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.

1 Introduction

Given two positions $i$ and $j$ in a text $T$ of length $N$, a longest common extension (LCE) query 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 $S$ 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 $\ell$ 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 $O(\frac{N^2}{\pi} \lg N)$ 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 $O(n^2h)$ time and $O(n^2)$ space from $S$, and supports LCE queries in $O(h \log N)$ time, where $h$ is the height of $S$. 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 $O(z \log N \log^* N)$ space, can support LCE queries in $O(\log N + \log \ell \log^* N)$ time, where $z$ is the size of LZ77 factorization\(^1\) of $T$ and $\log^*$ is the iterated logarithm. The signature encoding is based on the locally 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 $\ell$ in $T$ is guaranteed to be compressed in almost the same way leaving only $O(\log \ell \log^* N)$ discrepancies in its surrounding. As a result, an LCE query can be answered by tracing the $O(\log \ell \log^* N)$ surroundings created over two occurrences of the longest common extension. Since the cost $O(\log N)$ is needed anyway to traverse the derivation tree of height $O(\log N)$ from the root, an LCE query is supported in $O(\log N + \log \ell \log^* N)$ 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 $O(\log N)$ 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 $O(\log N)$ discrepancies in its surrounding. This leads to an $O(\log N)$-time algorithm to answer LCE queries, improving the $O(\log N + \log \ell \log^* N)$-time algorithm of [17]. We also improve the data structure size from $O(z \log N \log^* N)$ of [17]\(^2\) to $O(z \log(N/z))$.

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 $O(\log N)$ on which we can simulate the traversal of the derivation tree in constant time per move. Thus, it naturally supports $\text{Extract}(i, \ell)$ queries, which asks for retrieving the substring $T[i..i + \ell - 1]$, in $O(\log N + \ell)$ 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 $O(z \log(N/z) \log^* N)$ by using the same trick we use in Lemma 13.
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 $O(z \log(N/z))$ that supports $\text{Extract}(i, \ell)$ in $O(\log N + \ell)$ time and $\text{LCE}$ queries in $O(\log N)$ time.

Theorem 2. Given an SLP of size $n$ generating a string $T$ of length $N$, we can compute in $O(n \log(N/n))$ time and $O(n + z \log(N/z))$ space a compressed representation of $T$ of size $O(z \log(N/z))$ that supports $\text{Extract}(i, \ell)$ in $O(\log N + \ell)$ time and $\text{LCE}$ queries in $O(\log N)$ 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 $O(z \log(N/z))$ [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 $O(z \log^2(N/z))$ time and $O(z \log(N/z))$ space a compressed representation of $T$ of size $O(z \log(N/z))$ that supports $\text{Extract}(i, \ell)$ in $O(\log N + \ell)$ time and $\text{LCE}$ queries in $O(\log N)$ 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 $O(g^* \log(N/g^*))$, 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 $\text{LCE}$ 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 $z \leq g^*$ 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 $\text{BComp}$ (see Section 4.1.2). Also, we achieve a better construction time $O(n \log(N/n))$ than what we obtain by straightforwardly applying the analysis in [9]—$O(n \log N)$.

2 Preliminaries

An alphabet $\Sigma$ is a set of characters. A string over $\Sigma$ is an element in $\Sigma^*$. For any string $w \in \Sigma^*$, $|w|$ denotes the length of $w$. Let $\varepsilon$ be the empty string, i.e., $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the $i$-th character of $w$. For any $1 \leq i \leq j \leq |w|$, $w[i..j]$
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 \( w \) if \( w[i..i + |x| - 1] = x \). A substring \( w[i..j] = c^d \) (\( c \in \Sigma, d \geq 1 \)) of \( w \) is called a block if it is a maximal run of a single character, i.e., \( (i = 1 \lor w[i - 1] \neq c) \land (j = |w| \lor w[j + 1] \neq c) \).

The text on which LCE queries are performed is denoted by \( T \in \Sigma^* \) with \( N = |T| \) throughout this paper. We assume that \( \Sigma \) 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:

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

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 \( \Sigma \) 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 \( \Sigma \) 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 \( \text{val}(p) \) 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 \( \text{val}(p) \). 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.

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

\(^3\) We treat the last variable as the starting variable.
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 $\geq 2$ with fresh letters. Note that BComp eliminates all blocks of length $\geq 2$ 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 $\geq 2$, PComp compresses $w$ by replacing all pairs from $[1..|w|]$ with fresh letters, where $(\Sigma, \Sigma)$ is a partition of $\Sigma$, i.e., $\Sigma = \Sigma \cup \Sigma$ and $\Sigma \cap \Sigma = \emptyset$. We can deterministically compute in $O(|w|)$ time and space a partition of $\Sigma$ by which at least $|w|/4$ 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, $T_1, T_2, \ldots, T_h$, where $|T_h| = 1$. For any $0 \leq h < \hat{h}$, 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 $c \rightarrow \check{c}^d$ 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 $c \rightarrow \check{c}$ are introduced.

Let $\Sigma_h$ be the set of letters appearing in $T_h$. For any even $h$ ($0 \leq h < \hat{h}$), let $\Sigma_h$ denote the set of letters with which there is a block of length $\geq 2$ in $T_h$. For any odd $h$ ($0 \leq h < \hat{h}$), let $(\Sigma_h, \Sigma_h)$ denote the partition of $\Sigma_h$ 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 $\geq 2$, let $\text{Freq}_w(c, \check{c}, 0)$ (resp. $\text{Freq}_w(c, \check{c}, 1)$) with $c > \check{c} \in \Sigma$ denote the number of occurrences of $c\check{c}$ (resp. $\check{c}c$) 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.
Figure 1. An example of how TtoG compresses $T_0$. Below we enumerate non-empty $\Sigma_0$, $\Sigma_1$, $\Sigma_2$, and production rules introduced in each level. From $T_0$ to $T_1$: $\Sigma_0 = \{1, 2, 4\}$, $\{5 \to 1^2, 6 \to 1^3, 7 \to 2^2, 8 \to 4^2\}$. From $T_1$ to $T_2$: $\Sigma_1 = \{1, 3, 5, 6, 7\}$, $\Sigma_2 = \{2, 4, 8\}$, $\{9 \to (1, 2), 10 \to (3, 4), 11 \to (3, 8), 12 \to (5, 2), 13 \to (6, 2)\}$. From $T_2$ to $T_3$: $\Sigma_2 = \{9\}$, $\{14 \to 9^2\}$. From $T_3$ to $T_4$: $\Sigma_3 = \{3, 7, 12, 13\}$, $\Sigma_4 = \{10, 14\}$, $\{15 \to (7, 14), 16 \to (12, 10), 17 \to (13, 10)\}$. From $T_4$ to $T_5$: $\Sigma_5 = \{3, 10, 16, 17\}$, $\Sigma_6 = \{15\}$, $\{18 \to (16, 15), 19 \to (17, 15)\}$. From $T_5$ to $T_6$: $\Sigma_6 = \{3, 10, 11\}$, $\Sigma_7 = \{18, 19\}$, $\{20 \to (3, 19), 21 \to (10, 18)\}$. From $T_6$ to $T_7$: $\Sigma_7 = \{11, 20\}$, $\Sigma_8 = \{21\}$, $\{22 \to (20, 21)\}$. From $T_7$ to $T_8$: $\Sigma_8 = \{22\}$, $\Sigma_9 = \{11\}$, $\{23 \to (22, 21)\}$.

refer to the list of non-zero $\text{Freq}_w(c, \bar{c}, \cdot)$ 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 $\text{Freq}_w(c, \bar{c}, 0)$ (resp. $\text{Freq}_w(c, \bar{c}, 1)$) edges from $c$ to $\bar{c}$ (resp. from $\bar{c}$ 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 $(\Sigma, \Sigma)$ of $\Sigma$ 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 $(\Sigma, \Sigma)$ of $\Sigma$ such that the number of occurrences of pairs from $\Sigma^2$ in $w$ is at least $(|w| - 1)/4$.

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

Lemma 9. Given a string $T$ over $\Sigma = [1..N^{O(1)}]$, we can compute $\text{TtoG}(T)$ 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 $\text{TtoG}(T)$. 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 $\text{val}(PSeq(w)) = w$ 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 $\text{TtoG}(T)$ is $O(\lg N)$. The significant property of $PSeq(w)$ is that it occurs “every” occurrence of $w$. A similar property has been observed in CFGs produced by locally consistent parsing and utilized for compressed indexes [14, 16]
We iterate this until the string disappears. \( PSeq(w) \) 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 \( PSeq(w) \) is guaranteed (see also Figure 2).

Table 1: How to compute a partition of \( \Sigma \) for \( PComp \) to compress \( w \) by 3/4.

<table>
<thead>
<tr>
<th>Input:</th>
<th>Adjacency list of ( w \in \Sigma^* ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>((\hat{\Sigma}, \hat{\Sigma})) s.t. # occurrences of pairs from ( \hat{\Sigma}\hat{\Sigma} ) in ( w ) is at least ((</td>
</tr>
<tr>
<td>*/ The information whether ( c \in \Sigma ) is in ( \hat{\Sigma} ) or ( \hat{\Sigma} ) is written in the data space for ( c ), which can be accessed in ( O(1) ) time. */</td>
<td></td>
</tr>
</tbody>
</table>

```
1 \hat{\Sigma} ← \Sigma ← \emptyset;
2 \textbf{foreach } c \in \Sigma \textbf{ in increasing order do}
3 \quad \text{if } \sum_{c \in \Sigma} \text{Freq}_w(c, \cdot, \cdot) \geq \sum_{c \in \Sigma} \text{Freq}_w(c, \cdot, \cdot) \text{ then}
4 \quad \quad \text{add } c \text{ to } \hat{\Sigma};
5 \quad \text{else}
6 \quad \quad \text{add } c \text{ to } \hat{\Sigma};
7 \textbf{if } \# \text{ occurrences of pairs from } \hat{\Sigma}\hat{\Sigma} < \# \text{ occurrences of pairs from } \hat{\Sigma}\hat{\Sigma} \text{ then}
8 \quad \text{switch } \hat{\Sigma} \text{ and } \hat{\Sigma};
9 \textbf{return } (\hat{\Sigma}, \hat{\Sigma});
```

---

**Figure 2** \( PSeq \) for \( w_0 = [1, 1, 2, 3, 4, 2, 2, 1, 2, 1, 2, 3, 4] \) under \( \hat{\Sigma}_h, \hat{\Sigma}_h, \hat{\Sigma}_h \) 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 \( \in \hat{\Sigma}_0 \). At level 1, a letter 2 (resp. 3) is popped out from the leftend (resp. rightend) of \( w_1 \) because 2 \( \in \hat{\Sigma}_1 \) and 3 \( \in \hat{\Sigma}_1 \). At level 2, a block of 9 is popped out from the rightend of \( w_2 \) because 9 \( \in \hat{\Sigma}_2 \). At level 3, a letter 7 (resp. 7) is popped out from the leftend (resp. rightend) of \( w_3 \) because 10 \( \in \hat{\Sigma}_1 \) and 7 \( \in \hat{\Sigma}_1 \). Then, \( PSeq(w_0) = [1, 1, 2, 10, 7, 9, 9, 3, 4] \). 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, \( PSeq(w_0) \) 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 \( O(\lg N) \).

\( PSeq(w) \) 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 \( c \in \hat{\Sigma}_h \). 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 \( \hat{\Sigma}_h \) and \( \hat{\Sigma}_h \), respectively. Then we get \( w_{h+1} \) by applying \( PComp \) to the remaining string.

We iterate this until the string disappears. \( PSeq(w) \) 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 \( PSeq(w) \) 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 $TtoG(T)$ 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 $f_1 \ldots f_z$ 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 $f_1 f_2 \ldots f_i$. Clearly $|L_i| = O(\lg N)$. For any $1 < i \leq z$, we estimate $|L_i \setminus L_{i-1}|$. Since $f_i$ occurs in $f_1 \ldots f_{i-1}$, 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 $[f_1 \ldots f_{i-1}+1]$ 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 $TtoG(T)$ 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 $h$, $|T_h| \leq N(3/4)^{N/2} = z$, 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 $TtoG(T)$.

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

**Proof.** We compute $LCE(i,j)$ by matching the common sequence of letters occurring at $i$ and $j$ from left to right. First we traverse the derivation tree of $TtoG(T)$ 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, $LCE(i,j) = 0$, and we are done. Otherwise we stop at some internal nodes that contain the common block in their children. Let $\ell_1$ be the length of the string generated by the block. Because $LCE(i,j) \geq \ell_1$, 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 $LCE(i,j) = \sum_{k=1}^{m} \ell_k$, where $\ell_1, \ell_2, \ldots, \ell_m$ 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 \( S = (\Sigma, V, D) \) of size \( n \) generating \( T \). Basically what we consider is to simulate \( T \) on \( S \), namely, compute \( T \) without decompressing \( S \) explicitly. In Section 4.1, we present an algorithm \( \text{SimTtoG} \) that simulates \( T \) 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 \( \text{SimTtoG} \) to get Theorem 2.

4.1 SimTtoG: Simulating TtoG on CFGs

We present an algorithm \( \text{SimTtoG} \) to simulate \( T \) on \( S \). To begin with, we compute the CFG \( S_0 = (\Sigma_0, V, D_0) \) obtained by replacing, for all variables \( X \in V \) with \( D(X) \in \Sigma \), every occurrence of \( X \) in the righthand sides of \( D \) with the letter generating \( D(X) \). Note that \( \Sigma_0 \) is the set of terminals of \( S_0 \) and \( S_0 \) generates \( T_0 \). \( \text{SimTtoG} \) transforms level by level \( S_0 \) into CFGs, \( S_1 = (\Sigma_1, V, D_1), S_2 = (\Sigma_2, V, D_2), \ldots, S_h = (\Sigma_h, V, D_h) \), where each \( S_h \) generates \( T_h \). Namely, compression from \( T_h \) to \( T_{h+1} \) is simulated on \( S_h \). We can correctly compute the letters introduced in each level \( h+1 \) while modifying \( S_h \) into \( S_{h+1} \), and hence, we get all the letters of \( T \) 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 \( S_h \) (it is a generalization of SLPs in a different sense from RLSLPs): For any \( X \in V, D_h(X) \) consists of an “arbitrary number” of letters and at most “two” variables. More precisely, the following condition holds:

For any variable \( X \in V \) with \( D(X) = X\hat{X}, D_h(X) \) is either \( w_1\hat{X} w_2\hat{X} w_3, w_1\hat{X} w_2, w_2\hat{X} w_3 \) or \( w_2 \) with \( w_1, w_2, w_3 \in \Sigma^*_h \), where \( w_1 = w_3 \) if \( X \) is not the starting variable.

As opposed to SLPs and RLSLPs, we define the size of \( S_h \) by the total lengths of righthand sides and denote it by \( |S_h| \).

4.1.1 PComp on CFGs

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

\textbf{Lemma 15 (Lemma 6.1 of [10]).} For any odd \( h \) (0 ≤ \( h \) < \( h \)), the adjacency list of \( T_h \), whose size is \( O(|S_h|) \), can be computed in \( O(|S_h| + n) \) time and space.

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

Now observe that any occurrence \( i \) of a pair \( \hat{c}\hat{c} \) 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 \( S_h \) (recall that \( S_h \) generates \( T_h \)). We intend to count all the occurrences of pairs associated with \( X \) in \( D_h(X) \). For example, let \( D_h(X) = \hat{X} w_2 \hat{X} \hat{X} \hat{X} w_2 \hat{X} w_2 \) with \( w_2 \in \Sigma^*_h \). Then \( \hat{c}\hat{c} \)

\textit{\textsuperscript{5} It is sufficient to compute \( \text{VOcc}(X) \) once at the very beginning of \( \text{SimTtoG} \).}
appears explicitly in $w_2$ or crosses the boundaries of $X$ and/or $\hat{X}$. If $c\hat{c}$ crosses the boundary of $\hat{X}$, $RML(\hat{X})$ is $\hat{c}$ and $\hat{c}$ follows, i.e., $(w_2)[1] = \hat{c} \lor (w_2 = \varepsilon \land LML(\hat{X}) = \hat{c})$. Using $RML(\hat{X})$ and $LML(\hat{X})$, we can compute in $O(|D_h(X)|)$ time and space a $(|D_h(X)| - 1)$-size multiset that lists all the explicit and crossing pairs in $D_h(X)$. Each pair $c\hat{c}$ with $c > \hat{c}$ (resp. $\hat{c} < c$) is listed by a quadruple $(\hat{c}, \hat{c}, 0, VOcc(X))$ (resp. $(\hat{c}, \hat{c}, 1, VOcc(X))$). $VOcc(X)$ means that the pair has a weight $VOcc(X)$ 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 $O(|S_h|)$ 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 $O(|S_h|)$.

The next lemma shows how to implement $PComp$ on CFGs:

\textbf{Lemma 16.} For any odd $h$ ($0 \leq h < \hat{h}$), we can compute $S_{h+1}$ from $S_h$ in $O(|S_h| + n)$ time and space. In addition, $|S_{h+1}| \leq |S_h| + 2n$.

\textbf{Proof.} We first compute the partition $(\Sigma_h, \widehat{\Sigma}_h)$ of $\Sigma_h$, which can be done in $O(|S_h| + n)$ time and space by Lemmas 15 and 7.

Given $(\Sigma_h, \widehat{\Sigma}_h)$, we can detect all the positions of the pairs from $\widehat{\Sigma}_h \widehat{\Sigma}_h$ in the righthands of $D_h$, 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 \textit{uncross} them by popping out every $LML(Y) \in \widehat{\Sigma}_h$ and $RML(Y) \in \widehat{\Sigma}_h$ from $val_{S_h}(Y)$ and popping them into the appropriate positions in the other rules. More precisely, we do the followings (for technical reason, do $PopInLet$ first):

\textbf{PopInLet.} For any variable $X$, if $D_h(X)[i] = Y \in V$ with $i > 1$ ($i \geq 1$ if $X$ is the starting variable) and $LML(Y) \in \widehat{\Sigma}_h$, replace the occurrence of $Y$ with $LML(Y)Y$: if $D_h(X)[i] = Y \in V$ with $i < |D_h(X)|$ ($i \leq |D_h(X)|$ if $X$ is the starting variable) and $RML(Y) \in \widehat{\Sigma}_h$, replace the occurrence of $Y$ with $Y RML(Y)$.

\textbf{PopOutLet.} For any variable $X$ other than the starting variable, if $D_h(X)[1] \in \widehat{\Sigma}_h$, remove the first letter of $D_h(X)$; and if $D_h(X)[|D_h(X)|] \in \widehat{\Sigma}_h$, remove the last letter of $D_h(X)$.

In addition, if $X$ becomes empty, we remove all the appearances of $X$ in $D_h$.

$PopOutLet$ removes $LML(Y) \in \widehat{\Sigma}_h$ and $RML(Y) \in \widehat{\Sigma}_h$ from $val_{S_h}(Y)$ (which can be a part of a crossing pair), and $PopInLet$ introduces the removed letters into appropriate positions in $D_h$ so that the modified $S_h$ 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 $S_h$ increases at most $2n$. The uncrossing can be conducted in $O(|S_h| + n)$ time.

Since all the pairs to be compressed become explicit, we can conduct $BComp$ in $O(|S_h| + n)$ time as follows. We scan righthand sides in $O(|S_h|)$ time and list all the occurrences of pairs to be compressed. Each occurrence of pair $c\hat{c}$ is listed by a triple $(\hat{c}, \hat{c}, p)$, where $p$ is the pointer to the occurrence. Then we sort the list according to the pair of integers $(\hat{c}, \hat{c})$, which can be done in $O(|S_h| + n)$ time and space by radix sort because $\hat{c}$ and $\hat{c}$ are $O(n^2)$. Finally, we replace each pair at position $p$ with a fresh letter based on the rank of $(\hat{c}, \hat{c})$. \hfill \blacktriangleled
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 \( \geq 2 \) 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 \( S_h \) (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 \( D_h(X) \) and the others are crossing the boundaries of variables in \( D_h(X) \). The numbers of explicit blocks and crossing blocks in \( D_h \) is at most \(|S_h| \) 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 \( D_h \).

However here we have a problem. In order to give a unique letter to a block \( e^d \), we have to sort the pairs of integers \((e, d)\). Since \( d \) might be exponentially larger than \(|S_h| + n\), radix sort cannot be executed in \( O(|S_h| + n) \) time and space. In Section 6.2 of [10], Jež 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 \( e^d \) is short if \( d = O(|S_h| + n) \) 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 \( T_h[b..e] = e^d \) is short if it does not include a substring generated from a unary variable.

**Proof.** Consider the derivation tree of \( S_h \) and the shortest path from \( T_h[b] \) to \( T_h[e] \). Let \( X_1X_2\ldots X_mX_{m+1}\ldots X_m \) 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 \( d \leq \sum_{i=1}^{m'} |D_h(X_i)| \leq 2|S_h| \) 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 \( D_h \), 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 \( D_h \), 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 $\text{SimTtoG}$. 

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

Lemma 20. For any even $h$ ($0 \leq h < h'$), we can compute $S_{h+1}$ from $S_h$ in $O(|S_h| + n + m \log m)$ time and $O(|S_h| + n)$ space, where $m$ is the number of long blocks in $D_h$. In addition, $|S_{h+1}| \leq |S_h| + 2n$.

4.1.3 The complexities of $\text{SimTtoG}$

Theorem 21. $\text{SimTtoG}$ runs in $O(n \log^2(N/n))$ time and $O(n \log(N/n))$ space.

Proof. Using $P\text{Comp}$ and $B\text{Comp}$ implemented on CFGs (see Lemma 16 and 20), $\text{SimTtoG}$ transforms level by level $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_k$. In each level, the size of CFGs can increase at most $2n$ by the procedure of uncrossing. Since $|S_h| = O(n \log N)$ for any $h$ ($0 \leq h < h'$), we get the time complexity $O(n \log^2 N)$ by simply applying Lemmas 16 and 20.

We can improve it to $O(n \log^2(N/n))$ by a similar trick used in the proof of Lemma 13. At some level $h'$ where $|T_{h'}|$ becomes less than $n$, we decompress $S_{h'}$ and switch to $T\text{toG}$, which transforms $T_{h'}$ into $T_h$ in $O(n)$ time by Lemma 9. We apply Lemmas 16 and 20 only for $h$ with $0 \leq h < h'$. Since $h' = O(\log(N/n))$, $|S_{h'}| = O(n \log(N/n))$ for any $h$ ($0 \leq h < h'$). Hence, we get the time complexity $O(n \log^2(N/n))$. The space complexity is bounded by the maximum size of CFGs $S_0, S_1, \ldots, S_{h'}$, which is $O(n \log(N/n))$.

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

We modify $\text{SimTtoG}$ slightly to run in $O(n \log(N/n))$ time and $O(n + z \log(N/z))$ space. The idea is the same as what has been presented in Section 6.1 of [10]. The problem of $\text{SimTtoG}$ is that the sizes of intermediate CFGs $S_h$ can grow up to $O(n \log(N/n))$. If we can keep their sizes to $O(n)$, everything goes fine. This can be achieved by using two different types of partitions of $\Sigma_h$ for $P\text{Comp}$: One is for compressing $T_h$ by a constant factor, and the other for compressing $|S_h|$ by a constant factor (unless $|S_h|$ is too small to compress). Recall that the former partition has been used in $T\text{toG}$ and $\text{SimTtoG}$, 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 $|S_h|$ by a constant factor by considering the adjacency list from a set of strings in the right-hand sides of $D_h$. The adjacency list can be easily computed in $O(|S_h| + n)$ time and space by modifying the algorithm described in the proof of Lemma 15: We just ignore the weight $\text{VOcc}(X)$, 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 $G\text{toG}$ and the resulting RLSLP by $G\text{toG}(S)$. Note that $G\text{toG}(S)$ is not identical to $T\text{toG}(T)$ in general because the partitions used in $G\text{toG}$ depend on the input $S$. Still the height of $G\text{toG}(S)$ is $O(\log N)$ and the properties of $P\text{Seqs}$ hold. Hence we can support LCE queries on $G\text{toG}(S)$ as we did on $T\text{toG}(T)$ by Lemma 14.

4.3 Proof of Theorem 2

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

References


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 \( \geq 2 \). Each block \( c^d \ (c \in \Sigma, d \geq 2) \) at position \( i \) is listed by a triple \((c, d, i)\) of integers in \( \Sigma \). 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 \( \hat{\Sigma} \) or to \( \bar{\Sigma} \) depending on \( \sum_{\hat{c} \in \hat{\Sigma}} \text{Freq}(c, \hat{c}, \cdot) \geq \sum_{\bar{c} \in \bar{\Sigma}} \text{Freq}(c, \bar{c}, \cdot) \). Note that \( \sum_{\hat{c} \in \hat{\Sigma}} \text{Freq}(c, \hat{c}, \cdot) \) (resp. \( \sum_{\bar{c} \in \bar{\Sigma}} \text{Freq}(c, \bar{c}, \cdot) \)) represents the number of edges between \( c \) and a character in \( \hat{\Sigma} \) (resp. \( \bar{\Sigma} \)). By greedy choice, at least half of the edges in question become the ones connecting two characters each from \( \hat{\Sigma} \) and \( \bar{\Sigma} \). Hence, in the end, \( |E| \) becomes at least \( (|w| - 1)/2 \), where let \( E \) denote the set.
of edges between characters from \( \hat{\Sigma} \) and \( \Sigma \) (recalling that there are exactly \( |w| - 1 \) edges). Since each edge in \( E \) corresponds to an occurrence of a pair from \( \hat{\Sigma} \Sigma \cup \Sigma \hat{\Sigma} \) in \( w \), at least one of the two partitions \((\hat{\Sigma}, \Sigma)\) and \((\Sigma, \hat{\Sigma})\) 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 \( \sum_{c \in \Sigma} \text{Freq}(c, \hat{c}, \cdot) \) and \( \sum_{c \in \Sigma} \text{Freq}(c, \cdot, \cdot) \) by going through all \( \text{Freq}(c, \cdot, \cdot) \) 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 \((\hat{\Sigma}, \Sigma)\) 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 \( \hat{c} c \) at position \( i \) is listed by a triple \((\hat{c}, c, i)\) of integers in \( \Sigma \). Then we sort the list according to the pair of integers \((\hat{c}, c)\), 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 \((\hat{c}, c)\).

### 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 \( \text{BComp} \) and \( \text{PComp} \) by turns and get \( T_1, T_2 \ldots T_h \). One technical problem is that characters used in an input string \( w \) of \( \text{BComp} \) and \( \text{PComp} \) 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 \( \text{PComp} \) 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 \( C \).

---

**CPM 2017**