Lightweight Lempel-Ziv Parsing[⋆]

Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi

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

Abstract. We introduce a new approach to LZ77 factorization that uses $O(n/d)$ words of working space and $O(dn)$ time for any $d \geq 1$ (for polylogarithmic alphabet sizes). We also describe carefully engineered implementations of alternative approaches to lightweight LZ77 factorization. Extensive experiments show that the new algorithm is superior, and particularly so at the lowest memory levels and for highly repetitive data. As a part of the algorithm, we describe new methods for computing matching statistics which may be of independent interest.

1 Introduction

The Lempel-Ziv factorization [28], also known as the LZ77 factorization, or LZ77 parsing, is a fundamental tool for compressing data and string processing, and has recently become the basis for several compressed full-text pattern matching indexes [17,11]. These indexes are designed to efficiently store and search massive, highly-repetitive data sets — such as web crawls, genome collections, and versioned code repositories — which are increasingly common [22].

In traditional compression settings (for example the popular `gzip` tool) LZ77 factorization is kept timely by factorizing relative to only a small, recent window of data, or by breaking the data up into blocks and factorizing each block separately. This approach fails to capture widely spaced repetitions in the input, and anyway, in many applications, including construction of the above mentioned LZ77-based text indexes, whole-string LZ77 factorizations are required.

The fastest LZ77 algorithms (see [12] for the latest comparison) use a lot of space, at least $6n$ bytes for an input of n symbols and often more. This prevents them from scaling to really large inputs. Space-efficient algorithms are desirable even on smaller inputs, as they place less burden on the underlying system.

One approach to more space efficient LZ factorization is to use compressed suffix arrays and succinct data structures [21]. Two proposals in this direction are due to Kreft and Navarro [16] and Ohlebusch and Gog [23]. In this paper, we describe carefully engineered implementations of these algorithms. We also propose a new, space-efficient variant of the recent ISA family of algorithms [15].

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

Compressed indexes are usually built from the uncompressed suffix array (SA) which requires $4n$ bytes. Our implementations are instead based on the Burrows-Wheeler transform (BWT), constructed directly in about 2–2.5n bytes using the algorithm of Okanohara and Sadakane [26]. There also exists two online algorithms based on compressed indexes [25,27] but they are not competitive in practice in the offline context.

The main contribution of this paper is a new algorithm to compute the LZ77 factorization without ever constructing SA or BWT for the whole input. At a high-level, the algorithm divides the input up into blocks, and processes each block in turn, by first computing a pattern matching index for the block, then scanning the prefix of the input prior to the block through the index to compute longest-matches, which are then massaged into LZ77 factors. For a string of length n and σ distinct symbols, the algorithm uses $n \log \sigma + \mathrm{O}(n \log n/d)$ bits of space, and $\mathrm{O}(dnt_{\mathsf{rank}})$ time, where d is the number of blocks, and t_{rank} is the time complexity of the rank operation over sequences with alphabet size σ (see e.g. [2]). The $n \log \sigma$ bits in the space bound is for the input string itself which is treated as read-only.

Our implementation of the new algorithm does not, for the most part, use compressed or succinct data structures. The goal is to optimize speed rather than space in the data structures, because we can use the parameter d to control the tradeoff. Our experiments demonstrate that this approach is superior to algorithms using compressed indexes.

As a part of the new algorithm, we describe new techniques for computing matching statistics [5] that may be of independent interest. In particular, we show how to invert matching statistics, i.e., to compute the matching statistics of a string B w.r.t. a string A from the matching statistics of A w.r.t. B, which saves a lot of space when A is much longer than B.

All our implementations operate in main memory only and thus need at least n bytes just to hold the input. Reducing the memory consumption further requires some use of external memory, a direction largely unexplored in the literature so far. We speculate that the scanning, block-oriented nature of the new algorithm will allow efficient secondary memory implementations, but that study is left for the future.

2 Basic Notation and Algorithmic Machinery

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 the alphabet $[0..\sigma-1]$. We assume $\mathsf{X}[n]$ is a special "end of string" symbol, \$, smaller than all other symbols in the alphabet. The reverse of X is denoted $\hat{\mathsf{X}}$. 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. $\mathsf{X}[i..j]$ is 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. By $\mathsf{X}[i..j)$ we denote $\mathsf{X}[i..j-1]$. If $j < i$ we define $\mathsf{X}[i..j]$ to be the empty string, also denoted by ε.

Suffix Arrays. The suffix array [19] SA_X (we drop subscripts when they are clear from the context) of a string X is an array $\mathsf{SA}[1..n]$ which contains 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 $\mathsf{SA}[j] = i$.

Let $\mathsf{lcp}(i, j)$ denote the length of the longest-common-prefix of suffix i and suffix j. For example, in the string $\mathsf{X} = zzzzzapzap$, $\mathsf{lcp}(1, 4) = 2 = |zz|$, and $\mathsf{lcp}(5, 8) = 3 = |zap|$. The longest-common-prefix (LCP) array [14,13], $\mathsf{LCP}_\mathsf{X} = \mathsf{LCP}[1..n]$, is defined such that $\mathsf{LCP}[1] = 0$, and $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i-1])$ for $i \in [2..n]$.

For a string Y, the Y-interval in the suffix array SA_X is the interval $\mathsf{SA}[s..e]$ that contains all suffixes having Y as a prefix. The Y-interval is a representation of the occurrences of Y in X. For a character c and a string Y, the computation of the $c\mathsf{Y}$-interval from the Y-interval is called a *left extension* and the computation of the Y-interval from the $\mathsf{Y}c$-interval is called a *right contraction. Left contraction* and *right extension* are defined symmetrically.

BWT and backward search. The Burrows-Wheeler Transform [3] $\mathsf{BWT}[1..n]$ is a permutation of X such that $\mathsf{BWT}[i] = \mathsf{X}[\mathsf{SA}[i]-1]$ if $\mathsf{SA}[i] > 1$ and \$ otherwise. We also define $\mathsf{LF}[i] = j$ iff $\mathsf{SA}[j] = \mathsf{SA}[i] - 1$, except when $\mathsf{SA}[i] = 1$, in which case $\mathsf{LF}[i] = \mathsf{ISA}[n]$. Let $\mathsf{C}[c]$, for symbol c, be the number of symbols in X lexicographically smaller than c. The function $\mathsf{rank}(\mathsf{X}, c, i)$, for string X, symbol c, and integer i, returns the number of occurrences of c in $\mathsf{X}[1..i]$. It is well known that $\mathsf{LF}[i] = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}(\mathsf{BWT}, \mathsf{BWT}[i], i)$. Furthermore, we can compute the left extension using C and rank. If $\mathsf{SA}[s..e]$ is the Y-interval, then $\mathsf{SA}[\mathsf{C}[c]+\mathsf{rank}(\mathsf{BWT}, c, s), \mathsf{C}[c]+\mathsf{rank}(\mathsf{BWT}, c, e)]$ is the $c\mathsf{Y}$-interval. This is called *backward search* [8].

NSV/PSV and RMQ. For an array A, the *next and previous smaller value* (NSV/PSV) operations are defined as $\mathsf{NSV}(i) = \min\{j \in [i+1..n] \mid \mathsf{A}[j] < \mathsf{A}[i]\}$ and $\mathsf{PSV}(i) = \max\{j \in [1..i-1] \mid \mathsf{A}[j] < \mathsf{A}[i]\}$. A related operation on A is *range minimum query*: $\mathsf{RMQ}(\mathsf{A}, i, j)$ is $k \in [i..j]$ such that $\mathsf{A}[k]$ is the minimum value in $\mathsf{A}[i..j]$. Both NSV/PSV operations and RMQ operations over the LCP array can be used for implementing right contraction (see Section 4).

LZ77. Before defining the LZ77 factorization, we introduce the concept of a *longest previous factor* (LPF). The LPF at position i in string X is a pair $\mathsf{LPF}_\mathsf{X}[i] = (p_i, \ell_i)$ such that, $p_i < i$, $\mathsf{X}[p_i..p_i+\ell_i) = \mathsf{X}[i..i+\ell_i)$, and ℓ_i is maximized. In other words, $\mathsf{X}[i..i+\ell_i)$ is the longest prefix of $\mathsf{X}[i..n]$ which also occurs at some position $p_i < i$ in X. Note also that there may be more than one potential source (that is, p_i value), 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 $i + \ell_i$. The exception is the case $\ell_i = 0$, which happens iff $\mathsf{X}[i]$ is the leftmost occurrence of a symbol in X. In this case we output $(\mathsf{X}[i], 0)$ (to represent $\mathsf{X}[i..i]$) and the next phrase starts at position $i + 1$. When $\ell_i > 0$, the substring $\mathsf{X}[p_i..p_i + \ell_i)$ is called the *source* of phrase $\mathsf{X}[i..i + \ell_i)$. We denote the number of phrases in the LZ77 parsing of X by z.

Matching Statistics. Given two strings Y and Z, the matching statistics of Y w.r.t. Z, denoted $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$, is an array of $|\mathsf{Y}|$ pairs, (p_1, ℓ_1), (p_2, ℓ_2), ..., $(p_{|\mathsf{Y}|}, \ell_{|\mathsf{Y}|})$, such that for all $i \in [1..|\mathsf{Y}|]$, $\mathsf{Y}[i..i + \ell_i) = \mathsf{Z}[p_i..p_i + \ell_i)$ is the longest substring starting at position i in Y that is also a substring of Z. The observant reader will note the resemblance to the LPF array. Indeed, if we replace LPF_Y with $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$ in the computation of the LZ factorization of Y, the result is the relative LZ factorization of Y w.r.t. Z [18].

3 Lightweight, Scan-Based LZ77 Parsing

In this section we present a new algorithm for LZ77 factorization called LZscan.

Basic Algorithm. Conceptually LZscan divides X up into $d = \lceil n/b \rceil$ fixed size blocks of length b: $\mathsf{X}[1..b]$, $\mathsf{X}[b + 1..2b]$, ... The last block could be smaller than b, but this does not change the operation of the algorithm. In the description that follows we will refer to the block currently under consideration as B, and to the prefix of X that ends just before B as A. Thus, if $\mathsf{B} = \mathsf{X}[kb + 1..(k + 1)b]$, then $\mathsf{A} = \mathsf{X}[1..kb]$.

To begin, we will assume no LZ factor or its source crosses a boundary of the block B. Later we will show how to remove these assumptions.

The outline of the algorithm for processing a block B is shown below.

1. Compute $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$
2. Compute $\mathsf{MS}_{\mathsf{B}|\mathsf{A}}$ from $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$, SA_B and LCP_B
3. Compute $\mathsf{LPF}_{\mathsf{AB}}[kb + 1..(k + 1)b]$ from $\mathsf{MS}_{\mathsf{B}|\mathsf{A}}$ and LPF_B
4. Factorize B using $\mathsf{LPF}_{\mathsf{AB}}[kb + 1..(k + 1)b]$

Step 1 is the computational bottleneck of the algorithm in theory and practice. Theoretically, the time complexity of Step 1 is $\mathrm{O}((|A|+|B|)t_{\mathsf{rank}})$, where t_{rank} is the time complexity of the rank operation on BWT_B (see, e.g., [2]). Thus the total time complexity of LZscan is $\mathrm{O}(dnt_{\mathsf{rank}})$ using $\mathrm{O}(b)$ words of space in addition to input and output. The practical implementation of Step 1 is described in Section 4. In the rest of this section, we describe the details of the other steps.

Step 2: Inverting Matching Statistics. We want to compute $\mathsf{MS}_{\mathsf{B}|\mathsf{A}}$ but we cannot afford the space of the large data structures on A required by standard methods [1,23]. Instead, we compute first $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$ involving large data structures on B, which we can afford, and only a scan of A (see Section 4 for details). We then *invert* $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$ to obtain $\mathsf{MS}_{\mathsf{B}|\mathsf{A}}$. The inversion algorithm is given in Fig. 1.

```
Algorithm MS-Invert
1:  for $i \leftarrow 1$ to $|\mathsf{B}|$ do $\mathsf{MS}_{\mathsf{B|A}}[i] \leftarrow (0,0)$
2:  for $i \leftarrow 1$ to $|A|$ do            // transfer information from $\mathsf{MS}_{\mathsf{A|B}}$ to $\mathsf{MS}_{\mathsf{B|A}}$
3:      $(p_\mathsf{A}, \ell_\mathsf{A}) \leftarrow \mathsf{MS}_{\mathsf{A|B}}[i]$
4:      $(p_\mathsf{B}, \ell_\mathsf{B}) \leftarrow \mathsf{MS}_{\mathsf{B|A}}[p_\mathsf{A}]$
5:      if $\ell_\mathsf{A} > \ell_\mathsf{B}$ then $\mathsf{MS}_{\mathsf{B|A}}[p_\mathsf{A}] \leftarrow (i, \ell_\mathsf{A})$
6:  $(p,\ell) \leftarrow \mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[1]]$            // spread information in $\mathsf{MS}_{\mathsf{B|A}}$
7:  for $i \leftarrow 2$ to $|\mathsf{B}|$ do            // in lexicograhically ascending direction
8:      $\ell \leftarrow \min(\ell, \mathsf{LCP}_\mathsf{B}[i])$
9:      $(p_\mathsf{B}, \ell_\mathsf{B}) \leftarrow \mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[i]]$
10:     if $\ell > \ell_\mathsf{B}$ then $\mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[i]] \leftarrow (p,\ell)$
11:     else $(p,\ell) \leftarrow (p_\mathsf{B}, \ell_\mathsf{B})$
12: $(p,\ell) \leftarrow \mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[|\mathsf{B}|]]$            // spread information in $\mathsf{MS}_{\mathsf{B|A}}$
13: for $i \leftarrow |B|-1$ downto 1 do        // in lexicograhically descending direction
14:     $\ell \leftarrow \min(\ell, \mathsf{LCP}_\mathsf{B}[i+1])$
15:     $(p_\mathsf{B}, \ell_\mathsf{B}) \leftarrow \mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[i]]$
16:     if $\ell > \ell_\mathsf{B}$ then $\mathsf{MS}_{\mathsf{B|A}}[\mathsf{SA}_\mathsf{B}[i]] \leftarrow (p,\ell)$
17:     else $(p,\ell) \leftarrow (p_\mathsf{B}, \ell_\mathsf{B})$
```

Fig. 1. Inverting matching statistics

Note that the algorithm accesses each entry of $\mathsf{MS}_{\mathsf{A|B}}$ only once and the order of these accesses does not matter. Thus we can execute the code on lines 3–5 immediately after computing $\mathsf{MS}_{\mathsf{A|B}}[i]$ in Step 1 and then discard that value. This way we can avoid storing $\mathsf{MS}_{\mathsf{A|B}}$.

Step3: Computing LPF. Consider the pair $(p,\ell) = \mathsf{LPF}_{\mathsf{AB}}[i]$ for $i \in [kb+1..(k+1)b]$ that we want to compute and assume $\ell > 0$ (otherwise i is the position of the leftmost occurrence of $\mathsf{X}[i]$ in X, which we can easily detect). Clearly, either $p \le kb$ and $\mathsf{LPF}_{\mathsf{AB}}[i] = \mathsf{MS}_{\mathsf{B|A}}[i]$, or $kb < p < i$ and $\mathsf{LPF}_{\mathsf{AB}}[i] = (kb + p_\mathsf{B}, \ell_\mathsf{B})$, where $(p_\mathsf{B}, \ell_\mathsf{B}) = \mathsf{LPF}_\mathsf{B}[i - kb]$. Thus computing $\mathsf{LPF}_{\mathsf{AB}}$ from $\mathsf{MS}_{\mathsf{B|A}}[i]$ and LPF_B is easy.

The above is true if the sources do not cross the block boundary, but the case where $p \le kb$ but $p + \ell > kb + 1$ is not handled correctly. An easy correction is to replace $\mathsf{MS}_{\mathsf{A|B}}$ with $\mathsf{MS}_{\mathsf{AB|B}}[1..kb]$ in all of the steps.

Step 4: Parsing. We use the standard LZ77 parsing to factorize B except LPF_B is replaced with $\mathsf{LPF}_{\mathsf{AB}}[kb+1..(k+1)b]$.

So far we have assumed that every block starts with a new phrase, or, put another way, that a phrase ends at the end of every block. Let $\mathsf{X}[i..(k+1)b]$ be the last factor in B, after we have factorized B as described above. This may not be a true LZ factor when considering the whole X but may continue beyond the end of B. To find the true end point, we treat $\mathsf{X}[i..n]$ as a pattern, and apply the constant extra space pattern matching algorithm of Crochemore [7], looking for the longest prefix of $\mathsf{X}[i..n]$ starting in $\mathsf{X}[1..i-1]$. We must modify the algorithm of Crochemore so that it finds the longest matching prefix of the pattern rather than a full match, but this is possible without increasing its time or space complexity.

4 Computation of Matching Statistics

In this section, we describe how to compute the matching statistics $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$. As mentioned in Section 3, what we actually want is $\mathsf{MS}_{\mathsf{AB}|\mathsf{B}}[1..kb]$. However, the only difference is that the starting point of the computation is the B-interval in SA_B instead of the ε-interval.

Similarly to most algorithms for computing the matching statistics, we first construct some data structures on B and then scan A. During the whole LZ factorization, most of the time is spend on the scanning and the time for constructing the data structures is insignificant in practice. Thus we omit the construction details here. The space requirement of the data structures is more important but not critical as we can compensate for increased space by reducing the block size b. Using more space (per character of B) is worth doing if it increases scanning speed more than it increases space. Consequently, we mostly use plain, uncompressed arrays.

Standard approach. The standard approach of computing the matching statistics using the suffix array is to compute for each position i the longest prefix $\mathsf{P}_i = \mathsf{A}[i..i+\ell_i)$ of the suffix $\mathsf{A}[i..|A|]$ such that the P_i-interval in SA_B is non-empty. Then $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}[i] = (p_i, \ell_i)$, where p_i is any suffix in the P_i-interval. This can be done either with a forward scan of A, computing each P_i-interval from P_{i-1}-interval using the extend right and contract left operations [1], or with a backward scan computing each P_i-interval from P_{i+1}-interval using the extend left and contract right operations [24]. We use the latter alternative but with bigger and faster data structures.

The extend left operation is implemented by backward search. We need the array C of size σ and an implementation of the rank function on BWT. For the latter, we use the fast rank data structure of Ferragina et al. [9], which uses $4b$ bytes.

The contract right operation is implemented using the NSV and PSV operations on LCP_B similarly to Ohlebusch and Gog [24], but instead of a compressed representation, we store the NSV/PSV values as plain arrays. As a nod towards reducing space, we store the NSV/PSV values as offsets using 2 bytes each. If the offset is too large (which is very rare), we obtain the value using the NSV/PSV data structure of Cánovas and Navarro [4], which needs less than $0.1b$ bytes. Here the space saving was worth it as it had essentially no effect on speed.

The peak memory use of the resulting algorithm is $n + (24.1)b + \mathrm{O}(\sigma)$ bytes.

New approach. Our second approach is similar to the first, but instead of maintaining both end points of the P_i-interval, we keep just one, arbitrary position s_i within the interval. In principle, we perform left extension by backward search, i.e., $s_i = \mathsf{C}[\mathsf{X}[i]] + \mathsf{rank}(\mathsf{BWT}, \mathsf{X}[i], s_{i+1})$. However, checking whether the resulting interval is empty and performing right contractions if it is, is more involved. To compute s_i and ℓ_i from s_{i+1} and ℓ_{i+1}, we execute the following steps:

1. Let $c = \mathsf{X}[i]$. If $\mathsf{BWT}[s_{i+1}] = c$, set $s_i = \mathsf{C}[c] + \mathsf{rank}(\mathsf{BWT}, c, s_{i+1})$ and $\ell_i = \ell_{i+1} + 1$.
2. Otherwise, let $\mathsf{BWT}[u]$ be the nearest occurrence of c in BWT before the position s_{i+1}. Compute the rank of that occurrence $r = \mathsf{rank}(\mathsf{BWT}, c, u)$ and $\ell_u = \mathsf{LCP}[\mathsf{RMQ}(\mathsf{LCP}, u + 1, s_{i+1})]$. If $\ell_u \geq \ell_{i+1}$, set $s_i = \mathsf{C}[c] + r$ and $\ell_i = \ell_{i+1} + 1$.
3. Otherwise, let $\mathsf{BWT}[v]$ be the nearest occurrence of c in BWT after the position s_{i+1} and compute $\ell_v = \mathsf{LCP}[\mathsf{RMQ}(\mathsf{LCP}, s_{i+1} + 1, v)]$. If $\ell_v \leq \ell_u$, set $s_i = \mathsf{C}[c] + r$ and $\ell_i = \ell_u + 1$.
4. Otherwise, set $s_i = \mathsf{C}[c] + r + 1$ and $\ell_i = \min(\ell_{i+1}, \ell_v) + 1$.

The implementation of the above algorithm is based on the arrays BWT, LCP and $\mathsf{R}[1..b]$, where $\mathsf{R}[i] = \mathsf{rank}(\mathsf{BWT}, \mathsf{BWT}[i], i)$. All the above operations can be performed by scanning BWT and LCP starting from the position s_{i+1} and accessing one value in R. To avoid long scans, we divide BWT and LCP into blocks of size 2σ, and store for each block and each symbol c that occurs in B, the values r, ℓ_u and ℓ_v that would get computed if scans starting inside the block continued beyond the block boundaries.

The peak memory use is $n + 27b + \mathrm{O}(\sigma)$ bytes. This is more than in the first approach, but this is more than compensated by increased scanning speed.

Skipping repetitions. During the preceding stages of the LZ factorization, we have built up knowledge of repetition present in A, which can be exploited to skip (sometimes large) parts of A during the matching-statistics scan. Consider an LZ factor $\mathsf{A}[i..i+\ell)$. Because, by definition, $\mathsf{A}[i..i+\ell)$ occurs earlier in A too, any source of an LZ factor of B that is completely inside $\mathsf{A}[i..i+\ell)$ could be replaced with an equivalent source in that earlier occurrence. Thus such factors can be skipped during the computation of $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}$ without an effect on the factorization.

More precisely, if during the scan we compute $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}[j] = (p, k)$ and find that $i \leq j < j+k \leq i+\ell$ for an LZ factor $\mathsf{A}[i..i+\ell)$, we will compute $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}[i-1]$ and continue the scanning from $i - 1$. However, we will do this only for long phrases with $\ell \geq 40$. To compute $\mathsf{MS}_{\mathsf{A}|\mathsf{B}}[i-1]$ from scratch, we use right extension operations implemented by a binary search on SA.

To implement this "skipping trick" we use a bitvector of n bits to mark LZ77 phrase boundaries adding $0.125n$ bytes to the peak memory.

5 Algorithms Based on Compressed Indexes

We went to some effort to ensure the baseline system used to evaluate LZscan in our experiments was not a "straw man". This required careful study and improvement of some existing approaches, which we now describe.

FM-Index. The main data structure in all the algorithms below is an implementation of the FM-index (FMI) [8]. It consists of two main components:

- BWT_X *with support for the rank operation.* This enables backward search and the LF operation as described in Section 2. We have tried several rank data structures and found the one by Navarro [20, Sect. 7.1] to be the best in practice.
- *A sampling of* SA_X. This together with the LF operation enables arbitrary SA access since $\mathsf{SA}[i] = \mathsf{SA}[\mathsf{LF}^k[i]] + k$ for any $k < \mathsf{SA}[i]$. The sampling rate is a major space–time tradeoff parameter.

In many implementations of FMI, the construction starts with computing the uncompressed suffix array but we cannot afford the space. Instead, we construct BWT directly using the algorithm of Okanohara and Sadakane [26]. The method uses roughly 2–2.5n bytes of space but destroys the text, which is required later during LZ parsing. Thus, once we have BWT, we build a rank structure over it and use it to invert the BWT. During the inversion process we recover and store the text and gather the SA sample values.

CPS2 simulation. The CPS2 algorithm [6] is an LZ parsing algorithm based on SA_X. To compute the LZ factor starting at i, it computes the $\mathsf{X}[i..i+\ell)$-interval for $\ell = 1, 2, 3, \ldots$ as long as the $\mathsf{X}[i..i+\ell)$-interval contains a value $p < i$, indicating an occurrence of $\mathsf{X}[i..i+\ell)$ starting at p.

The key operations in CPS2 are right extension and checking whether an SA interval contains a value smaller than i. Kreft and Navarro [16] as well as Ohlebusch and Gog [23] are using FMI for $\hat{\mathsf{X}}$, the reverse of X, which allows simulating right extension on SA_X by left extension on $\mathsf{SA}_{\hat{\mathsf{X}}}$. The two algorithms differ in the way they implement the interval checks:

- Kreft and Navarro use the RMQ operation. They use the RMQ data structure by Fischer and Heun [10] but we use the one by Cánovas and Navarro [4]. The latter is easy and fast to construct during BWT inversion but queries are slow without an explicit SA. We speed up queries by replacing a general RMQ with the check whether the interval contains a value smaller than i. This implementation is called $\mathsf{LZ\text{-}FMI\text{-}RMQ}$.
- Ohlebusch and Gog use NSV/PSV queries. The position s of i in SA must be in the $\mathsf{X}[i..i+\ell)$-interval. Thus we just need to check whether either $\mathsf{NSV}(s)$ or $\mathsf{PSV}(s)$ is in the interval too. They as well as we implement NSV/PSV using a balanced parentheses representation (BPR). This representation is initialized by accessing the values of SA left-to-right, which makes the construction slow using FMI. However, NSV/PSV queries with this data structure are fast, as they do not require accessing SA. This implementation is called $\mathsf{LZ\text{-}FMI\text{-}BPR}$.

ISA variant. Among the most space efficient prior LZ factorization algorithms are those of the ISA family [15] that use a sampled ISA, a full SA and a rank/LF implementation that relies on the presence of the full SA. We reduce the space further by replacing SA and the rank/LF data structure with the FM-index described above to obtain an algorithm called $\mathsf{LZ\text{-}FMI\text{-}ISA}$.

Table 1. Data set used in the experiments. The files are from the Pizza & Chili standard corpus[1] (S) and the Pizza & Chili repetitive corpus[2] (R). The value of n/z (average length of an LZ77 phrase) is included as a measure of repetitiveness. We use 100MiB prefixes of original files in order to reduce the time required to run the experiments with several algorithms and a large number of parameter combinations.

Name	σ	n/z	$n/2^{20}$	Source	Description
dna	16	14.2	100	S	Human genome
english	215	14.1	100	S	Gutenberg Project
sources	227	16.8	100	S	Linux and GCC sources
cere	5	84	100	R	yeast genome
einstein	121	2947	100	R	Wikipedia articles
kernel	160	156	100	R	Linux Kernel sources

6 Experiments

We performed experiments with the files listed in Table 1. All tests were conducted on a 2.53GHz Intel Xeon Duo CPU with 32GiB main memory and 8192K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Linux (Ubuntu 10.04) running kernel 3.0.0-26. The compiler was g++ (gcc version 4.4.3) executed with the `-O3 -static -DNDEBUG` options. Times were recorded with the C `clock` function. All algorithms operate strictly in-memory. The implementations are available at `http://www.cs.helsinki.fi/group/pads/`.

LZscan vs. other algorithms. We compared the LZscan implementation using our new approach for matching statistics boosted with the "skipping trick" (Section 4) to algorithms based on compressed indexes (Section 5). The experiments measured the LZ factorization time and the memory usage with varying parameter settings for each algorithm. The results are shown in Fig. 2. In all cases LZscan outperforms other algorithm across the whole tradeoff spectrum. Moreover, it can operate with very small memory (close to n bytes) unlike other algorithms, which all require at least $2n$ bytes to compute BWT. It achieves a superior performance for highly repetitive data even at very low memory levels.

Variants of LZscan. We made a separate comparison of LZscan with the different variants of the matching statistics computation (see Section 4). As can be seen from Fig. 3, our new algorithm for matching statistics computation is a significant improvement over the standard approach. Adding the skipping trick usually improves further (english) but can also slightly deteriorate the speed (dna). On the other hand, for highly repetitive data, the skipping trick alone gives a dramatic time reduction (einstein).

[1] `http://pizzachili.dcc.uchile.cl/texts.html`
[2] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

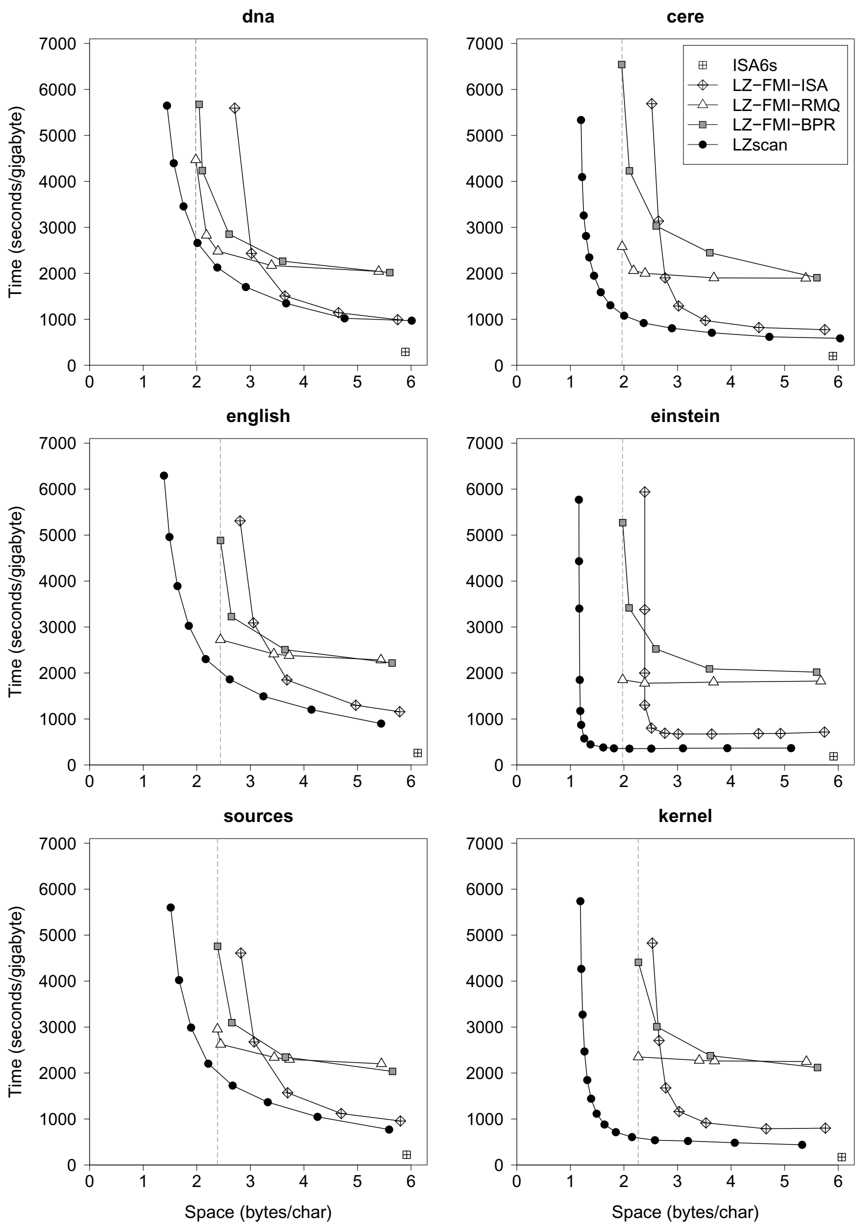

Fig. 2. Time-space tradeoffs for various LZ77 factorization algorithms. The times do not include reading from or writing to disk. For algorithms with multiple parameters controlling time/space we show only the optimal points, that is, points forming the lower convex hull of the points "cloud". The vertical line is the peak memory usage of the BWT construction algorithm [26], which is a space lower bound for all algorithms except LZscan. For comparison, we show the runtimes of ISA6s [15], currently the fastest LZ77 factorization algorithm using $6n$ bytes.

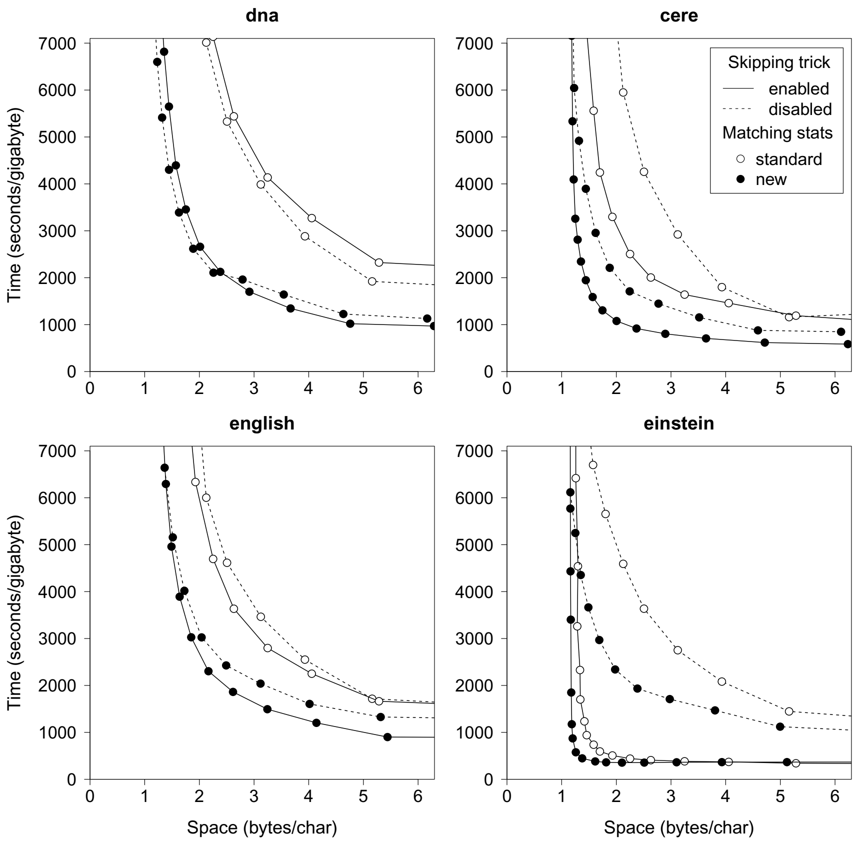

Fig. 3. Time-space tradeoffs for variants of LZscan (see Section 4)

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. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 315–326. Springer, Heidelberg (2010)
3. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California (1994)
4. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 94–105. Springer, Heidelberg (2010)
5. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica 12(4-5), 327–344 (1994)
6. Chen, G., Puglisi, S.J., Smyth, W.F.: Lempel-Ziv factorization using less time and space. Mathematics in Computer Science 1(4), 605–623 (2008)
7. Crochemore, M.: String-matching on ordered alphabets. Theoretical Computer Science 92, 33–47 (1992)
8. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)

9. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)
10. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
11. 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)
12. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel–Ziv factorization: Simple, fast, small. In: CPM 2013. LNCS. Springer (to appear, 2013), `http://arxiv.org/abs/1212.2952`
13. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
14. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
15. Kempa, D., Puglisi, S.J.: Lempel-Ziv factorization: simple, fast, practical. In: Zeh, N., Sanders, P. (eds.) ALENEX 2013, pp. 103–112. SIAM (2013)
16. Kreft, S., Navarro, G.: LZ77-like compression with fast random access. In: Storer, J.A., Marcellin, M.W. (eds.) DCC, pp. 239–248. IEEE Computer Society (2010)
17. 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)
18. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
19. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
20. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms 2(1), 87–114 (2004)
21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
22. Navarro, G.: Indexing highly repetitive collections. In: Arumugam, S., Smyth, B. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
23. 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)
24. Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 347–358. Springer, Heidelberg (2010)
25. Okanohara, D., Sadakane, K.: An online algorithm for finding the longest previous factors. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 696–707. Springer, Heidelberg (2008)
26. Okanohara, D., Sadakane, K.: A linear-time Burrows-Wheeler transform using induced sorting. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 90–101. Springer, Heidelberg (2009)
27. Starikovskaya, T.: Computing Lempel-Ziv factorization online. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 789–799. Springer, Heidelberg (2012)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)