On the Computation of Longest Previous Non-overlapping Factors

Enno Ohlebusch(✉) and Pascal Weber

Institute of Theoretical Computer Science, Ulm University, 89069 Ulm, Germany
{Enno.Ohlebusch,Pascal-1.Weber}@uni-ulm.de

Abstract. The f-factorization of a string is similar to the well-known Lempel-Ziv (LZ) factorization, but differs from it in that the factors must be non-overlapping. There are two linear time algorithms that compute the f-factorization. Both of them compute the array of longest previous non-overlapping factors (LPnF-array), from which the f-factorization can easily be derived. In this paper, we present a simple algorithm that computes the LPnF-array from the LPF-array and an array prevOcc that stores positions of previous occurrences of LZ-factors. The algorithm has a linear worst-case time complexity if prevOcc contains leftmost positions. Moreover, we provide an algorithm that computes the f-factorization directly. Experiments show that our first method (combined with efficient LPF-algorithms) is the fastest and our second method is the most space efficient way to compute the f-factorization.

1 Introduction

The Lempel-Ziv (LZ) factorization [20] of a string has played an important role in data compression for more than 40 years and it is also the basis of important algorithms on strings, such as the detection of all maximal repetitions (runs) in a string [16] in linear time. Because of its importance in data compression, there is extensive literature on algorithms that compute the LZ-factorization and [1–4,10,12,13,18,19] is an incomplete list.

A variant of the LZ-factorization is the f-factorization, which played an important role in solving a long standing open problem: it enabled the development of the first linear time algorithm for seeds computation by Kociumaka et al. [15].

Definition 1. *Let $S = S[0..n-1]$ be a string of length n on an alphabet Σ. The* f-factorization $s_1 s_2 \cdots s_m$ *of S can be defined as follows. Given $s_1 s_2 \cdots s_{j-1}$, the next factor s_j is obtained by a case distinction on the character $c = S[i]$, where $i = |s_1 s_2 \cdots s_{j-1}|$:*

(a) if c does not occur in $s_1 s_2 \cdots s_{j-1}$ then $s_j = c$
(b) else s_j is the longest prefix of $S[i..n-1]$ that is a substring of $s_1 s_2 \cdots s_{j-1}$.

N. R. Brisaboa and S. J. Puglisi (Eds.): SPIRE 2019, LNCS 11811, pp. 372–381, 2019.
https://doi.org/10.1007/978-3-030-32686-9_26

The difference to the LZ-factorization is that the factors must be non-overlapping. There are two linear time algorithms that compute the f-factorization [5,6]. Both of them compute the LPnF-array (defined below), from which the f-factorization can be derived (in case (b), the factor s_j is the length $\mathsf{LPnF}[i]$ prefix of $S[i..n-1]$).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S[i]$	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
LPnF$[i]$	0	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1
LPF$[i]$	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
(rm)prevOcc$[i]$	$\perp$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(lm)prevOcc$[i]$	$\perp$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 1. The LPnF, LPF, and prevOcc arrays of the string $S = aaaaaaaaaaaaaaaa$.

Definition 2. *For a string S of length n, the* longest previous non-overlapping factor (LPnF) *array of size n is defined for $0 \leq i < n$ by*

$$\mathsf{LPnF}[i] = \max\{\ell \mid 0 \leq \ell \leq n-i; S[i..i+\ell-1] \text{ is a substring of } S[0..i-1]\}$$

In the following, we will give a simple algorithm that directly bases the computation of the LPnF-array on the LPF-array, which is used in several algorithms that compute the LZ-factorization. The LPF-array is defined by $(0 \leq i < n)$

$$\mathsf{LPF}[i] = \max\{\ell \mid 0 \leq \ell \leq n-i; S[i..i+\ell-1] \text{ is a substring of } S[0..i+\ell-2]\}$$

In data compression, we are not only interested in the length of the longest previous factor but also in a previous position at which it occurred (because otherwise decompression would be impossible). For an LPF-array, the positions of previous occurrences are stored in an array prevOcc. If $\mathsf{LPF}[i] = 0$, we set $\mathsf{prevOcc}[i] = \perp$ (for decompression, one can use the definition $\mathsf{prevOcc}[i] = S[i]$). Figure 1 depicts the LPF-array of $S = a^{16}$ and two of many possible instances of the prevOcc-array: one that stores the rightmost (rm) positions of occurrences of longest previous factors and one that stores the leftmost (lm) positions.

2 Computing LPnF from LPF

Algorithm 1 computes the LPnF-array by a right-to-left scan of the LPF-array and its prevOcc-array. The computation of an entry $\ell = \mathsf{LPnF}[i]$ is solely based on entries $\mathsf{LPF}[j]$ and $\mathsf{prevOcc}[j]$ with $j \leq i$. Consequently, after the calculation of ℓ, it can be stored in $\mathsf{LPF}[i]$. Since Algorithm 1 overwrites the LPF-array with the LPnF-array (and the prevOcc-array of LPF with the prevOcc-array of LPnF), no extra space is needed. Algorithm 1 is based on the following simple idea:

1. If the factor starting at position i and its previous occurrence starting at position $j = \mathsf{prevOcc}[i]$ do not overlap, then clearly $\mathsf{LPnF}[i] = \mathsf{LPF}[i]$.

Algorithm 1. Given LPF and its prevOcc-array, the algorithm computes LPnF and stores it in LPF.

```
1:  function computeLPnF(LPF,prevOcc)
2:      for i ← n − 1 downto 0 do
3:          if LPF[i] > 0 then                          ▷ hence prevOcc[i] ≠ ⊥
4:              j ← prevOcc[i]
5:              if j + LPF[i] > i then                  ▷ overlapping case
6:                  ℓ ← i − j
7:                  while LPF[j] > ℓ do                 ▷ hence prevOcc[j] ≠ ⊥
8:                      ℓ ← min{LPF[i], LPF[j]}
9:                      j ← prevOcc[j]
10:                     if j + ℓ ≤ i then               ▷ non-overlapping case
11:                         break
12:                     else                            ▷ overlapping case
13:                         ℓ ← i − j
14:                 prevOcc[i] ← j
15:                 LPF[i] ← ℓ
```

2. Otherwise, the length of the (currently best) previous non-overlapping factor is $\ell = i - j$. A longer previous non-overlapping factor exists if $\mathsf{LPF}[j] > \ell$ (note that $\mathsf{LPF}[i] > \ell$ holds): the prefix of $S[i..n-1]$ of length $\min\{\mathsf{LPF}[i], \mathsf{LPF}[j]\}$ also occurs at position $\mathsf{prevOcc}[j]$ and even if the two occurrences (starting at i and $\mathsf{prevOcc}[j]$) overlap, their non-overlapping part must be greater than ℓ because $\mathsf{prevOcc}[j] < j$.
3. Step 2 is repeated until there is no further candidate (condition of the while-loop in line 7) or the two occurrences under consideration do not overlap (line 11 of Algorithm 1).

On the one hand, the example in Fig. 1 shows that Algorithm 1 may have a quadratic run-time if it uses the prevOcc-array that stores the rightmost positions of previous occurrences. On the other hand, the next lemma proves that Algorithm 1 has a linear worst-case time complexity if it uses the prevOcc-array that stores the leftmost positions of previous occurrences. Its proof is based on the following notion: An integer p with $0 < p \leq |\omega|$ is called a period of $\omega \in \Sigma^+$ if $\omega[i] = \omega[i+p]$ for all $i \in \{0, 1, \ldots, |\omega| - p - 1\}$.

Lemma 1. *If* prevOcc *stores the leftmost positions of previous occurrences, then the else-case on line 12 in Algorithm 1 cannot occur.*

Proof. For a proof by contradiction, suppose that the else-case on line 12 in Algorithm 1 occurs for some i. We have $\mathsf{LPF}[i] > 0$, $j = \mathsf{prevOcc}[i]$ is the leftmost occurrence of the longest previous factor ω_i starting at i, and $j + \mathsf{LPF}[i] > i$. Suppose $\mathsf{LPF}[j] > i - j$, i.e., the while-loop is executed. Let $m = \min\{\mathsf{LPF}[i], \mathsf{LPF}[j]\}$ and $k = \mathsf{prevOcc}[j]$. If $m = \mathsf{LPF}[i]$, then it would follow that an occurrence of ω_i starts at k. This contradicts the fact that j is the leftmost occurrence of ω_i. Consequently, $m = \mathsf{LPF}[j] < \mathsf{LPF}[i]$. The else-case on line 12 occurs when $k + m > i$. This implies $k + m > j$ because $i > j$. Let ω_j be the longest previous

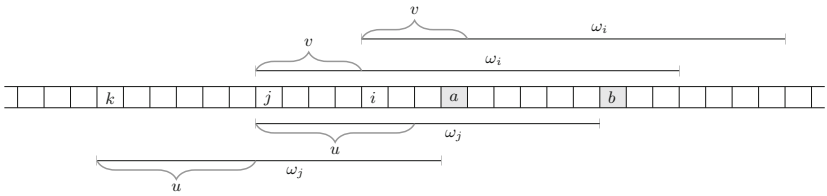

Fig. 2. Proof of Lemma 1: i, j, and k are positions, while a and b are characters.

factor starting at j. Let $a = S[k+m]$ (the character following the occurrence of ω_j starting at k) and $b = S[j+m]$ (the character following the occurrence of ω_j starting at j); see Fig. 2. By definition, $a \neq b$. We will derive a contradiction by showing that $a = b$ must hold in the else-case on line 12.

Since $k + m > j$, the occurrence of ω_j starting at k overlaps with the occurrence of ω_j starting at j. Let u be the non-overlapping part of the occurrence of ω_j starting at k, i.e., $u = S[k..j-1]$. Because the occurrence of ω_j starting at j has u as a prefix and overlaps with the occurrence of ω_j starting at k, it follows that $|u|$ is a period of ω_j; see Fig. 2. By a similar reasoning, one can see that $|v|$ is a period of ω_i, where $v = S[j..i-1]$. Since ω_j is a length m prefix of $S[j..n-1]$ and ω_i is a length $\mathsf{LPF}[i]$ prefix of $S[j..n-1]$, where $m = \mathsf{LPF}[j] < \mathsf{LPF}[i]$, it follows that ω_j is a prefix of ω_i. Hence $|v|$ is also a period of ω_j. In summary, both $|u|$ and $|v|$ are periods of ω_j. Fine and Wilf's theorem [8] states that if $|\omega_j| \geq |u| + |v| - gcd(|u|, |v|)$, then the greatest common divisor $gcd(|u|, |v|)$ of $|u|$ and $|v|$ is also a period of ω_j. Since $m = |\omega_j| \geq |u| + |v|$, the theorem is applicable. Let γ be the length $gcd(|u|, |v|)$ prefix of ω_j. It follows that $v = \gamma^q$ for some integer $q > 0$, hence $|\gamma|$ is a period of ω_i. Recall that $a = S[k+m]$ is the character $\omega_j[m - |u|] = \omega_i[m - |u|]$ and $b = S[j+m] = \omega_i[m]$. We derive $a = \omega_i[m - |u|] = \omega_i[m] = b$ because $|\gamma|$ is a period of ω_i and $|u|$ is a multiple of $|\gamma|$. This contradiction proves the lemma.

To the best of our knowledge, Abouelhoda et al. [1] first computed the LZ-factorization based on the suffix array (and the LCP-array) of S. Their algorithm computes the LPF-array and the $\mathsf{prevOcc}$-array that stores leftmost positions of previous occurrences of longest factors. So the combination of their algorithm and Algorithm 1 gives a linear-time algorithm that computes the LPnF-array. Subsequent work (e.g. [2–4,12,13,18]) concentrated on LZ-factorization algorithms that are faster in practice or more space-efficient (or both). Some of them also first compute the arrays LPF and $\mathsf{prevOcc}$, but their $\mathsf{prevOcc}$-arrays neither store leftmost nor rightmost occurrences (in fact, these algorithms are faster because they use lexicographically nearby suffixes–a *local* property–while being the leftmost occurrence is a *global* property). However, leftmost occurrences can easily be obtained by Algorithm 2. The algorithm is based on the following simple observation: If $\mathsf{LPF}[i] > 0$, $j = \mathsf{prevOcc}[i]$, and $\mathsf{LPF}[j] \geq \mathsf{LPF}[i]$, then $\mathsf{prevOcc}[j]$ is also the starting position of an occurrence of the factor starting at i. Since $\mathsf{prevOcc}[j] < j$, an occurrence left of j has been found.

Algorithm 2. Given LPF and its prevOcc-array, the algorithm computes the leftmost occurrence of each factor and stores it in prevOcc.

```
1: function COMPUTE-LEFTMOST-OCCURRENCE(LPF,prevOcc)
2:     for i ← 0 to n − 1 do
3:         if LPF[i] > 0 then                          ▷ hence prevOcc[i] ≠ ⊥
4:             j ← prevOcc[i]
5:             while LPF[j] ≥ LPF[i] do
6:                 j ← prevOcc[j]
7:             prevOcc[i] ← j
```

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$S[i]$	a	$\#_1$	a	a	$\#_2$	a	a	a	$\#_3$	a	a	a	a	$\#_4$
LPF$[i]$	0	0	1	1	0	2	2	1	0	3	3	2	1	0
prevOcc$[i]$	$\perp$	$\perp$	0	2	$\perp$	2	5	5	$\perp$	5	9	9	9	$\perp$
(lm)prevOcc$[i]$	$\perp$	$\perp$	0	0	$\perp$	2	2	0	$\perp$	5	5	2	0	$\perp$
iterations	0		0	1		0	1	2		0	1	2	3	

Fig. 3. LPF and prevOcc arrays of the string $S = a^1\#_1a^2\#_2a^3\#_3a^4\#_4$.

The while-loop in Algorithm 2 repeats this procedure until the leftmost occurrence is found. Note that the algorithm overwrites the prevOcc-array. Consequently, if its for-loop is executed for i, then for every $0 \le j < i$, prevOcc$[j]$ stores a leftmost position. The next example shows that Algorithm 2 is not linear in the worst-case. Consider the string $S = a^1\#_1a^2\#_2a^3\#_3a^4\#_4 \dots a^m\#_m$, where $m > 0$ and $\#_k$ are pairwise distinct separator symbols. Clearly, the length of S is $n = m + \sum_{k=1}^{m} k = m + m(m+1)/2 = m(m+3)/2$. If Algorithm 2 is applied to the arrays LPF and prevOcc in Fig. 3, it computes the leftmost (lm) prevOcc array and the number of iterations of its while-loop (last row in Fig. 3) is $\sum_{j=1}^{m-1}\sum_{k=1}^{j} k = (\sum_{j=1}^{m-1} j^2 + \sum_{j=1}^{m-1} j)/2 = (m-1)m(m+1)/6$.

3 Direct Computation of the f-Factorization

Algorithm 3 computes the f-factorization of S based on backward search on $T = S^{rev}$ and range maximum queries (RMQs) on the suffix array of T.[1] It uses ideas of [2, Algorithm CPS2] and [18, Algorithm LZ_bwd]. In fact, Algorithm 3 computes the right-to-left f-factorization of the reverse string S^{rev} of S. It is not difficult to see that $s_1s_2\cdots s_m$ is the (left-to-right) f-factorization of S if and only if $s_m^{rev}\cdots s_2^{rev}s_1^{rev}$ is the right-to-left f-factorization of S^{rev}. In this subsection, we assume a basic knowledge of suffix arrays (SA), the Burrows-Wheeler transform (BWT), and wavelet trees; see e.g. [7,18]. Given a substring ω of T, there is a suffix array interval $[sp..ep]$— called the ω-interval— so that ω is a prefix of every suffix T[SA$[k]..n]$ if and only if $sp \le k \le ep$. For a character c, the $c\omega$-interval can be computed by one backward search

[1] In the implementation, T is terminated by a special (EOF) symbol.

Algorithm 3. f-factorization of S based on backward search on $T = S^{rev}$

```
1: function COMPUTE-F-FACTORIZATION(SA, wavelet tree of BWT)
2:     i ← n − 1                                   ▷ T = S^rev[0..n − 1]
3:     while i ≥ 0 do
4:         sp ← 0; ep ← n; pos ← i; m ← ⊥
5:         repeat
6:             [sp..ep] ← backwardSearch(T[i], [sp..ep])
7:             max ← SA[RMQ(sp, ep)]
8:             if max ≤ pos then
9:                 break
10:            m ← max
11:            i ← i − 1
12:        until i < 0
13:        output pos − i
```

step $backwardSearch(c, [sp..ep])$; this takes $O(\log|\Sigma|)$ time if backward search is based on the wavelet tree of the BWT of T. A linear time preprocessing is sufficient to obtain a space-efficient data structure that supports RMQs in constant time; see [9] and the references therein. $\mathsf{RMQ}(sp, ep)$ returns the index of the maximum value among $\mathsf{SA}[sp], \mathsf{SA}[sp+1], \ldots, \mathsf{SA}[ep]$; hence $\mathsf{SA}[\mathsf{RMQ}(sp, ep)]$ is the maximum of these SA-values. Suppose Algorithm 3 has already computed $s_{j-1}^{rev} \cdots s_2^{rev} s_1^{rev}$ and let $i = n-(|s_{j-1}^{rev} \cdots s_2^{rev} s_1^{rev}|+1)$. It computes the next factor s_j^{rev} as follows. First, it stores the starting position i in a variable pos. In line 6, $backwardSearch(T[i], [0..n])$ returns the c-interval $[sp..ep]$, where $c = T[i]$. In line 7, the maximum max of $\mathsf{SA}[sp], \mathsf{SA}[sp+1], \ldots, \mathsf{SA}[ep]$ is determined. If $max = pos$ ($max < pos$ is impossible because $c = T[pos]$), then there is no occurrence of c in $T[pos+1..n]$, so that $s_j^{rev} = c$ (the algorithm outputs 0, meaning that the next factor is the next character). Otherwise, there is a previous occurrence of c at position $max > pos$ and the process is iterated, i.e., i is decremented by one and the new $T[i..pos]$-interval is computed etc. Consider an iteration of the repeat-loop, where $[sp..ep]$ is the $T[i..pos]$-interval for some $i < pos$. The repeat-loop must be terminated early (line 9) if $max \leq pos$ because then the rightmost occurrence of $T[i..pos]$ starts left of $pos+1$. In other words, $T[i..pos]$ is not a substring of $T[pos+1..n]$. Since the repeat-loop did not terminate in the previous iteration, $T[i+1..pos]$ is a substring of $T[pos+1..n]$ that has a previous occurrence at position $m > pos$, where m is the maximum SA-value of the previous iteration. So $s_j^{rev} = T[i+1..pos]$ and the algorithm outputs its length $|s_j^{rev}| = pos-(i+1)+1 = pos-i$, which coincides with $|s_j|$. Note that the algorithm can easily be extended so that it also computes positions of previous occurrences. Algorithm 3 has run-time $O(n \log|\Sigma|)$ because one backward search step takes $O(\log|\Sigma|)$ time.

Kolpakov and Kucherov [17] used the reversed f-factorization (they call it reversed LZ-factorization) for searching for gapped palindromes. The reversed f-factorization is defined by replacing case (b) in Definition 1 with: (b) else s_j is the longest prefix of $S[i..n-1]$ that is a substring of $(s_1 s_2 \cdots s_{j-1})^{rev}$. It is not

difficult so see that Algorithm 3 can be modified in such a way that it computes the reversed f-factorization of S in $O(n \log |\Sigma|)$ time (to find the next factor s_j, match prefixes of $S[i..n-1]$ against $T = S^{rev}$).

4 Experimental Results

Our implementation is based on the `sdsl-lite` library [11] and we experimentally compared it with the LPnF construction algorithm of Crochemore and Tischler [6], called CT-algorithm henceforth. Another LPnF construction algorithm is described in [5], but we could not find an implementation (this algorithm is most likely slower than the CT-algorithm because it uses *two* kinds of range minimum queries—one on the suffix array and one on the LCP-array—and range minimum queries are slow; see below). The experiments were conducted on a 64 bit Ubuntu 16.04.4 LTS system equipped with two 16-core Intel Xeon E5-2698v3 processors and 256 GB of RAM. All programs were compiled with the O3 option using g++ (version 5.4.1). Our programs are publically available.[2] The test data—the files dblp.xml, dna, english, and proteins—originate from the Pizza & Chili corpus.[3] In our first experiment, we computed the LPnF-array from the LPF-array. Three algorithms that compute the LPF-array were considered:

- AKO: algorithm by Abouelhoda et al. [1]
- LZ_OG: algorithm by Ohlebusch and Gog [18]
- KKP3: algorithm by Kärkkäinen et al. [14]

It is known that AKO is slower than the others, but in contrast to the other algorithms it calculates leftmost prevOcc-arrays. Thus, there was a slight chance that AKO in combination with Algorithm 1 is faster than LZ_OG or KKP3 in combination with Algorithm 1. However, our experiments showed that this is not the case. AKO is missing in Fig. 4 because the differences between the run-times of the other algorithms become more apparent without it. For the same reason, we did not take the suffix array construction time into account (note that each of the algorithms needs the suffix array). To find out whether or not it is advantageous to compute a leftmost prevOcc-array by Algorithm 2 before Algorithm 1 is applied, we also considered the combinations of LZ_OG and KKP3 with both algorithms. Figures 4 and 5 show the results of the first experiment. As one can see in Fig. 4, for real world data it seems disadvantageous to apply Algorithm 2 before Algorithm 1 because the overall run-time becomes slightly worse. However, for 'problematic' strings such as a^n and $a^n b$ it is advisable to use Algorithm 2: *With* it both LZ_OG and KKP3 outperformed the CT-algorithm (data not shown), but *without* it both did not terminate after 20 min. All in all, KKP3 in combination with Algorithms 1 and 2 is the best choice for the construction of the LPnF-array. In particular, it clearly outperforms the CT-algorithm in terms of run-time and memory usage.

[2] https://www.uni-ulm.de/in/theo/research/seqana/.
[3] http://pizzachili.dcc.uchile.cl.

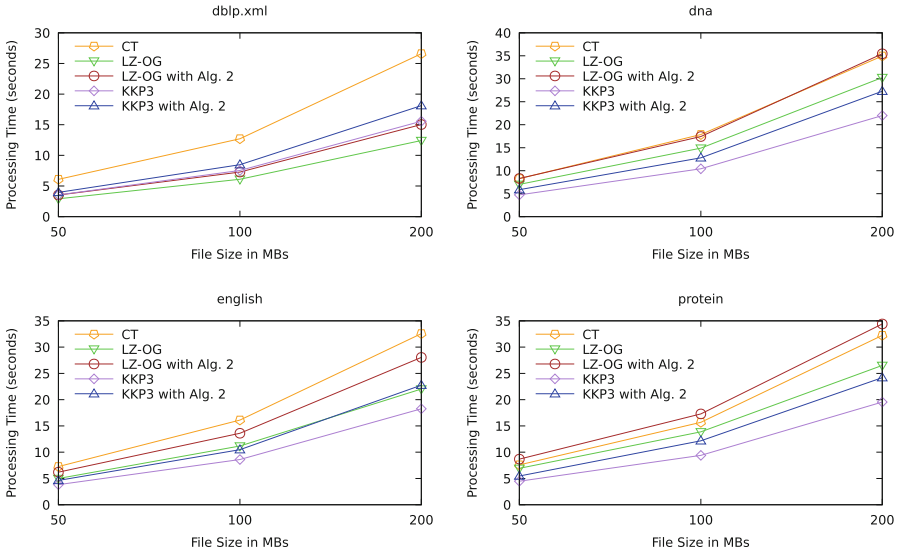

Fig. 4. Run-time comparison of LPnF-array construction (without suffix array construction, which on average takes 50% of the overall run-time)

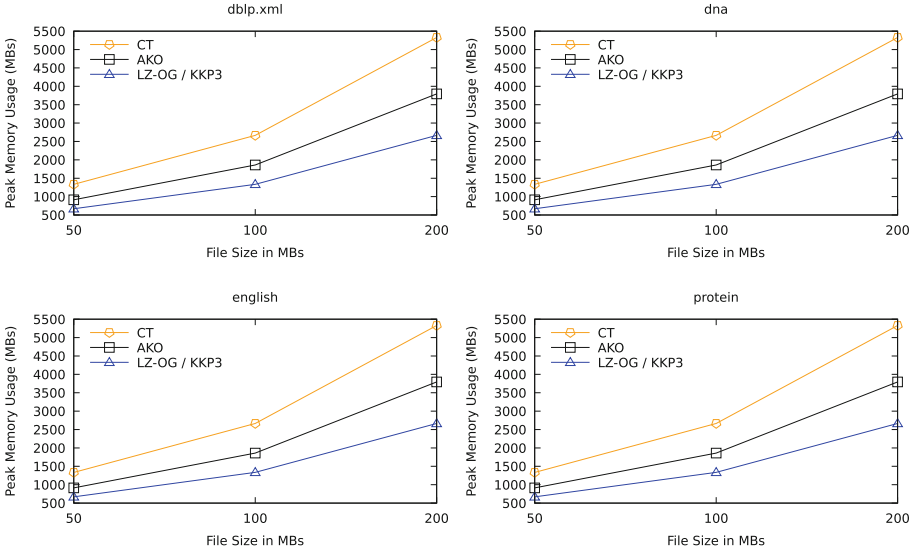

Fig. 5. Peak memory comparison of LPnF-array construction (with suffix array construction)

In the second experiment, we compared Algorithm 3—the only algorithm that computes the f-factorization directly—with the other algorithms (which first compute the LPnF-array and then derive the f-factorization from it). Algorithm 3 uses only 44% of the memory required by KKP3, but its run-time is by an order of magnitude worse (data not shown). We blame the range maximum queries for the rather bad run-time because these are slow in practice.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004)
2. Chen, G., Puglisi, S.J., Smyth, W.F.: Lempel-Ziv factorization using less time & space. Math. Comput. Sci. **1**(4), 605–623 (2008)
3. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Inf. Process. Lett. **106**(2), 75–80 (2008)
4. Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Waleń, T.: LPF computation revisited. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 158–169. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10217-2_18
5. Crochemore, M., Kubica, M., Iliopoulos, C.S., Rytter, W., Waleń, T.: Efficient algorithms for three variants of the LPF table. J. Discrete Algorithms **11**, 51–61 (2012)
6. Crochemore, M., Tischler, G.: Computing longest previous non-overlapping factors. Inf. Process. Lett. **111**, 291–295 (2011)
7. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
8. Fine, N.J., Wilf, H.S.: Uniqueness theorem for periodic functions. Proc. Am. Math. Soc. **16**, 109–114 (1965)
9. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
10. Fischer, J., I, T., Köppl, D., Sadakane, K.: Lempel-Ziv factorization powered by space efficient suffix trees. Algorithmica **80**(7), 2048–2081 (2018)
11. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_28
12. Goto, K., Bannai, H.: Simpler and faster Lempel Ziv factorization. In: Proceedings of 23rd Data Compression Conference, pp. 133–142. IEEE Computer Society (2013)
13. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel-Ziv factorization: simple, fast, small. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 189–200. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38905-4_19
14. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lazy Lempel-Ziv factorization algorithms. ACM J. Exp. Algorithmics **21**(2), Article 2.4 (2016)
15. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for seeds computation. In: Proceedings of 23rd Symposium on Discrete Algorithms, pp. 1095–1112 (2012)

16. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proceedings of 40th Annual IEEE Symposium on Foundations of Computer Science, pp. 596–604 (1999)
17. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theor. Comput. Sci. **410**(51), 5365–5373 (2009)
18. Ohlebusch, E., Gog, S.: Lempel-Ziv factorization revisited. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 15–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21458-5_4
19. Policriti, A., Prezza, N.: LZ77 computation based on the run-length encoded BWT. Algorithmica **80**(7), 1986–2011 (2018)
20. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)