Computing Lexicographic Parsings

Dominik Köppl

December 18, 2019

Abstract

We give a memory-friendly algorithm computing the compression scheme plcpcomp or lex-parse in linear or near-linear time.

1 Introduction

In this article, we focus on computing the compression schemes *plcpcomp* [2] and *lex-parse* [10] within low memory. Both schemes are macro schemes [13] like the well-known Lempel-Ziv-77 factorization [15]. While Lempel-Ziv-77 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.

2 Preliminaries

Text 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, where $1 \leq j \leq n$. 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 $V = T[i\,..\,j]$ and $W = T[j+1\,..]$. In the following, we take an element $T \in \Sigma^*$ with $|T| = n$, and call it ***text***. We stipulate that T ends with a sentinel $T[n] = \$ \notin \Sigma$ that is lexicographically smaller than every character of Σ.

Text Data Structures Let SA denote the ***suffix array*** [8] of T. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[\mathsf{SA}[i]\,..] \prec T[\mathsf{SA}[i+1]\,..]$ for all integers i with $1 \leq i \leq n-1$. Let ISA of T be the inverse of SA, i.e., $\mathsf{ISA}[\mathsf{SA}[i]] = i$ for every i with $1 \leq i \leq n$. The ***Burrows-Wheeler transform (BWT)*** [1] 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 with $1 \leq i \leq n$. The ***LCP array*** is an array with the property that $\mathsf{LCP}[i]$ is the length of the longest common prefix (LCP) of $T[\mathsf{SA}[i]\,..]$ and $T[\mathsf{SA}[i-1]\,..]$ for $i = 2, \ldots, n$. 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., $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. Figure 1 illustrates the introduced data structures.

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. [6], respectively. With SA, we can construct ISA in $\mathcal{O}(n)$ time by using the fact that SA is a permutation.

Lemma 2.1 ([11, 14])**.** PLCP can be represented by $2n + o(n)$ bits and can be constructed in $\mathcal{O}(n)$ time.

3 Parsing

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 triple (src_x, dst_x, ℓ_x) such that $F_x = T[dst_x\,..\,dst_x + \ell_x - 1] = T[src_x\,..\,src_x + \ell_x - 1]$ with $dst_x = \sum_{y=1}^{x-1} \ell_y$, or (b) F_x with $\ell_x := |F_x|$. We call the latter representation (b) of a factor ***literal***.

In [10, Sect. VI], Navarro and Prezza studied so-called *lexicographic parses*. A parse of T is called ***lexicographic*** if $T[src_x\,..\,] \prec T[dst_x\,..\,]$ for every non-literal factor. Here, we focus on a stricter class of those parses, where $src_x = \Phi[dst_x]$ holds for all non-literal factors. Note that $src_x = \Phi[dst_x]$ implies that $T[src_x\,..\,] = T[\mathsf{SA}[\mathsf{ISA}[dst_x]-1]\,..\,] \prec T[dst_x\,..\,]$ for $\mathsf{ISA}[dst_x] > 1$[1] Two lexicographic parses are lex-parse [10, Def. 11] and plcpcomp [2], on which we focus on the following.

The ***lex-parse*** is a parse $T = F_1, \ldots, F_z$ such that $F_x = T[dst_x\,..\,dst_x + \ell_x - 1]$ 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***-parse with a threshold $\xi > 0$ is recursively defined by replacing the longest reoccurring substring $T[dst_x\,..\,dst_x+\ell_x-1] = T[src_x\,..\,src_x+\ell_x-1]$ with $\ell \geq \xi$ 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. Dinklage et al. [2, Sect. 3] proposed an algorithm computing this parse in external memory. It can however be computed in memory like lex-parse when having PLCP available. For that, it detects so-called *peaks*. A text position dst is a ***peak*** if $\mathsf{PLCP}[dst] \geq \xi$ and $dst = 1$, $\mathsf{PLCP}[dst-1] < \mathsf{PLCP}[dst]$, or there is a referencing factor ending at $dst - 1$. A peak dst is called ***interesting*** if there is no text position j with $dst \in (j\,..\,j + \mathsf{PLCP}[j])$ and $\mathsf{PLCP}[j] \geq \mathsf{PLCP}[dst]$. An interesting peak dst is called ***maximal*** if there is no interesting peak j with $j \in (dst\,..\,dst + \mathsf{PLCP}[dst])$.

With these definitions, we can compute plcpcomp as follows: We linearly scan the text from left to right, adding interesting peaks to the list L. On finding a maximal peak dst, we can factorize $T[1\,..\,dst-1]$ by using the peaks stored in L and their associated PLCP values. This takes $\mathcal{O}(|L|) = \mathcal{O}(dst)$ time. We continue with the plcpcomp factorization of $T[dst + \mathsf{PLCP}[dst]\,..]$. In overall, this accumulates to $\mathcal{O}(n)$ time. Computing lex-parse is in fact easier, since we do not have to maintain L.

Space Analysis We compute PLCP in the representation of Sadakane [12] using $2n$ bits. We do not need the extra $o(n)$ bits for rank/select-support, 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 3.1. $|L| = \mathcal{O}(\min(\sqrt{n \lg n}, r))$, 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 with $dst < j < dst + \mathsf{PLCP}[dst]$ 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 $1 \leq i \leq m-1$ and $\mathsf{PLCP}[j_i] \geq i$ for $1 \leq i \leq m$.

However, $\sum_{i=1}^{m} i \leq \sum_{i=1}^{m} \mathsf{PLCP}[j_i] = \mathcal{O}(n \lg n)$ due to [5, Thm. 12], such that $m = \mathcal{O}(\sqrt{n \lg n})$. From the same reference [5, Sect. 4], we obtain that $m = \mathcal{O}(r)$. □

Lemma 3.2. 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 $I = [\mathsf{b}(I)\,..\,\mathsf{e}(I)]$, 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 $2 \leq i \leq m$. Then our algorithm fills L with $\Theta(\sqrt{n})$ interesting peaks on processing the text $T := F_1 \cdots F_m$. In the following, we show that each text position $\mathsf{b}(F_i)$ 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 $T = F_1 \cdots F_m = T[\mathsf{b}(F_1)\,..\,\mathsf{e}(F_1)] \cdots T[\mathsf{b}(F_m)\,..\,\mathsf{e}(F_m)]$.

First, $\Phi[\mathsf{b}(F_i)] = \mathsf{b}(F_{i+1}) + 1$ for each i with $1 \leq i \leq m-1$, since

- $T[\mathsf{b}(F_i)..] = F_i F_{i-1} \cdots = F_i \sigma_{i-1} F_i \sigma_{i-1} \cdots$ and
- $T[\mathsf{b}(F_{i+j}) + j..] = F_i \sigma_{i-1} \cdots \sigma_{i-j}$ for all j with $0 \leq j \leq i-1$

Hence, $T[\mathsf{b}(F_{i+j})+j..] \prec T[\mathsf{b}(F_{i+1})+1..] = F_i \sigma_{i-1} F_{i-2} \cdots = F_i \sigma_{i-1} \sigma_{i-2} F_{i-1} \sigma_{i-2} \cdots \prec T[\mathsf{b}(F_i)..]$ for all j with $i+2 \leq j \leq m$. For all positions $1 \leq j \leq n$, we have $\mathsf{lcp}(T[j..], T[\mathsf{b}(F_i)..]) \leq \mathsf{lcp}(T[\mathsf{b}(F_i)..], T[\mathsf{b}(T_{i+1})+1..]) = |F_i| + 1 = 2i$. Hence, $\mathsf{PLCP}[\mathsf{b}(F_i)] = 2i$ for each i with $1 \leq i \leq m-1$. Similarly, we obtain $\mathsf{PLCP}[\mathsf{b}(F_i) + j] = 2i - j$ for each j with $0 \leq j \leq |F_i|$ and $\mathsf{PLCP}[\mathsf{e}(F_i)] = 2$ for each i with $1 \leq i \leq m-1$.

[1] $\mathsf{ISA}[n] = 1$ since $T[n]\$$.

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	$
SA	22	21	16	19	17	6	1	8	13	3	10	20	15	18	5	7	12	2	9	14	4	11
ISA	7	18	10	21	15	6	16	8	19	11	22	17	9	20	13	3	5	14	4	12	2	1
Φ	6	12	13	14	18	17	5	1	2	3	4	7	8	9	20	21	19	15	16	10	22	11
LCP	0	0	1	1	3	5	4	7	2	4	5	0	2	2	4	5	3	5	6	1	3	4
PLCP	4	5	4	3	4	5	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0
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: Suffix array, its inverse, Φ, LCP array, PLCP array, 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. 4. If $\mathsf{BWT}[\mathsf{ISA}[i]] = \mathsf{BWT}[\mathsf{ISA}[i]-1]$, i.e., $T[i-1] = T[\Phi[i]-1]$, then $\Phi[i] = \Phi[i-1]+1$.

We conclude that the text positions $\mathsf{b}(F_i)$ are interesting peaks, for $1 \le i \le m-1$. Moreover, $\mathsf{b}(F_{m-1})$ is a maximum peak, since $T[\mathsf{b}(F_m)] = \sigma_1$ occurs only at $T[\mathsf{b}(F_m)]$ and at the last text position $\mathsf{e}(F_1)$ such that $\mathsf{PLCP}[\mathsf{b}(F_m)] = 1$.

Finally, the algorithm 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 $m = \Theta(\sqrt{n})$. □

4 Sparse Φ

We still require to compute the referred positions for outputting the parse. The referred position of a factor starting at position *dst* is $\Phi[dst]$, i.e., our task is to compute Φ. In the following, we present two space efficient solutions for computing Φ in memory.

Goto and Bannai [3] presented a linear-time algorithm computing Φ from the text with $\mathcal{O}(\sigma \lg n)$ additional working space on top of Φ stored in $n \lg n$ bits. In our scenario, 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 i with $1 \le j \le n$, we can access Φ with $\Phi[i] = \Phi_S[B.\mathsf{rank}_1(i)] + i - B.\mathsf{select}_1[B.\mathsf{rank}_1(i)]$, given that the bit vector B is endowed with a rank/select-support. Kärkkäinen and Kempa [4, 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 Fig. 1 for an example. With this technique, after computing the Φ algorithm with $n \lg n + \mathcal{O}(\sigma \lg n)$ bits of working space with Goto and Bannai's algorithm [3], our algorithm computing plcpcomp runs in linear time using

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

bits of space, instead of

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

bits when conducting all computation with Φ represented as a plain array. For lex-parse, we obtain the same space bounds without the space of L.

Alternatively, we can compute Φ_S and B directly with $\mathcal{O}(n \lg \sigma)$ additional space in $\mathcal{O}(n \lg^{\epsilon}_{\sigma} n)$ time. For that, we build the compressed suffix tree by the linear-time construction algorithm of Munro et al. [9]. It gives access to BWT and SA in constant and $\mathcal{O}(\lg^{\epsilon}_{\sigma} n)$ 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.

5 Future Work

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 $\mathcal{O}(n \lg n)$ using $\mathcal{O}(\sigma^2 \lg n)$ bits, it is practically faster than SAIS for small alphabets.

The upper bound and lower bound shown respectively in Lemmas 3.1 and 3.2 are not tight. On the one hand, our analysis in Lemma 3.1 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 3.2, 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 among all other lexicographic parse [10, Theorem 24], it is unknown whether there are upper or lower bounds that put plcpcomp in relation with the smallest number of factors a lexicographic parse can achieve.

On the practical side, different choices for factorization could improve the compression ratio. For instance, one could do a second run 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 wonder whether an 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.

References

[1] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

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

[3] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proc. DCC*, pages 163–172, 2014.

[4] J. Kärkkäinen and D. Kempa. LCP array construction in external memory. *ACM Journal of Experimental Algorithmics*, 21(1):1.7:1–1.7:22, 2016.

[5] J. Kärkkäinen, D. Kempa, and M. Piatkowski. Tighter bounds for the sum of irreducible LCP values. *Theor. Comput. Sci.*, 656:265–278, 2016.

[6] 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*, volume 2089 of *LNCS*, pages 181–192, 2001.

[7] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

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

[9] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. SODA*, pages 408–424, 2017.

[10] G. Navarro and N. Prezza. On the approximation ratio of greedy parsings. *CoRR*, abs/1803.09517, 2018.

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

[12] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

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

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

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