Computing Lexicographic Parsings

Dominik Köppl

M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

Abstract

We give memory-friendly algorithms computing the compression schemes plcpcomp or lex-parse in linear or near-linear time, and give upper and lower bounds on the space requirements of our algorithm computing plcpcomp.

Keywords: lossless data compression, factorization algorithms, memory-efficiency

1 Introduction

In this article, we focus on computing the compression schemes *plcpcomp* [1] and *lex-parse* [2] within low memory. Both schemes are macro schemes [3] like the well-known Lempel–Ziv 77 (LZ77) factorization. While LZ77 restricts factors to refer to previous text positions, the schemes in our focus restrict factors to refer to the starting positions of lexicographically preceding suffixes. Such kinds of schemes are also called *lexicographic parsings*.

Lexicographic parsings have been studied in the context of text compression [2, Theorem 26], where it is known that the smallest lexicographic parsing yielding v factors is of size $\mathcal{O}(v \log(n/v))$, where v is the size of the smallest macro scheme. Further, it has been shown that lex-parse attains this value v for all inputs. However, the authors only address the computation with a linear-time algorithm using $\mathcal{O}(n \log n)$ bits of space, which can be quite large in practice. As far as we are aware, we address here for the first time the space-efficient computation of lex-parse.

2 Preliminaries

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$ for a natural number n. The alphabet Σ induces the *lexicographic order* $\prec$ on the set of strings Σ^*. Let $|T|$ denote the length of a string $T \in \Sigma^*$. We write $T[j]$ for the j-th character of T with $j \in [1\mathinner{.\,.} n]$. Given $T \in \Sigma^*$ consists of the concatenation $T = UVW$ for $U, V, W \in \Sigma^*$, we call U, V, and W a *prefix*, a *substring*, and a *suffix* of T, respectively. Given that the substring V starts at the i-th and ends at the j-th position of T, we also write $V = T[i\mathinner{.\,.} j]$ and

Table 1: Suffix array, its inverse, Φ, the LCP array, PLCP, and the BWT of our running example string T. The last two rows depict the sparse representation Φ_{S} of Φ with bit vector B described in Sect. 5. If $\mathsf{BWT}[\mathsf{ISA}[i]] = \mathsf{BWT}[\mathsf{ISA}[i]-1]$, i.e., $T[i-1] = T[\Phi[i]-1]$, then $\Phi[i] = \Phi[i-1]+1$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	a	b	a	b	b	a	b	a	b	a	b	b	a	b	b	a	a	b	a	b	a	$
ISA	7	18	10	21	15	6	16	8	19	11	22	17	9	20	13	3	5	14	4	12	2	1
PLCP	4	5	4	3	4	5	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0
Φ	6	12	13	14	18	17	5	1	2	3	4	7	8	9	20	21	19	15	16	10	22	11
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
SA	22	21	16	19	17	6	1	8	13	3	10	20	15	18	5	7	12	2	9	14	4	11
LCP	0	0	1	1	3	5	4	7	2	4	5	0	2	2	4	5	3	5	6	1	3	4
BWT	a	b	b	b	a	b	$	b	b	b	b	a	b	a	b	a	b	a	a	a	a	a
Φ_{S}	6	12			18	17	5	1				7			20		19	15		10	22	11
B	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1

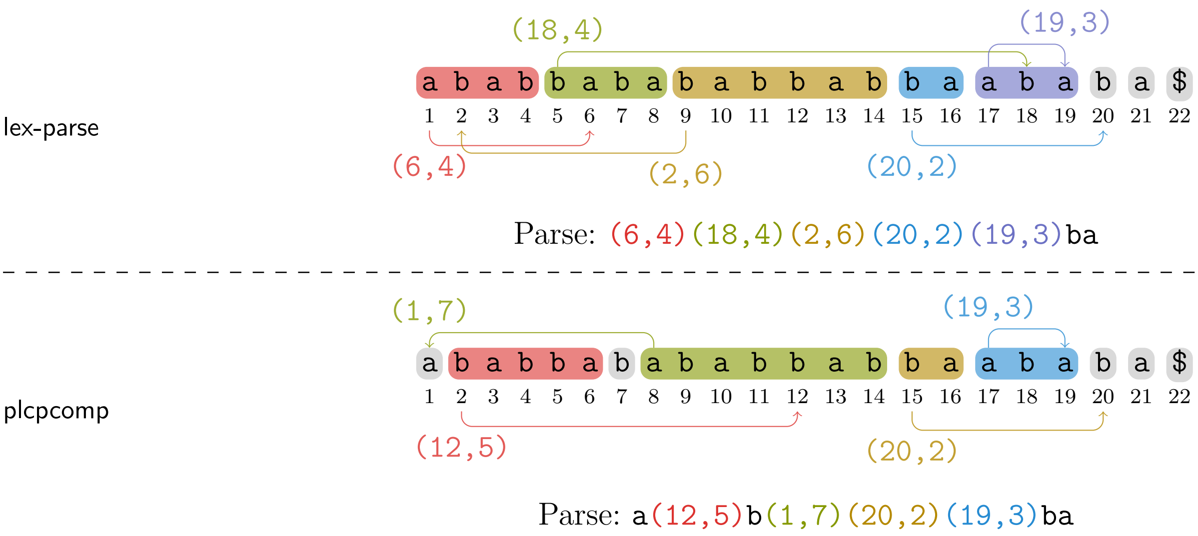

Figure 1: Factorization of our running example by lex-parse (*top*) and plcpcomp (*bottom*). A factor F is visualized by a rounded rectangle. If F is a literal factor, its parse consists of a mere character; otherwise, its parse consists of its referred position p and its length ℓ such that $F = T[p \mathinner{..} p + \ell - 1]$. To obtain the lengths and the referred positions, we can make use of Φ and PLCP of T displayed in Table 1.

$W = T[j+1 \mathinner{..}]$. Let $\mathsf{lcp}(V, W)$ denote the length of the longest common prefix between two strings V and W. In the following, we take an element $T \in \Sigma^*$ with $|T| = n$, and call it the *text*. We stipulate that T ends with a unique sentinel $T[n] = \$ \notin \Sigma$ that is lexicographically smaller than every character of Σ.

Text Data Structures. Let SA denote the *suffix array* [4] of T. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[\mathsf{SA}[i] \mathinner{..}] \prec T[\mathsf{SA}[i+1] \mathinner{..}]$ for all integers $i \in [1 \mathinner{..} n-1]$. Let ISA of T be the inverse of SA, i.e., $\mathsf{ISA}[\mathsf{SA}[i]] = i$ for every $i \in [1 \mathinner{..} n]$. The *Burrows–Wheeler transform (BWT)* of T is the string BWT with $\mathsf{BWT}[i] = T[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$ otherwise, for every $i \in [1 \mathinner{..} n]$. The *LCP array* is an array with the property that $\mathsf{LCP}[i]$ is the length of the longest common prefix (LCP) of $T[\mathsf{SA}[i] \mathinner{..}]$ and $T[\mathsf{SA}[i-1] \mathinner{..}]$ for every $i \in [2 \mathinner{..} n]$. For convenience, we stipulate that $\mathsf{LCP}[1] := 0$. The array Φ is defined as $\Phi[i] := \mathsf{SA}[\mathsf{ISA}[i] - 1]$, and $\Phi[i] := n$ in case that $\mathsf{ISA}[i] = 1$. The *PLCP array* PLCP stores the entries of LCP in text order, i.e., $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. Table 1 illustrates the introduced data structures on our running example string $T :=$ `ababbababbabbaababa$`.

Rank/Select. Given a character $\mathtt{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathsf{rank}_{\mathtt{c}}(j)$ counts the occurrences of c in $T[1 \mathinner{..} j]$, and the *select* query $T.\mathsf{select}_{\mathtt{c}}(j)$ gives the position of the j-th c in T. We stipulate that $\mathsf{rank}_{\mathtt{c}}(0) = \mathsf{select}_{\mathtt{c}}(0) = 0$. If the alphabet is binary, i.e., when T is a bit vector, there are data structures [5, 6] that use $o(|T|)$ extra bits of space, and can compute rank and select in constant time, respectively. Each of those data structures can be constructed in time linear in $|T|$. We say that a bit vector has a *rank-support* and a *select-support* if it is endowed with data structures providing constant time access to rank and select, respectively.

Computation Model. We use the word RAM model with word size $\Omega(\lg n)$ for some natural number n. The arrays SA and LCP can be constructed in $\mathcal{O}(n)$ time with the algorithms of Ko and Aluru [7] and Kasai et al. [8], respectively. With SA, we can construct ISA in $\mathcal{O}(n)$ time by using the fact that SA is a permutation.

Lemma 1 ([9, 10])**.** PLCP can be represented by $2n + o(n)$ bits and can be constructed in $\mathcal{O}(n)$ time.

3 Lexicographic Parsings

A *parse* is a representation of a factorization $F_1 \cdots F_z = T$ of a text T by a list whose x-th entry stores either

(a) the pair (src_x, ℓ_x) such that $F_x = T[dst_x \mathinner{..} dst_x + \ell_x - 1] = T[src_x \mathinner{..} src_x + \ell_x - 1]$ with $dst_x = 1 + \sum_{y=1}^{x-1} \ell_y$ being the starting position of F_x in T, or

(b) just the string F_x with $\ell_x := |F_x| = 1$.

We call the former representation (a) *referencing*, the latter representation (b) of a factor *literal*.

In [2, Sect. VI], Navarro et al. studied so-called *lexicographic parsings*. A parsing of T is called *lexicographic* if $T[src_x \mathinner{..}] \prec T[dst_x \mathinner{..}]$ for every referencing factor. Here, we focus on a stricter class of those parses, where $src_x = \Phi[dst_x]$ holds for all referencing factors. Note that $src_x = \Phi[dst_x]$ implies that $T[src_x \mathinner{..}] = T[\mathsf{SA}[\mathsf{ISA}[dst_x] - 1] \mathinner{..}] \prec T[dst_x \mathinner{..}]$ for $\mathsf{ISA}[dst_x] > 1$.[1] Two lexicographic parsings with $src_x = \Phi[dst_x]$ are lex-parse [2, Def. 11] and plcpcomp [1], which we focus on in the following. Figure 1 gives an example for both factorizations.

The *lex-parse* is a parsing $T = F_1 \cdots F_z$ such that $F_x = T[dst_x \mathinner{..} dst_x + \ell_x - 1]$ with $dst_1 = 1$ and $dst_{x+1} = dst_x + \ell_x$ if $\ell_x := \mathsf{PLCP}[dst] > 0$, or F_x is a literal factor with $\ell_x := |F_x| = 1$ otherwise.

The *plcpcomp*-parsing with a threshold $\xi \geq 1$ is recursively defined by replacing the longest reoccurring substring $T[dst_x \mathinner{..} dst_x + \ell_x - 1] = T[src_x \mathinner{..} src_x + \ell_x - 1]$ with $\ell_x \geq \xi$ by a factor F_x, where $src_x := \Phi[dst_x]$ and $\ell_x := \mathsf{PLCP}[dst_x]$. Ties are broken by choosing the smallest possible dst_x among all candidates with the same longest length ℓ_x. The recursion terminates when every reoccurring substring of the remaining text has a length smaller than ξ; each remaining character becomes a literal factor. For instance, if $\xi = 1$, there are σ' characters remaining, which are all distinct, if σ' is the number of distinct characters in the text. Note that $\xi = 1$ usually produces a less compressible output than larger threshold values since the parse of a referencing factor being a single character usually needs more bits than the character itself. For the sake of simplicity, we assume $\xi = 2$ in the following, which makes plcpcomp more similar to lex-parse, which implicitly has a threshold of 1 or 2 (depending on whether we make all factors of length 1 literal). Although the example of Fig. 1 is small, we observe that the number of literal factors of plcpcomp is larger than of lex-parse, while plcpcomp has the longest factors. This behavior is also reflected on real-world datasets shown in Table 2: Table 2 shows characteristics of both factorizations when built on datasets of the Pizza&Chili corpus. These statistics can be collected with the tudocomp[2] command-line tool `./tdc` with the parameters `-a 'lcpcomp(comp=X, threshold=2, coder=bi(ref=binary, len=binary, lit=huff))'` with `X` being `plcp` (for plcpcomp) or `lexparse`. Note that tudocomp has a threshold for both parsings, which we set to $\xi = 2$. We can see that the factors of plcpcomp are on average longer and have shorter distances than lex-parse. While the number of referencing factors of plcpcomp is less than lex-parse, it has much more literal factors, and may therefore produce a factorization with more factors than lex-parse. However, when considering compression, the literal factors can be easily compressed using a zeroth order entropy encoder like Huffman code, while long distances are rather hard to compress (see [11] considering encoding the distances of the LZ77 factorization). The numbers of observed factors are in line with the aforementioned fact that lex-parse produces the least number of factors among all other lexicographic parsings.

3.1 Factorization Algorithm

We briefly review the in-memory algorithm computing lex-parse or plcpcomp when having PLCP available, starting with the latter. The linear-time algorithm of Dinklage et al. [1] detects so-called *peaks*. A text position dst is a *peak* if

(a) $\mathsf{PLCP}[dst] \geq \xi$ and

(b) $dst = 1$, $\mathsf{PLCP}[dst - 1] < \mathsf{PLCP}[dst]$, or there is a referencing factor ending at $dst - 1$.

A peak dst is called *interesting* if there is no text position j with $dst \in (j \mathinner{..} j + \mathsf{PLCP}[j])$ and $\mathsf{PLCP}[j] \geq \mathsf{PLCP}[dst]$. An interesting peak dst is called *maximal* if there is no interesting peak j with $j \in (dst \mathinner{..} dst + \mathsf{PLCP}[dst])$.

With these definitions, we can compute plcpcomp as follows: We linearly scan the text from left to right, adding interesting peaks into a list L of text positions. On finding a maximal peak dst, we factorize $T[1 \mathinner{..} dst - 1]$ by using the peaks stored in L and their associated PLCP values. This takes $\mathcal{O}(|L|) = \mathcal{O}(dst)$ time.[3] We then continue with the plcpcomp factorization of $T[dst + \mathsf{PLCP}[dst] \mathinner{..}]$. Overall,

[1] This is well-defined for all referencing factors since $\mathsf{ISA}[n] = 1$ due to $T[n] = \$$ being the smallest character. Note that $\$$ is a unique character in the text, so $T[n]$ is always a literal factor.

[2] `https://github.com/tudocomp/tudocomp/tree/lex-parse`.

[3] Due to space restrictions, the detailed explanation of the algorithm is omitted. However, these details are not necessary for the understanding of this paper.

Table 2: Comparison of lex-parse and plcpcomp with $\xi = 2$ on Pizza&Chili datasets. n is the number of characters of the input text, z_r is the number of referencing factors, z_l is the number of literal factors, i.e., $z_r + z_l$ is the number of all factors. Next, $\varnothing d$ and $\varnothing \ell$ are the average distance and average length representing a referencing factor. Finally, [M] and [K] denote mega (10^6) and kilo (10^3), respectively.

input		lex-parse				plcpcomp			
name	n [M]	z [M]	$\varnothing d$ [M]	$\varnothing \ell$	$\varnothing z_l$ [K]	z [M]	$\varnothing d$ [M]	$\varnothing \ell$	$\varnothing z_l$ [K]
ESCHERICHIA_COLI	112.7	2.01	33.4	12	0.1	2.02	33.4	13	287.8
CERE	461.3	1.65	132.2	12	0.0	1.63	132.3	13	245.1
COREUTILS	205.3	1.43	26.3	10	6.1	1.32	28.5	10	166.2
EINSTEIN.DE.TXT	92.8	0.04	19.0	6	1.3	0.03	18.0	7	6.5
EINSTEIN.EN.TXT	467.6	0.10	81.4	7	1.8	0.09	75.6	8	14.3
INFLUENZA	154.8	0.77	30.3	51	0.1	0.66	27.4	57	142.4
KERNEL	258.0	0.79	52.5	9	2.2	0.73	50.8	10	90.3
PARA	429.3	2.24	130.9	12	0.0	2.14	131.0	13	364.4
WORLD_LEADERS	47.0	0.18	5.7	11	0.9	0.17	5.7	12	19.5

this accumulates to $\mathcal{O}(n)$ time. Computing lex-parse is in fact easier, since we do not have to maintain L: Starting with $F_1 := T[1 \mathinner{..} \max(1, \mathsf{PLCP}[1])]$, we greedily compute the factors from left to right, such that the factor F_x starting at position $p = |F_1| \cdots |F_{x-1}| + 1$ has length $\max(1, \mathsf{PLCP}[p])$.

3.2 Motivation of this Paper

Our search for more memory-efficient data structures computing plcpcomp or lex-parse stems from the fact that the actual computation is lightweight whereas the preprocessing computing the necessary text data structures is slow and causes the memory peaks in the implementation. The implementation in tudocomp [12] first builds SA, then Φ, and finally PLCP using the algorithm of [10]. The PLCP array is actually only needed for plcpcomp. For lex-parse, we can compute the factor lengths naively, such that we compute the longest common prefix of $T[i \mathinner{..}]$ with $T[\Phi[i] \mathinner{..}]$ in time linear to the number of compared characters. Hence, the computation of a factor F_x costs $\mathcal{O}(|F_x|)$ time, but the sum of all factor lengths (including the literal ones) is $\sum_x |F_x| = n$.

Figure 2 shows the peak memory requirement as a bar for each pre-computation step for lex-parse and plcpcomp. We can see that the actual factorization (rightmost bar) is fast and more memory-friendly (with respect to the additional memory requirement) than most pre-computation steps. The slowest part is the suffix array computation using divsufsort. We observe that the memory consumption to roughly two to three times the amount of RAM needed for SA, but for even less space, we need to think about what data we actually need for the factorization algorithms.

4 Space Requirements of plcpcomp

We are fine with PLCP in the representation of Sadakane [13] using $2n$ bits. We do not need the extra $o(n)$ bits for a rank/select-support data structure to gain constant time random access on PLCP, since we scan PLCP sequentially. For computing plcpcomp, we additionally maintain each interesting peak (along with its PLCP value) in the list L. We can bound the size of L with the following lemma:

Lemma 2. $|L| = \mathcal{O}(\min(\sqrt{n \lg n}, r))$, where r is the number of BWT runs.

Proof. The list L stores all interesting peaks between two different maximal peaks (or between the first position and the first maximal peak). Given an interesting peak dst with $\mathsf{PLCP}[dst]$, there is no peak j with $\mathsf{PLCP}[j] \geq \mathsf{PLCP}[dst]$ and $j < dst < j + \mathsf{PLCP}[j]$. In order to be added to L, the peak dst must not be a maximal peak, i.e., there must be a text position $j \in (dst \mathinner{..} dst + \mathsf{PLCP}[dst])$ and $\mathsf{PLCP}[j] > \mathsf{PLCP}[dst]$. The worst case is that $j = dst + 1$, $\mathsf{PLCP}[j] = \mathsf{PLCP}[dst] + 1$, and j is again an interesting peak that is not maximal. By induction, we may insert m interesting non-maximal peaks $\{j_i\}_{1 \leq i \leq m}$ into L with $j_i + 1 \leq j_{i+1}$ for $i \in [1 \mathinner{..} m-1]$ and $\mathsf{PLCP}[j_i] \geq i$ for $i \in [1 \mathinner{..} m]$.

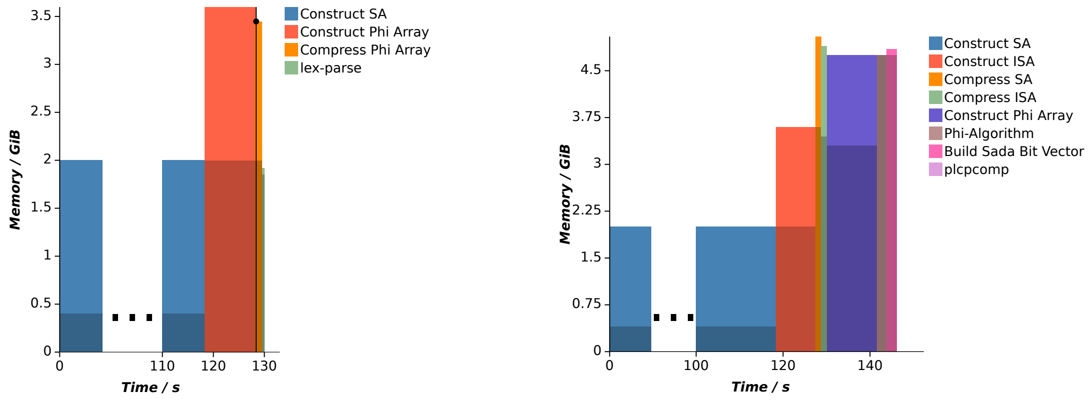

Figure 2: Computation of lex-parse (left) and plcpcomp (right) on the Pizza&Chili dataset PARA. The computation is divided into different phases. Each phase has a legend entry on the right side and is visualized as a block. Each block has a light and a dark shading. The dark shading is the memory consumption at the beginning of the phase, while the light shading on top is the additional maximum memory needed during that phase. For instance, the suffix array construction (construct SA) starts with the text already loaded into RAM, but additionally generates SA stored as a plain 32-bit array in RAM. The scale of the x-axis is linear, but the construction time of SA (identical for both factorizations) has been cut due to page limitations.

However, $\sum_{i=1}^{m} i \le \sum_{i=1}^{m} \mathsf{PLCP}[j_i] = \mathcal{O}(n \lg n)$ due to [14, Thm. 12], such that $m = \mathcal{O}(\sqrt{n \lg n})$. From the same reference [14, Sect. 4], we obtain that $m = \mathcal{O}(r)$. $\square$

There are also texts with a non-trivial lower bound on the size of L:

Lemma 3. There are texts of length n for which $|L| = \Theta(\sqrt{n})$.

Proof. For the proof, we use the following definition: Given an interval I, we define $\mathsf{b}(I)$ and $\mathsf{e}(I)$ to be the starting and the ending position of $I = [\mathsf{b}(I) \mathinner{.\,.} \mathsf{e}(I)]$, respectively.

Let $\Sigma := \{\sigma_1, \ldots, \sigma_m\}$ be an alphabet with $\sigma_1 > \sigma_2 > \ldots > \sigma_m$. Set $F_1 := \sigma_1$, and $F_i := \sigma_i F_{i-1} \sigma_i$ for $i \in [2 \mathinner{.\,.} m]$. Then our algorithm fills L with $\Theta(\sqrt{n})$ interesting peaks on processing the text $T := F_1 \cdots F_m = \sigma_1 \cdot \sigma_2\sigma_1\sigma_2 \cdot \sigma_3\sigma_2\sigma_1\sigma_2\sigma_3 \cdot \sigma_4 \cdots$.

In the following, we show that each text position $\mathsf{b}(F_i)$ with $i \in [1 \mathinner{.\,.} m-1]$ is an interesting peak, where $\mathsf{b}(F_i)$ and $\mathsf{e}(F_i)$ are the beginning and ending positions of the factor F_i within the factorization $T = F_1 \cdots F_m = T[\mathsf{b}(F_1) \mathinner{.\,.} \mathsf{e}(F_1)] \cdots T[\mathsf{b}(F_m) \mathinner{.\,.} \mathsf{e}(F_m)]$.

First, $|F_i| = 2i - 1$ for every $i \in [1 \mathinner{.\,.} m]$. Next, we show that $\Phi[\mathsf{b}(F_i)] = \mathsf{b}(F_{i+1}) + 1$ for each $i \in [1 \mathinner{.\,.} m-1]$. For that, we observe that

- $T[\mathsf{b}(F_i) \mathinner{.\,.}] = F_i F_{i+1} F_{i+2} F_{i+3} \cdots = F_i \sigma_{i+1} F_i \sigma_{i+1} F_{i+2} F_{i+3} \cdots$,
- $F_{i+j} = \sigma_{i+j} \cdots \sigma_{i+1} F_i \sigma_{i+1} \cdots \sigma_{i+j}$, and
- $T[\mathsf{b}(F_{i+j}) + j \mathinner{.\,.}] = F_i \sigma_{i+1} \cdots \sigma_{i+j} F_{i+j+1} F_{i+j+2} \cdots$ for all $j \in [0 \mathinner{.\,.} m-i]$.

Hence, all occurrences of the prefix $F_i\sigma_{i+1}$ of $T[\mathsf{b}(F_i) \mathinner{.\,.}]$ start at $\mathsf{b}(F_{i+j}) + j$ for $j \in [0 \mathinner{.\,.} m-i]$. One of these occurrences must be the prefix of the suffix that is the lexicographically predecessor of $T[\mathsf{b}(F_i) \mathinner{.\,.}]$, which is $T[\mathsf{b}(F_{i+1}) + 1 \mathinner{.\,.}]$ because of the following inequality.

$$
\begin{aligned}
T[\mathsf{b}(F_{i+j}) + j \mathinner{.\,.}] \prec T[\mathsf{b}(F_{i+1}) + 1 \mathinner{.\,.}] &= F_i \sigma_{i+1} F_{i+2} \cdots \\
&= F_i \sigma_{i+1} \sigma_{i+2} F_{i+1} \sigma_{i+2} \cdots \prec T[\mathsf{b}(F_i) \mathinner{.\,.}]
\end{aligned}
$$

for all $j \in [1 \mathinner{.\,.} m-i]$. Hence, $\Phi[\mathsf{b}(F_i)] = \mathsf{b}(F_{i+1}) + 1$ for each $i \in [1 \mathinner{.\,.} m-1]$. For each $i \in [1 \mathinner{.\,.} m-1]$ and all text positions $j \in [1 \mathinner{.\,.} n] \setminus \{\mathsf{b}(F_i)\}$, we have

$$
\begin{aligned}
\mathsf{lcp}(T[\mathsf{b}(F_i) \mathinner{.\,.}], T[j \mathinner{.\,.}]) \le \mathsf{PLCP}[\mathsf{b}(F_i)] &= \mathsf{lcp}(T[\mathsf{b}(F_i) \mathinner{.\,.}], T[\Phi[\mathsf{b}(F_i)] \mathinner{.\,.}]) \\
&= \mathsf{lcp}(T[\mathsf{b}(F_i) \mathinner{.\,.}], T[\mathsf{b}(F_{i+1}) + 1 \mathinner{.\,.}]) = |F_i| + 1 = 2i.
\end{aligned}
$$

In general, we obtain $\mathsf{PLCP}[\mathsf{b}(F_i)+j] = 2i - j$ for each $j \in [0\mathinner{.\,.}|F_i|-1]$. In particular, $\mathsf{PLCP}[\mathsf{e}(F_i)] = 2$ for each $i \in [1\mathinner{.\,.}m-1]$ since $T[\mathsf{e}(F_i)\mathinner{.\,.}] = \sigma_i\sigma_{i+1}\sigma_i\cdots$ and the occurrence of $\sigma_i\sigma_{i+1}\sigma_i$ is unique, while $\sigma_i\sigma_{i+1}\sigma_{i+2}$ $(\prec \sigma_i\sigma_{i+1}\sigma_i)$ occurs in each F_j with $j > i$. We conclude that the text positions $\mathsf{b}(F_i)$ are interesting peaks, for $i \in [1\mathinner{.\,.}m-1]$. Consequently, starting with $\mathsf{b}(F_1)$, the algorithm of Sect. 3.1 collects $\mathsf{b}(F_i)$, scans the next $\mathsf{PLCP}[\mathsf{b}(F_i)]$ text positions in which it finds $\mathsf{b}(F_{i+1})$ having a higher PLCP value as $\mathsf{b}(F_i)$, and thus needs to put $\mathsf{b}(F_i)$ into L. Finally, $\mathsf{b}(F_{m-1})$ is a maximum peak, since $T[\mathsf{b}(F_m)] = \sigma_m$ occurs only at $T[\mathsf{b}(F_m)]$ and at the last text position $\mathsf{e}(F_m)$ such that $\Phi[\mathsf{b}(F_m)] = \mathsf{e}(F_m)$ and $\mathsf{PLCP}[\mathsf{b}(F_m)] = 1$.

In total, the algorithm of Sect. 3.1 collects $m-2$ interesting peaks before finding the maximal peak at text position $\mathsf{b}(F_{m-1})$. Since $|F_i| = 2i-1$, we have $\sum_{i=1}^{m}|F_i| = \sum_{i=1}^{m}(2i-1) = n$, which holds for $m = \Theta(\sqrt{n})$. ☐

5 Sparse Φ Representation

With PLCP we can already compute the factor lengths of both parsings. However, we still require to compute the referred positions for outputting the parse. The referred position src_x of a referencing factor (src_x, ℓ_x) starting at position dst_x is $src_x = \Phi[dst_x]$, which can be computed if we have Φ available. For our purposes, it is sufficient to have only certain entries of Φ available: We call an entry $\Phi[i]$ *reducible* if $\Phi[i-1]+1 = \Phi[i]$, otherwise we call it *irreducible*. By storing only the *irreducible* entries of Φ in an array Φ_{S} and a bit vector B of length n marking whether the j-th text position is irreducible for each integer $j \in [1\mathinner{.\,.}n]$, we can access Φ with $\Phi[i] = \Phi_{\mathrm{S}}[B.\mathsf{rank}_1(i)] + i - B.\mathsf{select}_1[B.\mathsf{rank}_1(i)]$, given that the bit vector B is endowed with a rank/select-support. Kärkkäinen and Kempa [15, Lemma 3.3] show that $\mathsf{SA}[i]$ is an irreducible entry of Φ if $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$. Therefore, Φ_{S} has at most r entries, where r denotes the number of runs of the same character in BWT. See Table 1 for an example. To obtain this Φ representation, we present two space efficient solutions for computing Φ in memory. The first preprocessing approach works as follows: At the beginning, we compute Φ, then reduce Φ to Φ_{S} and B, and finally construct PLCP in linear time with Lemma 1. To compute Φ, we can use the algorithm BGone of Goto and Bannai [16], computing Φ from the text T in linear time with $\mathcal{O}(\sigma \lg n)$ additional working space on top of Φ stored in $n \lg n$ bits.

After this preprocessing, our algorithm computing $\mathsf{plcpcomp}$ runs in linear time using

$$\underbrace{r \lg n}_{\text{sparse }\Phi} + \underbrace{n + o(n)}_{\mathrm{B}} + \underbrace{2n}_{\mathsf{PLCP}} + \underbrace{\mathcal{O}(\min(\sqrt{n \lg n}, r) + 1) \lg n}_{L} = r \lg n + 3n + o(n)$$

bits of total space, instead of

$$\overbrace{n \lg n}^{\Phi} + \overbrace{2n}^{\mathsf{PLCP}} + \overbrace{\mathcal{O}(\min(\sqrt{n \lg n}, r) + 1) \lg n}^{L}$$

bits for conducting all computation with Φ represented as a plain array. For $\mathsf{lex\text{-}parse}$, we obtain the same space bounds without the space for L and PLCP.

Alternatively to first computing Φ, we can compute Φ_{S} and B directly with $\mathcal{O}(n \lg \sigma)$ additional space in $\mathcal{O}(n \log_\sigma^\epsilon n)$ time for a selectable constant $\epsilon > 0$. For that, we build the compressed suffix tree by the linear-time construction algorithm of Munro et al. [17]. It gives access to BWT and SA in constant and $\mathcal{O}(\log_\sigma^\epsilon n)$ time, respectively. We set $B[\mathsf{SA}[i]] = 1$ for all i with $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$, endow B with rank/select-support, and finally create Φ_{S} by setting $\Phi[\mathsf{SA}[i]] \leftarrow \mathsf{SA}[i-1]$ for all i with $\mathsf{BWT}[i] \neq \mathsf{BWT}[i-1]$. With this technique, the algorithm runs in slightly increased time $\mathcal{O}(n \lg^\epsilon n)$, but uses merely $\mathcal{O}(n \lg \sigma)$ bits of space. We obtain the main result of this paper:

Theorem 4. Given the sparse Φ representation, we can compute $\mathsf{lex\text{-}parse}$ in linear time with $r \lg n + n + o(n)$ bits of space. Having additionally the $2n$-bits representation of PLCP, we can also compute $\mathsf{plcpcomp}$ in linear time. Here, n is the length of the input text T and r is the number of runs of the BWT of T.

6 Future Work

The upper and lower bound shown respectively in Lemmas 2 and 3 are not tight. On the one hand, our analysis in Lemma 2 is based on the sum of all irreducible PLCP values. However, not all irreducible

PLCP values are considered as interesting peaks. A more detailed analysis on the sum of the LCP values of all interesting peaks may improve the upper bound. On the other hand, in Lemma 3, we did not exploit the fact that a factor may refer to positions that are covered by another factor referring back to parts of the previous factor. Here, building a long dependency chain could help to shrink the required length of the text to contain more interesting peaks.

While lex-parse as a greedy parsing has the smallest number of factors v among all other lexicographic parsings [2, Theorem 26], it is unknown whether there are upper or lower bounds that put plcpcomp in relation with v. Such a kind of relationship is also unknown towards plcpcomp's sibling lcpcomp [12], which uses different tie-breaking rules for selecting a candidate among all longest occurring substrings.

On the practical side, different choices for the factorization could improve the compression ratio. For instance, one could compute a second parsing that uses the inverse of Φ. For that, we use the array storing the longest common prefix of the i-th and the $(\Phi^{-1}[i])$-th suffix, which is PLCP shifted by one position.

We would also like to think of alternative ways to compute lexicographic parsings:

- For instance, the index data structure of Nishimoto and Tabei [18] built on the run-length compressed BWT allows to compute Φ^{-1} in constant time, and it seems possible to change their data structure to flip Φ^{-1} into Φ, and simulate a linear scan on the text by using the reverse of the LF-mapping.
- Instead of using Φ, we wonder how fast and space-efficient a combination of SA with a sampling of its inverse ISA could be. The sampling of [19] can store ISA in $\mathcal{O}(\epsilon^{-1}n)$ bits and provide access to ISA with $\mathcal{O}(1/\epsilon)$ time (provided SA is available). It can be computed in linear time using $\mathcal{O}(\lg n)$ bits of space.
- BGone is a modification of the SAIS algorithm. It computes Φ with $\mathcal{O}(\sigma \lg n)$ additional bits in linear time from the text. We think that it is possible to modify divsufsort to compute Φ instead of SA. Although divsufsort runs in $\mathcal{O}(n \lg n)$ time using $\mathcal{O}(\sigma^2 \lg n)$ bits, it is practically faster than SAIS for small alphabets.
- Finally, we wonder whether a plcpcomp-like scheme can be computed directly by modifying a suffix array construction algorithm computing simultaneously LCP: an idea could be to create referencing factors at positions whose LCP values are irreducible.

Acknowledgements

This research was funded by JSPS KAKENHI with grant numbers JP21H05847 and JP21K17701.

References

[1] P. Dinklage, J. Ellert, J. Fischer, D. Köppl, and M. Penschuck, "Bidirectional text compression in external memory," in *Proc. ESA*, 2019, pp. 41:1–41:16.

[2] G. Navarro, C. Ochoa, and N. Prezza, "On the approximation ratio of ordered parsings," *IEEE Trans. Inf. Theory*, vol. 67, no. 2, pp. 1008–1026, 2021.

[3] J. A. Storer and T. G. Szymanski, "Data compression via textural substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[4] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[5] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. FOCS*, 1989, pp. 549–554.

[6] D. R. Clark, "Compact Pat trees," Ph.D. dissertation, University of Waterloo, Canada, 1996.

[7] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," *J. Discrete Algorithms*, vol. 3, no. 2-4, pp. 143–156, 2005.

[8] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. CPM*, ser. LNCS, vol. 2089, 2001, pp. 181–192.

[9] K. Sadakane, "Succinct representations of lcp information and improvements in the compressed suffix arrays," in *Proc. SODA*, 2002, pp. 225–232.

[10] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit, "Engineering a compressed suffix tree implementation," *ACM Journal of Experimental Algorithmics*, vol. 14, 2009.

[11] D. Köppl, G. Navarro, and N. Prezza, "HOLZ: high-order entropy encoding of Lempel-Ziv factor distances," in *Proc. DCC*, 2022, to appear.
[12] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane, "Compression with the tudocomp framework," in *Proc. SEA*, ser. LIPIcs, vol. 75, 2017, pp. 13:1–13:22.
[13] K. Sadakane, "Compressed suffix trees with full functionality," *Theory Comput. Syst.*, vol. 41, no. 4, pp. 589–607, 2007.
[14] J. Kärkkäinen, D. Kempa, and M. Piatkowski, "Tighter bounds for the sum of irreducible LCP values," *Theor. Comput. Sci.*, vol. 656, pp. 265–278, 2016.
[15] J. Kärkkäinen and D. Kempa, "LCP array construction in external memory," *ACM Journal of Experimental Algorithmics*, vol. 21, no. 1, pp. 1.7:1–1.7:22, 2016.
[16] K. Goto and H. Bannai, "Space efficient linear time Lempel–Ziv factorization for small alphabets," in *Proc. DCC*, 2014, pp. 163–172.
[17] J. I. Munro, G. Navarro, and Y. Nekrich, "Space-efficient construction of compressed indexes in deterministic linear time," in *Proc. SODA*, 2017, pp. 408–424.
[18] T. Nishimoto and Y. Tabei, "Optimal-time queries on bwt-runs compressed indexes," in *Proc. ICALP*, ser. LIPIcs, vol. 198, 2021, pp. 101:1–101:15.
[19] J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Succinct representations of permutations and functions," *Theor. Comput. Sci.*, vol. 438, pp. 74–88, 2012.