Sigma}_0 = \{1, 2, 4\}$, $\{5 \to 1^2, 6 \to 1^3, 7 \to 2^3, 8 \to 4^2\}$. From T_1 to T_2: $\acute{\Sigma}_1 = \{1, 3, 5, 6, 7\}$, $\grave{\Sigma}_1 = \{2, 4, 8\}$, $\{9 \to (1, 2), 10 \to (3, 4), 11 \to (3, 8), 12 \to (5, 2), 13 \to (6, 2)\}$. From T_2 to T_3: $\ddot{\Sigma}_2 = \{9\}$, $\{14 \to 9^2\}$. From T_3 to T_4: $\acute{\Sigma}_3 = \{3, 7, 12, 13\}$, $\grave{\Sigma}_3 = \{10, 14\}$, $\{15 \to (7, 14), 16 \to (12, 10), 17 \to (13, 10)\}$. From T_5 to T_6: $\acute{\Sigma}_5 = \{3, 10, 11, 16, 17\}$, $\grave{\Sigma}_5 = \{15\}$, $\{18 \to (16, 15), 19 \to (17, 15)\}$. From T_7 to T_8: $\acute{\Sigma}_7 = \{3, 10, 11\}$, $\grave{\Sigma}_7 = \{18, 19\}$, $\{20 \to (3, 19), 21 \to (10, 18)\}$. From T_9 to T_{10}: $\acute{\Sigma}_9 = \{11, 20\}$, $\grave{\Sigma}_9 = \{21\}$, $\{22 \to (20, 21)\}$. From T_{11} to T_{12}: $\acute{\Sigma}_{11} = \{22\}$, $\grave{\Sigma}_{11} = \{11\}$, $\{23 \to (22, 11)\}$.

refer to the list of non-zero $\mathsf{Freq}_w(c, \tilde{c}, \cdot)$ sorted in increasing order of c as the *adjacency list* of w. Note that it is a representation of the weighted directed graph in which there are exactly $\mathsf{Freq}_w(c, \tilde{c}, 0)$ (resp. $\mathsf{Freq}_w(c, \tilde{c}, 1)$) edges from c to $\tilde{c}$ (resp. from $\tilde{c}$ to c). Each occurrence of a pair in w is counted exactly once in the adjacency list. Then the problem of computing a good partition $(\acute{\Sigma}, \grave{\Sigma})$ of Σ reduces to maximum directed cut problem on the graph. Algorithm 1 is based on a simple greedy 1/4-approximation algorithm of maximum directed cut problem.

▶ **Lemma 7.** *Given the adjacency list of size m of a string $w \in \Sigma^*$, Algorithm 1 computes in $O(m)$ time a partition $(\acute{\Sigma}, \grave{\Sigma})$ of Σ such that the number of occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ in w is at least $(|w| - 1)/4$.*

▶ **Lemma 8.** *Given a string w over $\Sigma = [1..|w|]$ that contains no block of length ≥ 2, we can conduct* PComp *in $O(|w|)$ time and space.*

▶ **Lemma 9.** *Given a string T over $\Sigma = [1..N^{O(1)}]$, we can compute* $\mathsf{TtoG}(T)$ *in $O(N)$ time and space.*

3.2 Popped sequences

We give a new characterization of recompression, which is a key to fast LCE queries as well as obtaining the upper bound $O(z \lg(N/z))$ for the size of $\mathsf{TtoG}(T)$. For any substring w of T, we define the *Popped Sequence (PSeq)*, denoted by $PSeq(w)$, of w (formal definition is in the next paragraph). $PSeq(w)$ is a sequence of letters such that $val(PSeq(w)) = w$ and consists of $O(\lg N)$ blocks of letters. It is not surprising that any substring can be represented by $O(\lg N)$ blocks of letters because the height of $\mathsf{TtoG}(T)$ is $O(\lg N)$. The significant property of $PSeq(w)$ is that it occurs at "every" occurrence of w. A similar property has been observed in CFGs produced by locally consistent parsing and utilized for compressed indexes [14, 16]

Algorithm 1: How to compute a partition of Σ for PComp to compress w by $3/4$.

```
Input: Adjacency list of $w \in \Sigma^*$.
Output: $(\acute{\Sigma}, \grave{\Sigma})$ s.t. # occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ in $w$ is at least $(|w| - 1)/4$.
/* The information whether $c \in \Sigma$ is in $\acute{\Sigma}$ or $\grave{\Sigma}$ is written in the data
   space for $c$, which can be accessed in $O(1)$ time.                              */
1 $\acute{\Sigma} \leftarrow \grave{\Sigma} \leftarrow \emptyset$;
2 foreach $c \in \Sigma$ in increasing order do
3     if $\sum_{\tilde{c}\in\grave{\Sigma}} \mathsf{Freq}_w(c, \tilde{c}, \cdot) \geq \sum_{\tilde{c}\in\acute{\Sigma}} \mathsf{Freq}_w(c, \tilde{c}, \cdot)$ then
4         add $c$ to $\acute{\Sigma}$;
5     else
6         add $c$ to $\grave{\Sigma}$;
7 if # occurrences of pairs from $\acute{\Sigma}\grave{\Sigma}$ < # occurrences of pairs from $\grave{\Sigma}\acute{\Sigma}$ then
8     switch $\acute{\Sigma}$ and $\grave{\Sigma}$;
9 return $(\acute{\Sigma}, \grave{\Sigma})$;
```

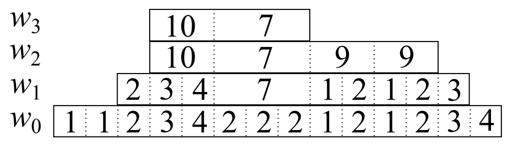


Figure 2 *PSeq* for $w_0 = [1, 1, 2, 3, 4, 2, 2, 2, 1, 2, 1, 2, 3, 4]$ under $\ddot{\Sigma}_h, \acute{\Sigma}_h, \grave{\Sigma}_h$ of Figure 1. At level 0, a block of 1 (resp. 4) is popped out from the leftend (resp. rightend) of w_0 because $1, 4 \in \ddot{\Sigma}_0$. At level 1, a letter 2 (resp. 3) is popped out from the leftend (resp. rightend) of w_1 because $2 \in \grave{\Sigma}_1$ and $3 \in \acute{\Sigma}_1$. At level 2, a block of 9 is popped out from the rightend of w_2 because $9 \in \ddot{\Sigma}_2$. At level 3, a letter 10 (resp. 7) is popped out from the leftend (resp. rightend) of w_3 because $10 \in \grave{\Sigma}_1$ and $7 \in \acute{\Sigma}_1$. Then, $\mathit{PSeq}(w_0) = [1, 1, 2, 10, 7, 9, 9, 3, 4]$. Observe that w_0 occurs twice in T_0 of Figure 1. and w_0, w_1, w_2 and w_3 are created over both occurrences. As a result, $\mathit{PSeq}(w_0)$ occurs everywhere w_0 occurs.

and a dynamic compressed LCE data structure [17]. For example, in [16, 17] the sequence having such a property is called the *common sequence* of w but its representation size is $O(\lg |w| \lg^* N)$ rather than $O(\lg N)$.

$\mathit{PSeq}(w)$ is the sequence of letters characterized by the following procedure. Let w_0 be the substring of T_0 that generates w. We consider applying BComp and PComp to w_0 exactly as we did to T but in each level we *pop* some letters out from both ends if the letters can be coupled with letters outside the scope. Formally, in increasing order of $h \geq 0$, we get w_{h+1} from w_h as follows:

- If h is even. We first pop out the leftmost and rightmost blocks of w_h if they are blocks of letter $c \in \ddot{\Sigma}_h$. Then we get w_{h+1} by applying BComp to the remaining string.
- If h is odd. We first pop out the leftmost letter and rightmost letter of w_h if they are letters in $\grave{\Sigma}_h$ and $\acute{\Sigma}_h$, respectively. Then we get w_{h+1} by applying PComp to the remaining string.

We iterate this until the string disappears. $\mathit{PSeq}(w)$ is the sequence obtained by concatenating the popped-out letters/blocks in an appropriate order, i.e., the order of the positions they occur. Note that for any occurrence of w the letters are compressed in the same way at least until they are popped out. Hence w_h is created for every occurrence of w and the occurrence of $\mathit{PSeq}(w)$ is guaranteed (see also Figure 2).

The next lemma formalizes the above discussion.

▶ **Lemma 10.** *For any substring w of T, PSeq(w) consists of $O(\lg N)$ blocks of letters. In addition, w occurs at position i iff PSeq(w) occurs at i.*

The next lemma and corollary are used to prove Lemmas 13 and 14.

▶ **Lemma 11.** *For any chain C whose label consists of m blocks of letters, the number of ancestor nodes of C is $O(m)$.*

▶ **Corollary 12.** *For any chain C corresponding to PSeq$(T[b..e])$ for some interval $[b..e]$, the number of ancestor nodes of C is $O(\lg N)$.*

▶ **Lemma 13.** *The size of* $\mathsf{TtoG}(T)$ *is* $O(z \lg(N/z))$.

Proof. We first show the bound $O(z \lg N)$ and later improve the analysis to $O(z \lg(N/z))$.

Let $f_1 \dots f_z$ be the LZ77 factorization of T. For any $1 \leq i \leq z$, let L_i be the set of letters used in the ancestor nodes of leaves corresponding to the prefix $f_1 f_2 \dots f_i$. Clearly $|L_1| = O(\lg N)$. For any $1 < i \leq z$, we estimate $|L_i \setminus L_{i-1}|$. Since f_i occurs in $f_1 \dots f_{i-1}$, we can see that the letters of *PSeq*(f_i) are in L_{i-1} thanks to Lemma 10. Let C_i be the chain corresponding to the occurrence $|f_1 \dots f_{i-1} + 1|$ of *PSeq*(f_i). Then, the letters in $L_i \setminus L_{i-1}$ are only in the labels of ancestor nodes of C_i. Since *PSeq*(f_i) consists of $O(\lg N)$ blocks of letters, $|L_i \setminus L_{i-1}|$ is bounded by $O(\lg N)$ due to Lemma 11. Therefore the size of $\mathsf{TtoG}(T)$ is $\sum_{i=1}^{z} |L_i \setminus L_{i-1}| = O(z \lg N)$.

In order to improve the bound to $O(z \lg(N/z))$, we employ the same trick that was used in [20, 9]. Let $h = 2\lg_{4/3}(N/z) = 2\lg_{3/4}(z/N)$. Recall that PComp compresses a given string by a constant factor $3/4$. Since PComp has been applied $h/2$ times until the level h, $|T_h| \leq N(3/4)^{h/2} = z$, and hence, the number of letters introduced in level $\geq h$ is bounded by $O(z)$. Then, we can ignore all the letters introduced in level $\geq h$ in the analysis of the previous paragraph, and by doing so, the bound $O(\lg N)$ of $|L_i \setminus L_{i-1}|$ is improved to $O(h) = O(\lg(N/z))$. This yields the bound $O(z \lg(N/z))$ for the size of $\mathsf{TtoG}(T)$. ◀

▶ **Lemma 14.** *Given* $\mathsf{TtoG}(T)$*, we can answer* $\mathsf{LCE}(i, j)$ *in* $O(\lg N)$ *time.*

Proof. We compute $\mathsf{LCE}(i, j)$ by matching the common sequence of letters occurring at i and j from left to right. First we traverse the derivation tree of $\mathsf{TtoG}(T)$ from the root down to the i-th and j-th leaves simultaneously while seeking the common block occurring at i and j. If there is no such block, $\mathsf{LCE}(i, j) = 0$, and we are done. Otherwise we stop at some internal nodes that contain the common block in their children. Let ℓ_1 be the length of the string generated by the block. Because $\mathsf{LCE}(i, j) \geq \ell_1$, we move on matching the next block by (possibly traversing up first and) traversing down to the $(i + \ell_1)$-th and $(j + \ell_1)$-th leaves. We iterate this procedure until we find no further common block. Then $\mathsf{LCE}(i, j) = \sum_{k=1}^{m} \ell_k$, where $\ell_1, \ell_2, \dots, \ell_m$ is the sequence of lengths of the common blocks we found.

Now we show that the above described algorithm runs in $O(\lg N)$ time. Note that it is bounded by the number of nodes we visit during the computation. In the light of Lemma 10, *PSeq*(w) occurs at both i and j, where w is the longest common prefix of two suffixes beginning at i and j. Let C_i (resp. C_j) be the chain that is labeled with *PSeq*(w) and begins at i (resp. j). Since the algorithm matches *PSeq*(w) or a succincter common sequence existing above C_i and C_j, we never go down below the parents of C_i or C_j during the computation. Hence, the number of visited nodes is bounded by the number of nodes that are ancestors of C_i or C_j, which is $O(\lg N)$ by Corollary 12. ◀

Theorem 1 is immediately from Lemmas 9, 5 and 14.

4 LCE data structure built from SLPs

In this section, we prove Theorem 2. Input is now an arbitrary SLP $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D})$ of size n generating T. Basically what we consider is to simulate TtoG on $\mathcal{S}$, namely, compute TtoG(T) without decompressing $\mathcal{S}$ explicitly. In Section 4.1, we present an algorithm SimTtoG that simulates TtoG in $O(n \lg^2(N/n))$ time and $O(n + z \lg(N/z))$ space. In Section 4.2, we present how to modify SimTtoG to get Theorem 2.

4.1 SimTtoG: Simulating TtoG on CFGs

We present an algorithm SimTtoG to simulate TtoG on $\mathcal{S}$. To begin with, we compute the CFG $\mathcal{S}_0 = (\Sigma_0, \mathcal{V}, \mathcal{D}_0)$ obtained by replacing, for all variables $X \in \mathcal{V}$ with $\mathcal{D}(X) \in \Sigma$, every occurrence of X in the righthand sides of $\mathcal{D}$ with the letter generating $\mathcal{D}(X)$. Note that Σ_0 is the set of terminals of $\mathcal{S}_0$, and $\mathcal{S}_0$ generates T_0. SimTtoG transforms level by level $\mathcal{S}_0$ into CFGs, $\mathcal{S}_1 = (\Sigma_1, \mathcal{V}, \mathcal{D}_1), \mathcal{S}_2 = (\Sigma_2, \mathcal{V}, \mathcal{D}_2), \ldots, \mathcal{S}_{\hat{h}} = (\Sigma_{\hat{h}}, \mathcal{V}, \mathcal{D}_{\hat{h}})$, where each $\mathcal{S}_h$ generates T_h. Namely, compression from T_h to T_{h+1} is simulated on $\mathcal{S}_h$. We can correctly compute the letters introduced in each level $h+1$ while modifying $\mathcal{S}_h$ into $\mathcal{S}_{h+1}$, and hence, we get all the letters of TtoG(T) in the end. We note that new "variables" are never introduced and the modification is done by rewriting righthand sides of the original variables.

Here we introduce the special formation of the CFGs $\mathcal{S}_h$ (it is a generalization of SLPs in a different sense from RLSLPs): For any $X \in \mathcal{V}$, $\mathcal{D}_h(X)$ consists of an "arbitrary number" of letters and at most "two" variables. More precisely, the following condition holds:

For any variable $X \in \mathcal{V}$ with $\mathcal{D}(X) = \acute{X}\grave{X}$, $\mathcal{D}_h(X)$ is either $w_1\acute{X}w_2\grave{X}w_3$, $w_1\acute{X}w_2$, $w_2\grave{X}w_3$ or w_2 with $w_1, w_2, w_3 \in \Sigma_h^*$, where $w_1 = w_3 = \varepsilon$ if X is not the starting variable.

As opposed to SLPs and RLSLPs, we define the size of $\mathcal{S}_h$ by the total lengths of righthand sides and denote it by $|\mathcal{S}_h|$.

4.1.1 PComp on CFGs

We firstly demonstrate that the adjacency list of T_h can be computed efficiently.

▶ **Lemma 15** (Lemma 6.1 of [10]). *For any odd h $(0 \leq h < \hat{h})$, the adjacency list of T_h, whose size is $O(|\mathcal{S}_h|)$, can be computed in $O(|\mathcal{S}_h| + n)$ time and space.*

Proof. For any variable $X \in \mathcal{V}$, let VOcc(X) denote the number of occurrences of the nodes labeled with X in the derivation tree of $\mathcal{S}$. It is well known that VOcc(X) for all variables can be computed in $O(n)$ time and space on the DAG representation of the tree.[5] Also, for any variable $X \in \mathcal{V}$, let LML(X) and RML(X) denote the leftmost letter and respectively rightmost letter of $val_{\mathcal{S}_h}(X)$. We can compute LML(X) for all variables in $O(|\mathcal{S}_h|)$ time by a bottom up computation, i.e., LML$(X) = $ LML(Y) if $\mathcal{D}_h(X)$ starts with a variable Y, and LML$(X) = w[1]$ if $\mathcal{D}_h(X)$ starts with a non-empty string w. In a completely symmetric way RML(X) can be computed in $O(|\mathcal{S}_h|)$ time.

Now observe that any occurrence i of a pair $\acute{c}\grave{c}$ in T_h can be uniquely associated with a variable X that is the label of the lowest node covering the interval $[i..i+1]$ in the derivation tree of $\mathcal{S}_h$ (recall that $\mathcal{S}_h$ generates T_h). We intend to count all the occurrences of pairs associated with X in $\mathcal{D}_h(X)$. For example, let $\mathcal{D}_h(X) = \acute{X}w_2\grave{X}$ with $w_2 \in \Sigma_h^*$. Then $\acute{c}\grave{c}$

[5] It is sufficient to compute VOcc(X) once at the very beginning of SimTtoG.

appears *explicitly* in w_2 or *crosses* the boundaries of $\acute{X}$ and/or $\grave{X}$. If $\acute{c}\grave{c}$ crosses the boundary of $\acute{X}$, $\mathsf{RML}(\acute{X})$ is $\acute{c}$ and $\grave{c}$ follows, i.e., $(w_2[1] = \grave{c}) \vee (w_2 = \varepsilon \wedge \mathsf{LML}(\grave{X}) = \grave{c})$. Using $\mathsf{RML}(\acute{X})$ and $\mathsf{LML}(\grave{X})$, we can compute in $O(|\mathcal{D}_h(X)|)$ time and space a $(|\mathcal{D}_h(X)| - 1)$-size multiset that lists all the explicit and crossing pairs in $\mathcal{D}_h(X)$. Each pair $\acute{c}\grave{c}$ with $\acute{c} > \grave{c}$ (resp. $\acute{c} < \grave{c}$) is listed by a quadruple $(\acute{c}, \grave{c}, 0, \mathsf{VOcc}(X))$ (resp. $(\grave{c}, \acute{c}, 1, \mathsf{VOcc}(X))$. $\mathsf{VOcc}(X)$ means that the pair has a weight $\mathsf{VOcc}(X)$ because the pair appears every time a node labeled with X appears in the derivation tree.

We compute such a multiset for every variable, which takes $O(|\mathcal{S}_h|)$ time and space in total. Next we sort the obtained list in increasing order of the first three integers in a quadruple. Note that the maximum value of letters is $O(z \lg(N/z))$ due to Lemma 13, and $O(z \lg(N/z)) = O(n^2)$ since $z \leq n$ and $\lg N \leq n$ hold. Thus the sorting can be done in $O(n)$ time and space by radix sort. Finally we can get the adjacency list of T_h by summing up weights of the same pair. The size of the list is clearly $O(|\mathcal{S}_h|)$. ◀

The next lemma shows how to implement PComp on CFGs:

▶ **Lemma 16.** *For any odd h $(0 \leq h < \hat{h})$, we can compute $\mathcal{S}_{h+1}$ from $\mathcal{S}_h$ in $O(|\mathcal{S}_h| + n)$ time and space. In addition, $|\mathcal{S}_{h+1}| \leq |\mathcal{S}_h| + 2n$.*

Proof. We first compute the partition $(\acute{\Sigma}_h, \grave{\Sigma}_h)$ of Σ_h, which can be done in $O(|\mathcal{S}_h| + n)$ time and space by Lemmas 15 and 7.

Given $(\acute{\Sigma}_h, \grave{\Sigma}_h)$, we can detect all the positions of the pairs from $\acute{\Sigma}_h\grave{\Sigma}_h$ in the righthands of $\mathcal{D}_h$, which are to be compressed. Some of the appearances of the pairs are explicit and the others are crossing. While explicit pairs can be compressed easily, crossing pairs need additional treatment. To deal with crossing pairs, we first *uncross* them by popping out every $\mathsf{LML}(Y) \in \grave{\Sigma}_h$ and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$ from $\mathit{val}_{\mathcal{S}_h}(Y)$ and popping them into the appropriate positions in the other rules. More precisely, we do the followings (for technical reason, do $\mathsf{PopInLet}$ first):

PopInLet. For any variable X, if $\mathcal{D}_h(X)[i] = Y \in \mathcal{V}$ with $i > 1$ ($i \geq 1$ if X is the starting variable) and $\mathsf{LML}(Y) \in \grave{\Sigma}_h$, replace the occurrence of Y with $\mathsf{LML}(Y)Y$; if $\mathcal{D}_h(X)[i] = Y \in \mathcal{V}$ with $i < |\mathcal{D}_h(X)|$ ($i \leq |\mathcal{D}_h(X)|$ if X is the starting variable) and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$, replace the occurrence of Y with $Y\mathsf{RML}(Y)$.

PopOutLet. For any variable X other than the starting variable, if $\mathcal{D}_h(X)[1] \in \grave{\Sigma}_h$, remove the first letter of $\mathcal{D}_h(X)$; and if $\mathcal{D}_h(X)[|\mathcal{D}_h(X)|] \in \acute{\Sigma}_h$, remove the last letter of $\mathcal{D}_h(X)$. In addition, if X becomes empty, we remove all the appearances of X in $\mathcal{D}_h$.

$\mathsf{PopOutLet}$ removes $\mathsf{LML}(Y) \in \grave{\Sigma}_h$ and $\mathsf{RML}(Y) \in \acute{\Sigma}_h$ from $\mathit{val}_{\mathcal{S}_h}(Y)$ (which can be a part of a crossing pair), and $\mathsf{PopInLet}$ introduces the removed letters into appropriate positions in $\mathcal{D}_h$ so that the modified $\mathcal{S}_h$ keeps to generate T_h. Notice that for each variable X the positions where letters popped in is at most two (four if X is the starting variable) and there is at least one variable that has no variables below, and hence, no letters popped in. Thus, the size of $\mathcal{S}_h$ increases at most $2n$. The uncrossing can be conducted in $O(|\mathcal{S}_h| + n)$ time.

Since all the pairs to be compressed become explicit, we can conduct BComp in $O(|\mathcal{S}_h| + n)$ time as follows. We scan righthand sides in $O(|\mathcal{S}_h|)$ time and list all the occurrences of pairs to be compressed. Each occurrence of pair $\acute{c}\grave{c} \in \acute{\Sigma}\grave{\Sigma}$ is listed by a triple $(\acute{c}, \grave{c}, p)$, where p is the pointer to the occurrence. Then we sort the list according to the pair of integers $(\acute{c}, \grave{c})$, which can be done in $O(|\mathcal{S}_h| + n)$ time and space by radix sort because $\acute{c}$ and $\grave{c}$ are $O(n^2)$. Finally, we replace each pair at position p with a fresh letter based on the rank of $(\acute{c}, \grave{c})$. ◀

4.1.2 BComp on CFGs

For any even h $(0 \leq h < \hat{h})$, BComp can be implemented in a similar way to PComp of Lemma 16. A block $T_h[b..e]$ of length ≥ 2 is uniquely associated with a variable X that is the label of the lowest node covering the interval $[b-1..e+1]$ in the derivation tree of $\mathcal{S}_h$ (if $b = 0$ or $e = |T_h|$, the block is associated with the starting variable). Here we take $[b-1..e+1]$ rather than $[b..e]$ to be sure that the block cannot extend outside the variable. Some blocks are explicitly written in $\mathcal{D}_h(X)$ and the others are crossing the boundaries of variables in $\mathcal{D}_h(X)$. The numbers of explicit blocks and crossing blocks in $\mathcal{D}_h$ is at most $|\mathcal{S}_h|$ and $2n$, respectively. The crossing blocks can be uncrossed in a similar way to uncrossing pairs. Then BComp can be done by replacing all the blocks with fresh letters on righthand sides of $\mathcal{D}_h$.

However here we have a problem. In order to give a unique letter to a block c^d, we have to sort the pairs of integers (c, d). Since d might be exponentially larger than $|\mathcal{S}_h| + n$, radix sort cannot be executed in $O(|\mathcal{S}_h| + n)$ time and space. In Section 6.2 of [10], Jeż showed how to solve this problem by tweaking the representation of lengths of long blocks, but its implementation and analysis are involved.[6]

We show in Lemma 17 our new observation, which leads to a simpler implementation and analysis of BComp. We say that a block c^d is *short* if $d = O(|\mathcal{S}_h| + n)$ and *long* otherwise. Also, we say that a variable is *unary* iff its righthand side consists of a single block.

▶ **Lemma 17.** *For any even h $(0 \leq h < \hat{h})$, a block $T_h[b..e] = c^d$ is short if it does not include a substring generated from a unary variable.*

Proof. Consider the derivation tree of $\mathcal{S}_h$ and the shortest path from $T_h[b]$ to $T_h[e]$. Let $X_1 X_2 \cdots X_{m'} \cdots X_m$ be the sequence of labels of internal nodes on the path, where $X_{m'}$ corresponds to the lowest common ancestor of $T_h[b]$ and $T_h[e]$. Since SLPs have no loops in the derivation tree, $X_1, \ldots, X_{m'}$ are all distinct. Similarly $X_{m'+1}, \ldots, X_m$ are all distinct. Since a unary variable is not involved to generate the block, it is easy to see that $d \leq \sum_{i=1}^{m} |\mathcal{D}_h(X_i)| \leq 2|\mathcal{S}_h|$ holds. ◀

Lemma 17 implies that most of blocks we find during the compression are short, which can be sorted efficiently by radix sort. If there is a long block in $\mathcal{D}_h$, an occurrence of a unary variable X must be involved to generate the block. Since BComp at level h pops out all the letters from X and removes the occurrences of X in $\mathcal{D}_h$, there are at most $2n$ long blocks in total. The number of long blocks can also be upper bounded by $2N/n$ with a different analysis based on the following fact:

▶ **Fact 18.** *If a substring of original text T generated from a long block overlaps with that generated from another long block, one substring must include the other, and moreover, the shorter block is completely included in "one" letter of the longer block. Hence the length of the substring of the longer block is at least n times longer than that of the shorter block.*

Let us consider the long blocks that generate substrings whose lengths are $[n^i..n^{i+1})$ for a fixed integer $i \geq 1$. By Fact 18, the substrings cannot overlap, and hence, the number of such long blocks is at most N/n^i. Therefore, the total number of long blocks is at most $\sum_{i \geq 1} N/n^i \leq 2N/n$. Thus we get the following lemma.

[6] Note that Section 6.2 of [10] also takes care of the case where the word size is $\Theta(\lg n)$ rather than $\Theta(\lg N)$. We do not consider the $\Theta(\lg n)$-bits model in this paper because using $\Theta(\lg N)$ bits to store the length of string generated by every letter is crucial for extract and LCE queries. However, we believe that our new observation stated in Lemma 17 will simplify the analysis for the $\Theta(\lg n)$-bits model, too.

▶ **Lemma 19.** *There are at most $O(\min(n, N/n))$ long blocks found during* SimTtoG.

By Lemma 19, we can employ a standard comparison-based sorting algorithm to sort all long blocks in $O(n \lg(\min(n, N/n)))$ time in total. In particular, BComp of one level can be implemented in the following complexities:

▶ **Lemma 20.** *For any even h $(0 \leq h < \hat{h})$, we can compute $\mathcal{S}_{h+1}$ from $\mathcal{S}_h$ in $O(|\mathcal{S}_h| + n + m \lg m))$ time and $O(|\mathcal{S}_h| + n)$ space, where m is the number of long blocks in $\mathcal{D}_h$. In addition, $|\mathcal{S}_{h+1}| \leq |\mathcal{S}_h| + 2n$.*

4.1.3 The complexities of SimTtoG

▶ **Theorem 21.** SimTtoG *runs in $O(n \lg^2(N/n))$ time and $O(n \lg(N/n))$ space.*

Proof. Using PComp and BComp implemented on CFGs (see Lemma 16 and 20), SimTtoG transforms level by level $\mathcal{S}_0$ into $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{\hat{h}}$. In each level, the size of CFGs can increase at most $2n$ by the procedure of uncrossing. Since $|\mathcal{S}_h| = O(n \lg N)$ for any h $(0 \leq h < \hat{h})$, we get the time complexity $O(n \lg^2 N)$ by simply applying Lemmas 16 and 20.

We can improve it to $O(n \lg^2(N/n))$ by a similar trick used in the proof of Lemma 13. At some level h' where $|T_{h'}|$ becomes less than n, we decompress $\mathcal{S}_{h'}$ and switch to TtoG, which transforms $T_{h'}$ into $T_{\hat{h}}$ in $O(n)$ time by Lemma 9. We apply Lemmas 16 and 20 only for h with $0 \leq h < h'$. Since $h' = O(\lg(N/n))$, $|\mathcal{S}_h| = O(n \lg(N/n))$ for any h $(0 \leq h < h')$. Hence, we get the time complexity $O(n \lg^2(N/n))$. The space complexity is bounded by the maximum size of CFGs $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{h'}$, which is $O(n \lg(N/n))$. ◀

4.2 GtoG: $O(n \lg(N/n))$-time recompression

We modify SimTtoG slightly to run in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space. The idea is the same as what has been presented in Section 6.1 of [10]. The problem of SimTtoG is that the sizes of intermediate CFGs $\mathcal{S}_h$ can grow up to $O(n \lg(N/n))$. If we can keep their sizes to $O(n)$, everything goes fine. This can be achieved by using two different types of partitions of Σ_h for PComp: One is for compressing T_h by a constant factor, and the other for compressing $|\mathcal{S}_h|$ by a constant factor (unless $|\mathcal{S}_h|$ is too small to compress). Recall that the former partition has been used in TtoG and SimTtoG, and the partition is computed from the adjacency list of T_h by Algorithm 1. Algorithm 1 can be extended to work on a set of strings by just inputting the adjacency list from a set of strings. Then, we can compute the partition for compressing $|\mathcal{S}_h|$ by a constant factor by considering the adjacency list from a set of strings in the righthand sides of $\mathcal{D}_h$. The adjacency list can be easily computed in $O(|\mathcal{S}_h| + n)$ time and space by modifying the algorithm described in the proof of Lemma 15: We just ignore the weight $\mathsf{VOcc}(X)$, i.e., use a unit weight 1 for every listed pair. Using the two types of partitions alternately, we can compress strings by a constant factor while keeping the sizes of the intermediate CFGs to $O(n)$.

We denote the modified algorithm by GtoG and the resulting RLSLP by GtoG($\mathcal{S}$). Note that GtoG($\mathcal{S}$) is not identical to TtoG(T) in general because the partitions used in GtoG depend on the input $\mathcal{S}$. Still the height of GtoG($\mathcal{S}$) is $O(\lg N)$ and the properties of *PSeq*s hold. Hence we can support LCE queries on GtoG($\mathcal{S}$) as we did on TtoG(T) by Lemma 14.

4.3 Proof of Theorem 2

Proof of Theorem 2. Let $\mathcal{S}$ be an input SLP of size n generating T. We compute GtoG($\mathcal{S}$) in $O(n \lg(N/n))$ time and $O(n + z \lg(N/z))$ space as described in Section 4.2. Since the height of $\mathsf{GtoG}(\mathcal{S})$ is $O(\lg N)$, we can support $\mathsf{Extract}(i, \ell)$ queries in $O(\lg N + \ell)$ time due to Lemma 5. $\mathsf{GtoG}(\mathcal{S})$ supports LCE queries in $O(\lg N)$ time in the same way as what was described in Lemma 14. ◀

References

1 Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005. `doi:10.1016/j.jalgor.2005.08.001`.

2 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *LIPIcs*, pages 36:1–36:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.FSTTCS.2016.36`.

3 Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Proceedings of the 13th International Symposium on Algorithms and Data Structures (WADS 2013)*, volume 8037 of *LNCS*, pages 146–157. Springer, 2013. `doi:10.1007/978-3-642-40104-6_13`.

4 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. `doi:10.1007/978-3-319-19929-0_6`.

5 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014. `doi:10.1016/j.jda.2013.06.003`.

6 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. `doi:10.1017/CBO9780511574931`.

7 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015. `doi:10.1016/j.ic.2014.09.009`.

8 Artur Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In Christoph Dürr and Thomas Wilke, editors, *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, volume 14 of *LIPIcs*, pages 136–147. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012. `doi:10.4230/LIPIcs.STACS.2012.136`.

9 Artur Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015. `doi:10.1016/j.tcs.2015.05.027`.

10 Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. `doi:10.1145/2631920`.

11 Artur Jeż. One-variable word equations in linear time. *Algorithmica*, 74(1):1–48, 2016. `doi:10.1007/s00453-014-9931-3`.

12 Artur Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4, 2016. `doi:10.1145/2743014`.

13 Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammar. In Ernst W. Mayr and Natacha Portier, editors, *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 445–457. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. `doi:10.4230/LIPIcs.STACS.2014.445`.

14 Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. Esp-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013. `doi:10.1016/j.jda.2012.07.009`.

15 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. `doi:10.1007/BF02522825`.

16 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference (PSC 2016)*, pages 158–170, 2016. URL: `http://www.stringology.org/event/2016/p14.html`.

17 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.MFCS.2016.72`.

18 Nicola Prezza. In-place longest common extensions, 2017. `arXiv:1608.05100v9`.

19 Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008. `doi:10.1007/978-3-540-92182-0_14`.

20 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1–3):211–222, 2003. `doi:10.1016/S0304-3975(02)00777-6`.

21 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In Roberto Grossi and Moshe Lewenstein, editors, *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *LIPIcs*, pages 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.1`.

A Appendix: Omitted proofs

A.1 Proof of Lemma 6

Proof. We first scan w in $O(|w|)$ time and list all the blocks of length ≥ 2. Each block c^d $(c \in \Sigma, d \geq 2)$ at position i is listed by a triple (c, d, i) of integers in Σ. Next we sort the list according to the pair of integers (c, d), which can be done in $O(|w|)$ time and space by radix sort. Finally, we replace each block c^d by a fresh letter based on the rank of (c, d). ◀

A.2 Proof of Lemma 7

Proof. In the foreach loop, we first run a 1/2-approximation algorithm of maximum "undirected" cut problem on the adjacency list, i.e., we ignore the direction of the edges here. For each c in increasing order, we greedily determine whether c is added to $\acute{\Sigma}$ or to $\grave{\Sigma}$ depending on $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot) \geq \sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$. Note that $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ (resp. $\sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$) represents the number of edges between c and a character in $\grave{\Sigma}$ (resp. $\acute{\Sigma}$). By greedy choice, at least half of the edges in question become the ones connecting two characters each from $\acute{\Sigma}$ and $\grave{\Sigma}$. Hence, in the end, $|E|$ becomes at least $(|w| - 1)/2$, where let E denote the set of edges between characters from $\acute{\Sigma}$ and $\grave{\Sigma}$ (recalling that there are exactly $|w| - 1$ edges). Since each edge in E corresponds to an occurrence of a pair from $\acute{\Sigma}\grave{\Sigma} \cup \grave{\Sigma}\acute{\Sigma}$ in w, at least one of the two partitions $(\acute{\Sigma}, \grave{\Sigma})$ and $(\grave{\Sigma}, \acute{\Sigma})$ covers more than half of E. Hence we achieve our final bound $|E|/2 = (|w| - 1)/4$ by choosing an appropriate partition at Line 7.

In order to see that Algorithm 1 runs in $O(m)$ time, we only have to care about Line 3 and Line 7. We can compute $\sum_{\tilde{c} \in \grave{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ and $\sum_{\tilde{c} \in \acute{\Sigma}} \mathsf{Freq}(c, \tilde{c}, \cdot)$ by going through all $\mathsf{Freq}(c, \cdot, \cdot)$ for fixed c in the adjacency list, which are consecutive in the sorted list. Since each element of the list is used only once, the cost for Line 3 is $O(m)$ in total. Similarly the computation at Line 7 can be done by going through the adjacency list again. Thus the algorithm runs in $O(m)$ time. ◀

A.3 Proof of Lemma 8

Proof. We first compute the adjacency list of w. This can be easily done in $O(|w|)$ time and space by sorting the $|w| - 1$ size multiset $\{(w[i], w[i + 1], 0) \mid 1 \leq i < |w|, w[i] > w[i + 1]\} \cup \{(w[i + 1], w[i], 1) \mid 1 \leq i < |w|, w[i] < w[i + 1]\}$ by radix sort. Then by Lemma 7 we compute a partition $(\acute{\Sigma}, \grave{\Sigma})$ in linear time in the size of the adjacency list, which is $O(|w|)$. Next we scan w in $O(|w|)$ time and list all the occurrences of pairs to be compressed. Each pair $\acute{c}\grave{c} \in \acute{\Sigma}\grave{\Sigma}$ at position i is listed by a triple $(\acute{c}, \grave{c}, i)$ of integers in Σ. Then we sort the list according to the pair of integers $(\acute{c}, \grave{c})$, which can be done in $O(|w|)$ time and space by radix sort. Finally, we replace each pair with a fresh letter based on the rank of $(\acute{c}, \grave{c})$. ◀

A.4 Proof of Lemma 9

Proof. We first compute T_0 in $O(N)$ by sorting the characters used in T and replacing them with ranks of characters. Then we compress T_0 by applying BComp and PComp by turns and get $T_1, T_2 \ldots T_{\hat{h}}$. One technical problem is that characters used in an input string w of BComp and PComp should be in $[1..|w|]$, which is crucial to conduct radix sort efficiently in $O(|w|)$ time (see Lemmas 6 and 8). However letters in T_h do not necessarily hold this property. To overcome this problem, during computation we maintain ranks of letters among those used in the current T_h, which should be in $[1..|T_h|]$, and use the ranks instead of letters for radix sort. If we have such ranks in each level, we can easily maintain them by radix sort for the next level. Now, in every level h $(0 \leq h < \hat{h})$ the compression from T_h to T_{h+1} can be conducted in $O(|T_h|)$ time and space. Since PComp compresses a given string by a constant factor, the total running time can be bounded by $O(N)$ time. ◀

A.5 Proof of Lemma 11

Proof. Since a block is compressed into one letter, the number of parent nodes of C is at most m. As every internal node has two or more children, it is easy to see that there are $O(m)$ ancestor nodes of the parent nodes of C. ◀