Linear Time Lempel-Ziv Factorization: Simple, Fast, Small*

Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi

Department of Computer Science,
University of Helsinki
Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

Abstract. Computing the LZ factorization (or LZ77 parsing) of a string is a computational bottleneck in many diverse applications, including data compression, text indexing, and pattern discovery. We describe new linear time LZ factorization algorithms, some of which require only $2n \log n + \mathrm{O}(\log n)$ bits of working space to factorize a string of length n. These are the most space efficient linear time algorithms to date, using $n \log n$ bits less space than any previous linear time algorithm. The algorithms are also simple to implement, very fast in practice, and amenable to streaming implementation.

1 Introduction

In the 35 years since its discovery the LZ77 factorization of a string — named after its authors Abraham Lempel and Jacob Ziv, and the year 1977 in which it was published — has been applied all over computer science. The first uses of LZ77 were in data compression, and to this day it lies at the heart of efficient and widely used file compressors, like `gzip` and `7zip`. LZ77 is also important as a *measure* of compressibility. For example, its size is a lower bound on the size of the smallest context-free grammar that represents a string [2]. Our particular motivation is the construction of compressed full-text indexes [15], several recent and powerful instances of which are based on LZ77 [8,7,14]. In all these applications (and in most of the many others we have not listed) computation of the factorization is a time- and space-bottleneck in practice.

Related work. There exists a variety of worstcase linear time algorithms to compute the LZ factorization [1,9,16]. All of them require at least $3n \log n$ bits of working space[1] in the worstcase. The most space efficient linear time algorithm is due to Chen et al. [3]. By overwriting the suffix array it achieves a working space of $(2n + s) \log n$ bits, where s is the maximal size of the stack used in the algorithm. However, in the worstcase $s = \Theta(n)$. Another space efficient solution

* This research is partially supported by Academy of Finland grants 118653 (ALGODAN) and 250345 (CoECGR).

[1] Working space excludes input string, output factorization, and $\mathrm{O}(\log n)$ terms.

J. Fischer and P. Sanders (Eds.): CPM 2013, LNCS 7922, pp. 189–200, 2013.

requiring $(2n+\sqrt{n})\log n$ bits of space in the worstcase is from [6] but it computes only the lengths of LZ77 factorization phrases. It can be extended to compute the full parsing at the cost of extra $n \log n$ bits.

All of these algorithms rely on the suffix array, which can be constructed in $O(n)$ time and using $(1+\epsilon)n \log n$ bits of space (in addition to the input string but including the output of size $n \log n$ bits) [11]. This raises the question of whether the space complexity of linear time LZ77 factorization can be reduced from $3n \log n$ bits. In this paper, we answer the question in the affirmative by describing a linear time algorithm using $2n \log n$ bits.

In terms of practical performance, the fastest linear time LZ factorization algorithms are the very recent ones by Goto and Bannai [9], all using at least $3n \log n$ bits of working space. Other candidates for the fastest algorithms are described by Kempa and Puglisi [13]. Due to nearly simultaneous publication, no comparison between them exists so far. Experiments in this paper put the algorithms of Kempa and Puglisi slightly ahead. Their algorithms are also very space efficient; one of them uses $2n \log n + n$ bits of working space and others even less. However, their worstcase time complexity is $\Theta(n \log \sigma)$ for an alphabet of size σ. More details about these algorithms are given in Section 2.

Our contribution. We describe two linear time algorithms for LZ factorization. The first algorithm uses $3n \log n$ bits of working space and can be seen as a reorganization of an algorithm by Goto and Bannai [9]. However, this reorganization makes it smaller and faster. In our experiments, this is the fastest of all algorithms when the input is not highly repetitive.

The second algorithm reduces the working space to $2n \log n$ bits, which is at least $n \log n$ bits less than any previous linear time algorithm uses in the worstcase. The space reduction does not come at a great cost in performance. The algorithm is the fastest on some inputs and never far behind the fastest. It relies on novel combinatorial observations that might be of independent interest.

Both algorithms share several nice features. They are simple and easy to implement; they are alphabet-independent, using only character comparisons to access the input; and they make just one sequential pass over the suffix array, enabling streaming from disk. Our experiments show that streaming not only reduces the working space by a further $n \log n$ bits, but also speeds up the computation when the time for reading inputs from disk is taken into account.

2 Preliminaries

Strings. Throughout we consider a string $\mathsf{X} = \mathsf{X}[1..n] = \mathsf{X}[1]\mathsf{X}[2]\ldots\mathsf{X}[n]$ of $|\mathsf{X}| = n$ symbols drawn from an ordered alphabet of size σ.

For $i = 1, \ldots, n$ we write $\mathsf{X}[i..n]$ to denote the *suffix* of X of length $n - i + 1$, that is $\mathsf{X}[i..n] = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n]$. We will often refer to suffix $\mathsf{X}[i..n]$ simply as "suffix i". Similarly, we write $\mathsf{X}[1..i]$ to denote the *prefix* of X of length i. We write $\mathsf{X}[i..j]$ to represent the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$ of X that starts at position i and ends at position j. Let $\mathsf{lcp}(i, j)$ denote the length of the longest-common-prefix of suffix i and suffix j. For example, in the string $\mathsf{X} = zzzzzipzip$,

$\mathsf{lcp}(2,5) = 1 = |z|$, and $\mathsf{lcp}(5,8) = 3 = |zip|$. For technical reasons we define $\mathsf{lcp}(i,0) = \mathsf{lcp}(0,i) = 0$ for all i.

Suffix Arrays. The suffix array SA is an array $\mathsf{SA}[1..n]$ containing a permutation of the integers $1..n$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n]$ is the j^{th} suffix of X in ascending lexicographical order. The inverse suffix array ISA is the inverse permutation of SA, that is $\mathsf{ISA}[i] = j$ iff $\mathsf{SA}[j] = i$. Conceptually, $\mathsf{ISA}[i]$ tells us the position of suffix i in SA.

The array $\Phi[0..n]$ (see [12]) is defined by $\Phi[i] = \mathsf{SA}[\mathsf{ISA}[i]-1]$, that is, the suffix $\Phi[i]$ is the immediate lexicographical predecessor of the suffix i. For completeness and for technical reasons we define $\Phi[\mathsf{SA}[1]] = 0$ and $\Phi[0] = \mathsf{SA}[n]$ so that Φ forms a permutation with one cycle.

LZ77. The LZ77 factorization uses the notion of a *longest previous factor* (LPF). The LPF at position i in X is a pair (p_i, ℓ_i) such that, $p_i < i$, $\mathsf{X}[p_i..p_i + \ell_i - 1] = \mathsf{X}[i..i + \ell_i - 1]$ and $\ell_i > 0$ is maximized. In other words, $\mathsf{X}[i..i + \ell_i - 1]$ is the longest prefix of $\mathsf{X}[i..n]$ which also occurs at some position $p_i < i$ in X. If $\mathsf{X}[i]$ is the leftmost occurrence of a symbol in X then such a pair does not exist. In this case we define $p_i = \mathsf{X}[i]$ and $\ell_i = 0$. Note that there may be more than one potential p_i, and we do not care which one is used.

The LZ77 factorization (or LZ77 parsing) of a string X is then just a greedy, left-to-right parsing of X into longest previous factors. More precisely, if the jth LZ factor (or *phrase*) in the parsing is to start at position i, then we output (p_i, ℓ_i) (to represent the jth phrase), and then the $(j+1)$th phrase starts at position $i + \ell_i$, unless $\ell_i = 0$, in which case the next phrase starts at position $i + 1$. We call a factor (p_i, ℓ_i) *normal* if it satisfies $l_i > 0$ and *special* otherwise. The number of phrases in the factorization is denoted by z.

For the example string $\mathsf{X} = zzzzzipzip$, the LZ77 factorization produces:

$$(z,0), (1,4), (i,0), (p,0), (5,3).$$

The second and fifth factors are normal, and the other three are special.

NSV/PSV. The LPF pairs can be computed using *next and previous smaller values* (NSV/PSV) defined as

$$\mathsf{NSV}_{\mathsf{lex}}[i] = \min\{j \in [i+1..n] \mid \mathsf{SA}[j] < \mathsf{SA}[i]\}$$
$$\mathsf{PSV}_{\mathsf{lex}}[i] = \max\{j \in [1..i-1] \mid \mathsf{SA}[j] < \mathsf{SA}[i]\}.$$

If the set on the right hand side is empty, we set the value to 0. Further define

$$\mathsf{NSV}_{\mathsf{text}}[i] = \mathsf{SA}[\mathsf{NSV}_{\mathsf{lex}}[\mathsf{ISA}[i]]] \quad (1)$$
$$\mathsf{PSV}_{\mathsf{text}}[i] = \mathsf{SA}[\mathsf{PSV}_{\mathsf{lex}}[\mathsf{ISA}[i]]]. \quad (2)$$

If $\mathsf{NSV}_{\mathsf{lex}}[\mathsf{ISA}[i]] = 0$ ($\mathsf{PSV}_{\mathsf{lex}}[\mathsf{ISA}[i]] = 0$) we set $\mathsf{NSV}_{\mathsf{text}}[i] = 0$ ($\mathsf{PSV}_{\mathsf{text}}[i] = 0$).

If (p_i, ℓ_i) is a normal factor, then either $p_i = \mathsf{NSV}_{\mathsf{text}}[i]$ or $p_i = \mathsf{PSV}_{\mathsf{text}}[i]$ is always a valid choice for p_i [4]. To choose between the two (and to compute the ℓ_i component), we have to compute $\mathsf{lcp}(i, \mathsf{NSV}_{\mathsf{text}}[i])$ and $\mathsf{lcp}(i, \mathsf{PSV}_{\mathsf{text}}[i])$ and choose the larger of the two. This is given as a procedure `LZ-Factor` in Fig. 1.

Lazy LZ Factorization. The fastest LZ factorization algorithms in practice are from recent papers by Kempa and Puglisi [13] and Goto and Bannai [9]. A common feature between them is a lazy evaluation of LCP values: $\mathsf{lcp}(i, \mathsf{NSV}_{\mathsf{text}}[i])$ and $\mathsf{lcp}(i, \mathsf{PSV}_{\mathsf{text}}[i])$ are computed only when i is a starting position of a phrase. The values are computed by a plain character-by-character comparison of the suffixes, but it is easy to see that the total time complexity is $\mathrm{O}(n)$. This is in contrast to most previous algorithms that compute the LCP values for every suffix using more complicated techniques. The new algorithms in this paper use lazy evaluation too.

Goto and Bannai [9] describe algorithms that compute and store the full set of NSV/PSV values. One of their algorithms, BGT, computes the $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ arrays with the help of the Φ array. The LZ factorization is then easily computed by repeatedly calling `LZ-Factor`. Two other algorithms, BGS and BGL, compute the $\mathsf{NSV}_{\mathsf{lex}}$ and $\mathsf{PSV}_{\mathsf{lex}}$ arrays and use them together with SA and ISA to simulate $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ as in Eqs. (1) and (2). All three algorithms run in linear time and they use $3n \log n$ (BGT), $4n \log n$ (BGL) and $(4n+s) \log n$ (BGS) bits of working space, where s is the size of the stack used by BGS. In the worst case $s = \Theta(n)$. The algorithms for computing the NSV/PSV values are not new but come from [16] (BGT) and from [4] (BGL and BGS). However, the use of lazy LCP evaluation makes the algorithms of Goto and Bannai faster in practice than earlier algorithms.

Kempa and Puglisi [13] extend the lazy evaluation to the NSV/PSV values too. Using ISA and a small data structure that allows arbitrary NSV/PSV queries over SA to be answered quickly, they compute $\mathsf{NSV}_{\mathsf{text}}[i]$ and $\mathsf{PSV}_{\mathsf{text}}[i]$ only when i is a starting position of a phrase. The approach requires $(2+1/b)n \log n$ bits of working space and $\mathrm{O}(n+zb+z\log(n/b))$ time, where b is a parameter controlling a space-time tradeoff in the NSV/PSV data structure. If we set $b = \log n$, and given $z = \mathrm{O}(n/\log_\sigma n)$, then in the worstcase the algorithm requires $\mathrm{O}(n \log \sigma)$ time, and $2n \log n + n$ bits of space. Despite the superlinear time complexity, this algorithm (ISA9) is both faster and more space efficient than earlier linear time algorithms. Kempa and Puglisi also show how to reduce the space to $(1+\epsilon)n \log n + n + \mathrm{O}(\sigma \log n)$ bits by storing a succinct representation of ISA (algorithms ISA6r and ISA6s). Because of the lazy evaluation, these algorithms are especially fast when the resulting LZ factorization is small.

Optimized Parsing. Fig. 1 shows two versions of the basic parsing procedure. The standard version is essentially how the computation is done in all prior implementations using lazy LZ factorization. The optimized version is the first, small contribution of this paper. It is based on the observation that $\mathsf{lcp}(nsv, psv) = \min(\mathsf{lcp}(i, nsv), \mathsf{lcp}(i, psv))$ and performs $\mathsf{lcp}(nsv, psv)$ fewer symbol comparisons than the standard version.

3 $3n \log n$-Bit Algorithm

Our first algorithm is closely related to the algorithms of Goto and Bannai [9], particularly BGT and BGS. It first computes the $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ arrays and

Procedure `LZ-Factor`(i, nsv, psv)
1: $\ell_{nsv} \leftarrow \mathsf{lcp}(i, nsv)$
2: $\ell_{psv} \leftarrow \mathsf{lcp}(i, psv)$
3: **if** $\ell_{nsv} > \ell_{psv}$ **then**
4: $\quad (p, \ell) \leftarrow (nsv,\ \ell_{nsv})$
5: **else**
6: $\quad (p, \ell) \leftarrow (psv,\ \ell_{psv})$
7: **if** $\ell = 0$ **then** $p \leftarrow \mathsf{X}[i]$
8: **output** factor (p, ℓ)
9: **return** $i + \max(\ell, 1)$

Procedure `LZ-Factor`(i, nsv, psv)
1: $\ell \leftarrow \mathsf{lcp}(nsv, psv)$
2: **if** $\mathsf{X}[i + \ell] = \mathsf{X}[nsv + \ell]$ **then**
3: $\quad \ell \leftarrow \ell + 1$
4: $\quad (p, \ell) \leftarrow (nsv,\ \ell + \mathsf{lcp}(i + \ell, nsv + \ell))$
5: **else**
6: $\quad (p, \ell) \leftarrow (psv,\ \ell + \mathsf{lcp}(i + \ell, psv + \ell))$
7: **if** $\ell = 0$ **then** $p \leftarrow \mathsf{X}[i]$
8: **output** factor (p, ℓ)
9: **return** $i + \max(\ell, 1)$

Fig. 1. The standard (left) and optimized (right) versions of the basic procedure for computing a phrase starting at a position i given $nsv = \mathsf{NSV}_{\mathsf{text}}[i]$ and $psv = \mathsf{PSV}_{\mathsf{text}}[i]$. The return value is the starting position of the next phrase.

uses them for lazy LZ factorization similarly to the BGT algorithm (lines 11–13 in Fig. 2). However, the NSV/PSV values are computed using the technique of the BGS algorithm, which comes originally from [4].

The NSV/PSV computation scans the suffix array while maintaining a stack of suffixes, which are always in double ascending order: both in ascending lexicographical order and in ascending order of text position. The following are equivalent characterizations of the stack content after processing suffix $\mathsf{SA}[i]$:

- $\mathsf{SA}[i]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[i]]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[i]]]$, ..., 0
- 0 and all $\mathsf{SA}[k]$, $k \in [1..i]$, such that $\mathsf{SA}[k] = \min \mathsf{SA}[k..i]$
- 0 and all $\mathsf{SA}[k]$, $k \in [1..i]$, such that $\mathsf{NSV}_{\mathsf{text}}[\mathsf{SA}[k]] \notin \mathsf{SA}[k + 1..i]$.

Our version of this NSV/PSV computation is shown on lines 1–10 in Fig. 2. It differs from the BGS algorithm of Goto and Bannai in the following ways:

1. We write the NSV/PSV values to the text ordered arrays $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ instead of the lexicographically ordered arrays $\mathsf{NSV}_{\mathsf{lex}}$ and $\mathsf{PSV}_{\mathsf{lex}}$. Because of this, the second phase of the algorithm does not need the SA and ISA arrays.
2. BGS uses a dynamically growing separate stack while we overwrite the suffix array with the stack. This is possible because the stack is never larger than the already scanned part of SA, which we do not need any more (see above). The worst case size of the stack is $\Theta(n)$ (but it is almost always much smaller in practice).
3. Similar to the algorithms of Goto and Bannai, we store the arrays $\mathsf{PSV}_{\mathsf{text}}$ and $\mathsf{NSV}_{\mathsf{text}}$ interleaved so that the values $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$ are next to each other. We compute the PSV value when popping from the stack instead of when pushing to the stack as BGS does. This way $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$ are computed and written at the same time which can reduce the number of cache misses.

Because of these differences, our algorithm uses between $n \log n$ and $2n \log n$ bits less space and is significantly faster than BGS, which is the fastest of the algorithms in [9].

Algorithm KKP3
1: $\mathsf{SA}[0] \leftarrow 0$ // bottom of stack
2: $\mathsf{SA}[n+1] \leftarrow 0$ // empties the stack at end
3: $top \leftarrow 0$ // top of stack
4: **for** $i \leftarrow 1$ **to** $n+1$ **do**
5: **while** $\mathsf{SA}[top] > \mathsf{SA}[i]$ **do**
6: $\mathsf{NSV}_{\mathsf{text}}[\mathsf{SA}[top]] \leftarrow \mathsf{SA}[i]$
7: $\mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[top]] \leftarrow \mathsf{SA}[top-1]$
8: $top \leftarrow top - 1$ // pop from stack
9: $top \leftarrow top + 1$
10: $\mathsf{SA}[top] \leftarrow \mathsf{SA}[i]$ // push to stack
11: $i \leftarrow 1$
12: **while** $i \leq n$ **do**
13: $i \leftarrow \texttt{LZ-Factor}(i, \mathsf{NSV}_{\mathsf{text}}[i], \mathsf{PSV}_{\mathsf{text}}[i])$

Fig. 2. LZ factorization using $3n \log n$ bits of working space (the arrays SA, $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$)

4 $2n \log n$-Bit Algorithm

Our second algorithm reduces space by computing and storing only the NSV values at first. It then computes the PSV values from the NSV values on the fly. As a side effect, the algorithm also computes the Φ array! This is a surprising reversal of direction compared to some algorithms that compute NSV and PSV values from Φ [16,9].

For $t \in [0..n]$, let $\mathcal{X}_t = \{\mathsf{X}[i..n] \mid i \leq t\}$ be the set of suffixes starting at or before position t. Let Φ_t be Φ restricted to $\mathcal{X}_t$, that is, for $i \in [1..t]$, suffix $\Phi_t[i]$ is the immediate lexicographical predecessor of suffix i among the suffixes in $\mathcal{X}_t$. In particular, $\Phi_n = \Phi$. As with the full Φ, we make Φ_t a complete unicyclic permutation by setting $\Phi_t[i_{\min}] = 0$ and $\Phi_t[0] = i_{\max}$, where $i_{\min}$ and $i_{\max}$ are the lexicographically smallest and largest suffixes in $\mathcal{X}_t$. We also set $\Phi_0[0] = 0$. A useful way to view Φ_t is as a circular linked list storing $\mathcal{X}_t$ in the descending lexicographical order with $\Phi_t[0]$ as the head of the list.

Now consider computing Φ_t given Φ_{t-1}. We need to insert a new suffix t into the list, which can be done using standard insertion into a singly-linked list provided we know the position. It is easy to see that t should be inserted between $\mathsf{NSV}_{\mathsf{text}}[t]$ and $\mathsf{PSV}_{\mathsf{text}}[t]$. Thus

$$\Phi_t[i] = \begin{cases} t & \text{if } i = \mathsf{NSV}_{\mathsf{text}}[t] \\ \mathsf{PSV}_{\mathsf{text}}[t] & \text{if } i = t \\ \Phi_{t-1}[i] & \text{otherwise} \end{cases}$$

and furthermore

$$\mathsf{PSV}_{\mathsf{text}}[t] = \Phi_{t-1}[\mathsf{NSV}_{\mathsf{text}}[t]] \ .$$

The pseudocode for the algorithm is given in Fig 3. The NSV values are computed essentially the same way as in the first algorithm (lines 1–9) and stored in the

array Φ. In the second phase, the algorithm maintains the invariant that after t rounds of the loop on lines 12–18, $\Phi[0..t] = \Phi_t$ and $\Phi[t+1..n] = \mathsf{NSV}_{\mathsf{text}}[t+1..n]$.

Algorithm `KKP2`

```
1:  SA[0] ← 0        // bottom of stack
2:  SA[n + 1] ← 0    // empties the stack at end
3:  top ← 0          // top of stack
4:  for i ← 1 to n + 1 do
5:      while SA[top] > SA[i] do
6:          Φ[SA[top]] ← SA[i]    // Φ[SA[top]] = NSV_text[SA[top]]
7:          top ← top − 1         // pop from stack
8:      top ← top + 1
9:      SA[top] ← SA[i]           // push to stack
10: Φ[0] ← 0
11: next ← 1
12: for t ← 1 to n do
13:     nsv ← Φ[t]
14:     psv ← Φ[nsv]
15:     if t = next then
16:         next ← LZ-Factor(t, nsv, psv)
17:     Φ[t] ← psv
18:     Φ[nsv] ← t
```

Fig. 3. LZ factorization using $2n \log n$ bits of working space (the arrays SA and Φ)

An interesting observation about the algorithm is that the second phase computes Φ from $\mathsf{NSV}_{\mathsf{text}}$ without any additional information. Since the suffix array can be computed from Φ, the $\mathsf{NSV}_{\mathsf{text}}$ array alone contains sufficient information to reconstruct the suffix array.

5 Getting Rid of the Stack

The above algorithms overwrite the suffix array with the stack, which can be undesirable. First, we might need the suffix array later for another purpose. Second, since the algorithms make just one sequential pass over the suffix array, we could stream the suffix array from disk to further reduce the memory usage. In this section, we describe variants of our algorithms that do not overwrite SA (and still make just one pass over it).

The idea, already used in the BGL algorithm of Goto and Bannai [9], is to replace the stack with $\mathsf{PSV}_{\mathsf{text}}$ pointers. As observed in Section 3, if j is the suffix on the top of the stack, then the next suffixes in the stack are $\mathsf{PSV}_{\mathsf{text}}[j]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{PSV}_{\mathsf{text}}[j]]$, etcetera. Thus given $\mathsf{PSV}_{\mathsf{text}}$ we do not need an explicit stack at all. Both of our algorithms can be modified to exploit this:

– In KKP3, we need to compute the $\mathsf{PSV}_{\mathsf{text}}$ values when pushing on the stack rather than when popping. The body of the main loop (lines 5–10 in Fig. 2) now becomes:

```
while top > SA[i] do
    NSV_text[top] ← SA[i]
    top ← PSV_text[top]
PSV_text[SA[i]] ← top
top ← SA[i]
```

– KKP2 needs to be modified to compute $\mathsf{PSV}_{\mathsf{text}}$ values first instead of $\mathsf{NSV}_{\mathsf{text}}$ values. The $\mathsf{PSV}_{\mathsf{text}}$-first version is symmetric to the $\mathsf{NSV}_{\mathsf{text}}$-first algorithm. In particular, Φ_t is replaced by the inverse permutation Φ_t^{-1}. The algorithm is shown in Fig. 4.

Algorithm `KKP2n`

```
1:  top ← 0        // top of stack
2:  for i ← 1 to n do
3:      while top > SA[i] do
4:          top ← Φ^{-1}[top]    // pop from stack
5:      Φ^{-1}[SA[i]] ← top    // Φ^{-1}[SA[i]] = PSV_text[SA[i]]
6:      top ← SA[i]        // push to stack
7:  Φ^{-1}[0] ← 0
8:  next ← 1
9:  for t ← 1 to n do
10:     psv ← Φ^{-1}[t]
11:     nsv ← Φ^{-1}[psv]
12:     if t = next then
13:         next ← LZ-Factor(t, nsv, psv)
14:     Φ^{-1}[t] ← nsv
15:     Φ^{-1}[psv] ← t
```

Fig. 4. LZ factorization using $2n \log n$ bits of working space (the arrays SA and Φ^{-1}) without an explicit stack. The SA remains intact after the computation.

The versions without an explicit stack are slightly slower because of the non-locality of stack operations. A faster way to avoid overwriting SA would be to use a separate stack. However, the stack can grow as big as n (for example when $\mathsf{X} = a^{n-1}b$) which increases the worst case space requirement by $n \log n$ bits. We can get the best of both alternatives by adding a fixed size stack buffer to the stackless version. The buffer holds the top part of the stack to speed up stack operations. When the buffer gets full, the bottom half of its contents is discarded, and when the buffer gets empty, it is filled half way using the PSV pointers. This version is called KKP2b.

All the algorithm variants have linear time complexity.

Table 1. Files used in the experiments. They are from the standard (S) Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl/texts.html`) and from the repetitive (R) Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl/repcorpus.html`). We truncated all files to 150MiB. The repetitive corpus files are either multiple versions of similar data (R) or artificially generated (A). The value of n/z (the average length of a phrase in the LZ factorization) is included as a measure of repetitiveness.

Name	Abbr.	σ	n/z	Source	Description
proteins	pro	25	9.57	S	Swissprot database
english	eng	220	13.77	S	Gutenberg Project
dna	dna	16	14.65	S	Human genome
sources	src	228	17.67	S	Linux and GCC sources
coreutils	cor	236	110	R/R	GNU Coreutils sources
cere	cer	5	112	R/R	Baking yeast genomes
kernel	ker	160	214	R/R	Linux Kernel sources
einstein.en	ein	124	3634	R/R	Wikipedia articles
tm29	tm	2	2912K	R/A	Thue-Morse sequence
rs.13	rs	2	3024K	R/A	Run-Rich String sequence

6 Experimental Results

We implemented the algorithms described in this paper and compared their performance in practice to algorithms from [13] and [9]. The main experiment measured the time to compute the LZ factorization of the text. All algorithms take the text and the suffix array as an input hence we omit the time to compute SA. The data sets used in experiments are described in detail in Table 1. All algorithms use the optimized version of `LZ-Factor` (Fig. 1), which slightly reduces the time (e.g. for KKP3 by 2% on non-repetitive files). The implementations are available at `http://www.cs.helsinki.fi/group/pads/`.

Experiments Setup. We performed experiments on a 2.4GHz Intel Core i5 CPU equipped with 3072KiB L2 cache and 4GiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 10.04, 64bit) running kernel 2.6.32. All programs were compiled using `g++` version 4.4.3 with `-O3 -static -DNDEBUG` options. For each combination of algorithm and test file we report the median runtime from five executions.

Discussion. The LZ factorization times are shown in the top part of Table 2. In nearly all cases algorithms introduced in this paper outperform the algorithms from [9] (which are, to our knowledge, the fastest up-to-date linear time LZ factorization algorithms) while using the same or less space. In particular the KKP2 algorithms are always faster and simultaneously use at least $n \log n$ bits less space. A notably big difference is observed for non-repetitive data, where KKP3 significantly dominates all prior solutions.

Table 2. Time and space consumption for computing LZ factorization/LPF array. The timing values were obtained with the standard C `clock` function and are scaled to seconds per gigabyte. The times do not include any reading from or writing to disk. The second column summarizes the practical working space (excluding the output in case of LZ factorization) of each algorithm assuming byte alphabet and 32-bit integers. Note that LPF-online computes only the ℓ_i component of LPF array. If this is sufficient, KKP2-LPF can be modified (without affecting the speed) to use only $9n$ bytes.

	Alg.	Mem	pro	eng	dna	src	cor	cer	ker	ein	tm	rs
LZ factorization	KKP3	$13n$	74.5	75.7	81.7	50.5	43.6	63.2	45.7	56.9	38.2	77.8
	KKP2	$9n$	83.9	80.6	92.7	54.7	40.2	53.3	41.6	43.6	35.1	49.0
	KKP2b	$9n$	84.1	80.6	92.7	54.8	40.2	53.2	41.5	43.5	35.1	49.4
	KKP2n	$9n$	88.1	84.6	97.3	56.1	40.6	57.7	42.2	47.6	38.7	52.0
	ISA6r	$6n$	-	-	-	-	43.3	51.8	39.2	31.1	34.2	34.8
	ISA6s	$6n$	198.0	171.0	175.2	115.0	49.4	56.3	45.7	37.1	39.6	40.8
	ISA9	$9n$	92.7	83.9	86.1	59.3	41.9	53.0	42.8	45.2	36.4	51.8
	iBGS	$17n$	99.8	93.2	97.5	69.3	51.5	65.5	52.9	60.0	44.1	59.5
	iBGL	$17n$	123.2	108.6	113.4	77.8	52.2	66.1	53.0	58.6	44.2	59.5
	iBGT	$13n$	171.4	153.9	188.0	99.8	55.4	84.1	56.2	52.8	44.4	56.5
LPF	KKP3-LPF	$13n$	115.5	112.9	133.5	71.1	56.0	88.0	58.0	63.5	49.2	82.8
	KKP2-LPF	$13n$	140.3	132.4	167.2	83.6	54.6	82.6	55.6	51.3	41.1	58.0
	iOG	$13n$	210.1	188.0	243.7	121.3	66.8	104	66.4	60.6	50.3	62.7
	LPF-online	$13n$	160.4	162.3	187.2	114.2	103	137	109	127	100	148

The new algorithms (e.g. KKP2b) also dominate in most cases the general purpose practical algorithms from [13] (ISA9 and ISA6s), while offering stronger worst case time guarantees, but are a frame slower (and use about 50% more space in practice) than ISA6r for highly repetitive data.

The comparison of KKP2n to KKP2 reveals the expected slowdown (up to 16%) due to the non-local stack simulation. However, this effect is almost completely eliminated by buffering the top part of the stack (KKP2b). With a 256KiB buffer we obtained runtimes almost identical to KKP2 ($< 1\%$ difference in all cases). We observed a similar effect when applying this optimization to the KKP3 algorithm but, for brevity, we only present the results for KKP2.

Full LPF array. All our algorithms can be modified to compute the full LPF array, i.e. the set of longest previous factors (p_i, ℓ_i) for $i \in [1..n]$ in linear time. After obtaining $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ values, instead of repeatedly calling `LZ-Factor` to compute the LZ factorization, we compute all previous factors using the algorithm of Crochemore and Ilie [4, Fig. 2]. We compared this approach to the fastest algorithms for computing LPF array by Ohlebusch and Gog [16] (with the interleaving optimization from [9]) and LPF-online from [5] (see [13] and [16] for comparison). For LCP array computation we use the fastest version of Φ algorithm consuming $13n$ bytes of space [12].

As shown in Table 2, modified KKP2 algorithm consistently outperforms old methods. The LPF variant of KKP3 is even faster, when input is not repetitive.

Table 3. Times for computing LZ factorization, taking into account the disk reading time. The values are wallclock times scaled to seconds per gigabyte. KKP1s is a version of KKP2b that streams the suffix array from disk, and so requires only $n \log n$ bits of working space.

Alg.	pro	eng	dna	src	cor	cer	ker	ein	tm	rs
KKP1s	106.5	100.2	109.0	86.6	71.0	74.9	68.4	67.6	66.1	66.1
KKP2b	150.6	143.7	155.7	117.8	103.6	115.9	102.9	107.3	96.8	111.6

Streaming. As explained in Section 5 our new algorithms can be implemented so that SA is only accessed sequentially in a read-only manner, allowing it to be streamed from the disk. Furthermore, all algorithms (including full LPF variants) can stream the output, which is produced in order, directly to disk. The streaming versions of KKP2b and KKP2b-LPF, called KKP1s and KKP1s-LPF, use only $n \log n$ bits of working space in addition to the text and small stack and disk buffers. We have implemented KKP1s and compared its performance to KKP2b under the assumption that SA is stored on the disk and the disk reading time is included in the total runtime. Reading from the disk was performed with the standard C `fread` function, either as a single read (KKP2b) or using a 32KiB buffer (KKP1s).

Surprisingly, in such setting, KKP1s is significantly faster than KKP2b, as shown in Table 3. Further investigation revealed that the advantage of the streaming algorithm is apparently due to the implementation of I/O in the Linux operating system. The Linux kernel performs implicit asynchronous read ahead operations when a file is accessed sequentially, allowing an overlap of I/O and CPU computation (see [17]).

7 Future Work

We have reduced the working memory of linear time LZ factorization to $2n \log n$ bits, but one wonders if only $(1 + \epsilon)n \log n$ bits (for an arbitrary constant ϵ) is enough, as it is for suffix array construction [11]. In [13] working space of $(1+\epsilon)n \log n + n$ bits is achieved, but at the price of $O(n \log \sigma)$ runtime. We are also exploring even more space efficient (but slower) approaches [10].

Our streaming algorithms are a first step towards exploiting external memory in LZ factorization. We are currently exploring semi-external variants of these algorithms that keep little else than the input string in memory. This is achieved by permuting the NSV/PSV values from lex order to text order using external memory. Fully external memory as well as parallel and distributed approaches would also be of high interest, especially given the recent pattern matching indexes which use LZ77.

Finally, another problem is to find a scalable way to accurately estimate the size of the LZ factorization in lieu of actually computing it. Such a tool would be useful for entropy estimation, and to guide the selection of appropriate compressors and compressed indexes when managing massive data sets.

Acknowledgments. We thank Keisuke Goto and Hideo Bannai for an early copy of their paper [9].

References

1. Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylova, E., Smyth, W., Tischler, G., Yusufu, M.: A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. ACM Comput. Surv. 45(1), 5:1–5:17 (2012)
2. Charikar, M., Lehman, E., Liu, D., Panigrhy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
3. Chen, G., Puglisi, S.J., Smyth, W.F.: Fast and practical algorithms for computing all the runs in a string. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 307–315. Springer, Heidelberg (2007)
4. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Information Processing Letters 106(2), 75–80 (2008)
5. 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)
6. Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel-Ziv factorization. In: DCC 2008, pp. 482–488. IEEE Computer Society (2008)
7. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: A faster grammar-based self-index. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 240–251. Springer, Heidelberg (2012)
8. Gagie, T., Gawrychowski, P., Puglisi, S.J.: Faster approximate pattern matching in compressed repetitive texts. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 653–662. Springer, Heidelberg (2011)
9. Goto, K., Bannai, H.: Simpler and faster Lempel Ziv factorization. In: DCC 2013, pp. 133–142. IEEE Computer Society (2013)
10. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: Bonifaci, V. (ed.) SEA 2013. LNCS, vol. 7933, pp. 139–150. Springer, Heidelberg (2013)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)
12. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
13. Kempa, D., Puglisi, S.J.: Lempel-Ziv factorization: simple, fast, practical. In: Zeh, N., Sanders, P. (eds.) ALENEX 2013, pp. 103–112. SIAM (2013)
14. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 41–54. Springer, Heidelberg (2011)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
16. 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)
17. Wu, F.: Sequential file prefetching in Linux. In: Wiseman, Y., Jiang, S. (eds.) Advanced Operating Systems and Kernel Applications: Techniques and Technologies, ch. 11, pp. 217–236. IGI Global (2009)