Lazy Lempel-Ziv Factorization Algorithms

JUHA KÄRKKÄINEN, DOMINIK KEMPA, and SIMON J. PUGLISI, Helsinki Institute of Information Technology, Department of Computer Science, University of Helsinki

For decades the Lempel-Ziv (LZ77) factorization has been a cornerstone of data compression and string processing algorithms, and uses for it are still being uncovered. For example, LZ77 is central to several recent text indexing data structures designed to search highly repetitive collections. However, in many applications computation of the factorization remains a bottleneck in practice. In this article, we describe a number of simple and fast LZ77 factorization algorithms, which consistently outperform all previous methods in practice, use less memory, and still offer strong worst-case performance guarantees. A common feature of the new algorithms is that they compute longest common prefix information in a lazy fashion, with the degree of laziness in preprocessing characterizing different algorithms.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Lempel-Ziv factorization, Lempel-Ziv parsing, LZ77, suffix array, Burrows-Wheeler transform, data compression, string processing

ACM Reference Format:
Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. 2016. Lazy Lempel-Ziv factorization algorithms. J. Exp. Algorithmics 21, 2, Article 2.4 (October 2016), 19 pages.
DOI: http://dx.doi.org/10.1145/2699876

1. INTRODUCTION

For more than three decades, the Lempel-Ziv (LZ77) factorization [Ziv and Lempel 1977] has been a fundamental tool for compressing data. While many aspects of LZ77 have been heavily studied in that time, efficient computation of the factorization remains a bottleneck in many applications.

A recent focus in the field of compressed full-text indexing [Navarro 2012; Navarro and Mäkinen 2007] has been on *indexing highly repetitive collections*. Several types of large, modern data contain high amounts of duplication of relatively long substrings, which indexes based on LZ77 exploit particularly well [Kreft and Navarro 2013; Gagie et al. 2011, 2012; Ferrada et al. 2014]. Such data includes the new and rapidly growing genomic collections produced by high-throughput sequencing technology [1000 Genomes Project Consortium 2015; Mäkinen et al. 2010]; versioned

Partial preliminary versions of this paper appeared in the *Proceedings of the 2013 Meeting on Algorithm Engineering and Experiments (ALENEX)* and the *Proceedings of the 2013 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*.
This work is supported by the Academy of Finland, under grants 118653 (ALGODAN), 250345 (CoECGR), and 258308.
Authors' addresses: J. Kärkkäinen, D. Kempa, and S. J. Puglisi, Department of Computer Science, P.O. Box 68 (Gustaf Hällströmin katu 2b), FI-00014 University of Helsinki, Finland; emails: {juha.karkkainen, dominik.kempa, simon.puglisi}@cs.helsinki.fi.

DOI: http://dx.doi.org/10.1145/2699876

collections of source code and multiauthor documents, such as Wikipedia [Sirén et al. 2008]; and web crawls [Ferragina and Manzini 2010]. Efficient index construction is stated as an open problem by both Kreft and Navarro [2013] and Gagie et al. [2011] and, by our measurements, LZ77 computation accounts for more than 90% of index construction time.

In a more traditional setting, *compression of files using the 7zip tool* [Pavlov 2012], which is based on LZ77, has grown popular recently and is now bundled with most Linux distributions. 7zip is also effective for storing collections of files that later require fast random access, as is the case in information retrieval systems [Ferragina and Manzini 2010; Hoobin et al. 2011]. 7zip is capable of superior compression to gzip (which is also LZ77-based) on large files because it factorizes large blocks. However, our own measurements (see also those by Kreft and Navarro [2010]) indicate that 7zip has high memory overheads during factorization, with a memory peak of around $11n$ bytes, for a block of n bytes. More efficient factorization algorithms that allow bigger blocks to be processed and in less time, are thus of immediate practical benefit to systems and users. 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 [Charikar et al. 2005].

Aside from compression and indexing, LZ77 factorization is in heavy use as an algorithmic tool for string processing, in particular for *efficient detection of periodicities in strings* [Badkobeh et al. 2012; Duval et al. 2004; Gusfield and Stoye 2004; Kociumaka et al. 2012; Kolpakov et al. 2003; Kolpakov and Kucherov 2003]. Periodicities in turn have diverse applications throughout computer science, in the fields of bioinformatics, data mining, and extremal combinatorics.

Lazy LZ Factorization. The operation of an LZ factorization algorithm can be divided into two phases: a preprocessing phase that prepares some data structures and a parsing phase that performs a greedy parsing with the help of the data structures. We call an algorithm *eager* if it performs most of its work in the preprocessing phase and *lazy* if it leaves a lot of the computation to the parsing phase. The advantage of laziness is that typically most of the work in the parsing phase depends linearly on the number of phrases, denoted by z, rather than the length of input, denoted by n. Even in the worst case $z = \mathrm{O}(n/\log_\sigma n)$, where σ is the size of the alphabet, and for highly repetitive inputs, z can be much smaller.

1.1. Related Work

There exists a variety of worst-case linear-time algorithms to compute the LZ factorization (see Al-Hafeedh et al. [2012] for a survey up to 2010 and Ohlebusch and Gog [2011] and Goto and Bannai [2013] for later algorithms). Most of these belong to the eager camp, but the recent ones by Goto and Bannai [2013] exhibit a degree of laziness. These lazier algorithms are faster in practice than any previous algorithms for all kinds of inputs.

All these linear-time algorithms require at least $3n \log n$ bits of working space[1] in the worst case. The most space efficient linear-time algorithm is due to Chen et al. [2008]. 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 worst case $s = \Theta(n)$. Another space-efficient solution requiring $(2n + \sqrt{n}) \log n$ bits of space in the worst case is from Crochemore et al. [2008] but it computes only the lengths of the LZ77 phrases. It can be extended to compute the full parsing at the cost of extra $n \log n$ bits.

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

All of the linear-time 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) [Kärkkäinen et al. 2006]. This raises the question of whether the space complexity of linear-time LZ77 factorization can be reduced from $3n \log n$ bits. In this article, we answer the question in the affirmative by describing a linear-time algorithm using $2n \log n$ bits.

Further reduction of the space usage is possible if we relax the requirement of linear-time complexity. Some algorithms avoid the space requirement of the suffix array by using compressed text indexes [Okanohara and Sadakane 2008; Kreft and Navarro 2010; Ohlebusch and Gog 2011] or by constructing the suffix array for smaller text blocks at a time [Kärkkäinen et al. 2013a], but their practical performance is severely reduced.

The $O(n \log n)$ time CPS2 algorithm [Chen et al. 2008] uses the full suffix array but only $2n + o(n)$ bits of extra space in addition to the suffix array. CPS2 is the laziest of all efficient algorithms, which shows in its performance. While slow for nonrepetitive inputs, its performance for repetitive inputs matches many linear-time algorithms [Al-Hafeedh et al. 2012].

In practical terms, assuming four byte integers and one byte characters, all the linear-time algorithms use at least $13n$ bytes of space in total (including the input), while CPS2 uses less than $6n$ bytes.

1.2. Our Contribution

With the previous applications in mind, in this article we describe several efficient methods for computing the LZ77 factorization with different degrees of laziness. Our aim was to develop fast, practical algorithms that operate in a memory range common with previous algorithms for the problem, while still offering strong asymptotic guarantees on performance.

Our first contribution is to describe several improvements to the laziest of all efficient algorithms, CPS2, without changing its lazy nature. The resulting algorithm is the fastest of all algorithms for extremely repetitive inputs.

Second, we describe a family of algorithms occupying a new range in the laziness spectrum, not as lazy as CPS2 but lazier than the algorithms of Goto and Bannai [2013]. The algorithms are space efficient and fast in practice, and while they do not operate in linear time, they come with solid asymptotic guarantees on performance. A preliminary version of these algorithms appeared in Kempa and Puglisi [2013]. Two highlights are (1) an algorithm that uses $6n$ bytes of memory, and is many times faster than even our improved CPS2 except for extremely repetitive inputs; and (2) an algorithm using $9n$ bytes that is faster than all previous algorithms in the literature for all types of inputs (usually by a factor of almost 2).

Finally, we introduce two linear-time algorithms that in laziness terms are similar to the algorithms by Goto and Bannai [2013]. The first algorithm uses $3n \log n$ bits of working space and $13n$ bytes in total. It can be seen as a reorganization of an algorithm by Goto and Bannai [2013], but 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 linear-time algorithm reduces the working space to $2n \log n$ bits and total space to $9n$ bytes, which is at least $n \log n$ bits or $4n$ bytes less than any previous linear-time algorithm uses in the worst case. The space reduction does not come at a great cost in performance; it is competitive in practice for all types of input. The algorithm relies on novel combinatorial observations that might be of independent interest.

The linear-time 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 amount of RAM used by further $n \log n$ bits, but also speeds up the computation when the time for reading inputs from disk is taken into account. These linear-time algorithms appeared in preliminary form in Kärkkäinen et al. [2013b].

We present an extensive experimental comparison of our new algorithms to relevant prior algorithms in Section 6. In addition, throughout the article, we present smaller experiments involving different variants of our algorithms. The small experiments are done using two files: a typical nonrepetitive file (english) and a highly repetitive file (kernel). Further details about the files and the experimental setting can be found in Section 6.

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. The first, $n-1$ symbols are drawn from an ordered alphabet Σ of size $|\Sigma| = \sigma$ and $\mathsf{X}[n]$ is a special "end of string" symbol $\$ \notin \Sigma$, smaller than all symbols from 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* (or *factor*) $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$ 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 $1, \ldots, n$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ if and only if $\mathsf{X}[i..n]$ is the j^{th} suffix of X in ascending lexicographical order. The inverse suffix array ISA is the inverse permutation of SA, that is, $\mathsf{ISA}[i] = j$ if and only if $\mathsf{SA}[j] = i$. Conceptually, $\mathsf{ISA}[i]$ tells us the position of suffix i in SA.

The array $\Phi[0..n]$ (see Kärkkäinen et al. [2009]) 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.

BWT and LF. The Burrows-Wheeler transform [Burrows and Wheeler 1994] $\mathsf{BWT}[1..n]$ is a permutation of X such that $\mathsf{BWT}[i] = \mathsf{X}[\mathsf{SA}[i] - 1]$ if $\mathsf{SA}[i] > 1$ and $\mathsf{X}[n]$ otherwise. We also define $\mathsf{LF}[i] = j$ if and only if $\mathsf{SA}[j] = \mathsf{SA}[i] - 1$, except when $\mathsf{SA}[i] = 1$, in which case $\mathsf{LF}[i] = \mathsf{ISA}[n]$. Let $\mathsf{C}[1..\sigma]$ be an array such that $\mathsf{C}[c]$, for symbol $c \in \Sigma$, is the number of symbols in X lexicographically smaller than c. The function $\mathsf{rank}(i)$ returns the number of occurrences of symbol $\mathsf{BWT}[i]$ in $\mathsf{BWT}[1..i]$. It is well known that $\mathsf{LF}[i] = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}(i)$ [Ferragina and Manzini 2005; Navarro and Mäkinen 2007].

LZ77. The LZ77 factorization uses the notion of a *Longest Previous Factor* (LPF). The LPF at position i (denoted $\mathrm{LPF}[i]$) 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 j^{th} LZ factor (or *phrase*) in the parsing is to start at position i, then we output (p_i, ℓ_i) (to represent the j^{th} phrase), and then the $(j+1)^{\text{th}}$ 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. 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. We further define

$$\mathsf{NSV}_{\mathsf{text}}[i] = \mathsf{SA}[\mathsf{NSV}_{\mathsf{lex}}[\mathsf{ISA}[i]]], \tag{1}$$

$$\mathsf{PSV}_{\mathsf{text}}[i] = \mathsf{SA}[\mathsf{PSV}_{\mathsf{lex}}[\mathsf{ISA}[i]]]. \tag{2}$$

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

The usefulness of the NSV/PSV values is summarized by the following easily proved lemma due to Crochemore and Ilie [2008] (and later restated by Ohlebusch and Gog [2011]).

LEMMA 1 (CROCHEMORE AND ILIE [2008]). *For $i \in [1..n]$, let $i_{nsv} = \mathsf{NSV}_{\mathsf{text}}[i]$, $i_{psv} = \mathsf{PSV}_{\mathsf{text}}[i]$, $\ell_{nsv} = \mathsf{lcp}(i, i_{nsv})$, and $\ell_{psv} = \mathsf{lcp}(i, i_{psv})$. Then*

$$\mathrm{LPF}[i] = \begin{cases} (i_{nsv}, \ell_{nsv}) & \text{if} \quad \ell_{nsv} > \ell_{psv} \\ (i_{psv}, \ell_{psv}) & \text{if} \quad \ell_{psv} = \max(\ell_{nsv}, \ell_{psv}) > 0 \\ (\mathsf{X}[i], 0) & \text{if} \quad \ell_{nsv} = \ell_{psv} = 0. \end{cases}$$

Lazy LZ Factorization. The eager approach to LZ factorization employed by most linear-time algorithms is to first compute the full LPF array, that is, the pairs (p_i, ℓ_i) for all $i \in [1..n]$. Then, the actual parsing phase is trivial and very fast.

A lazier approach is to compute the NSV/PSV values for all i, but LPF[i] only if i is a starting position of a phrase. This is lazier because only $2z$ lcp values need to be computed. The eager algorithms need n lcp values, and while they can be computed in $O(n)$ time, this is not cheap. This approach was introduced by Goto and Bannai [2013] and is employed by our new algorithms described in Section 5.

An even lazier approach is to compute the NSV/PSV values for phrase starting positions only. Our algorithms in Section 4 are the first to use this strategy. They precompute a data structure that supports efficient NSV/PSV queries.

The laziest of all efficient algorithms, CPS2 [Chen et al. 2008], computes no NSV/PSV values at all. In Section 3, we describe several optimizations that improve the running time of CPS2 without changing its extreme laziness.

3. SPEEDING UP A LAZY LZ77 ALGORITHM

Before getting to the new algorithms, we describe in this section a series of optimizations to a factorization algorithm due to Chen et al. [2008], called CPS2. The original algorithm has two interesting properties: first, until recently, it was unique among LZ77 factorization algorithms in that it avoids computation of the LCP array (see Kärkkäinen et al. [2009]). For this reason, it is one of the most space-efficient algorithms known, even considering algorithms that use compressed data structures [Okanohara and Sadakane 2008; Ohlebusch and Gog 2011]. Secondly, it computes factors lazily, in order, one factor at a time, avoiding computing longest previous factors for all n positions in the input first.

CPS2 makes use of SA, which is preprocessed for Range Minimum Queries (RMQs) [Fischer and Heun 2011]. An RMQ $\mathsf{rmq}(i, j)$ returns the position of the minimum value in $\mathsf{SA}[i..j]$. Practical implementations of data structures supporting fast rmq are now well established.

To compute the factor starting at position i, CPS2 works in ℓ_i rounds. In round 1 it computes the range of the suffix array $\mathsf{SA}[s_1..e_1]$ containing all the suffixes having $\mathsf{X}[i]$ as a prefix. In a generic round j CPS2 maintains the invariant that its *active range*, $\mathsf{SA}[s_j..e_j]$, contains all the suffixes prefixed with $\mathsf{X}[i..i+j-1]$. However, it also enforces, via $\mathsf{rmq}(s_j, e_j)$, that at least one suffix in $\mathsf{SA}[s_j..e_j]$ begins at some position $q_j < i$. Of course, q_j is potentially an LPF for position i, and as soon as the active range does not hold a suffix less than i, the LZ77 factor for i is known, and we can output $(p_i, \ell_i) = (q_{j-1}, j-1)$.

CPS2 moves from one round to the next, and from range $\mathsf{SA}[s_j..e_j]$ to range $\mathsf{SA}[s_{j+1}..e_{j+1}]$, by binary searching to find the extents s_{j+1} and e_{j+1}, considering the $(j+1)^{\text{th}}$ symbols of the suffixes in $\mathsf{SA}[s_j..e_j]$. This is correct because of the lexicographic ordering of the SA. In effect, the suffix $\mathsf{X}[i..n]$ is being treated as a pattern and searched for one symbol at a time in SA.

3.1. Fast Interval Table Computation

An important optimization to CPS2 that is not described in Chen et al. [2008] but that appears in the source code of the algorithm's implementation is the computation of a lookup table storing the extents of the interval for each symbol in the suffix array, and the minimum value in that interval. More precisely, for each distinct symbol c in X the table stores a tuple (s_c, e_c, m_c), such that all suffixes prefixed with c lie in $\mathsf{SA}[s_c..e_c]$, and m_c is the minimum value in $\mathsf{SA}[s_c..e_c]$.

Assuming the alphabet is a small constant (the usual 256 symbols, say) this table is small and can be accessed in constant time. For each factor, looking up the interval in the table allows the first round of the successive binary search process to be bypassed, avoiding some cache misses, and leading to a consistent improvement in overall factorization times.

Because we are interested in total factorization time, the time to initialize the lookup table matters. In the previously mentioned CPS2 code, s_c and e_c are computed by scanning SA left-to-right and observing where $\mathsf{X}[\mathsf{SA}[i]] \neq \mathsf{X}[\mathsf{SA}[i-1]]$—as it is at these points where one interval ends and another starts. However, because of the unpredictable order of the values in SA, computing intervals this way causes roughly one cache miss each time we access X to examine a symbol.

This leads us to our first optimization. Instead of scanning SA and repeatedly accessing X in SA order, we instead scan X, in a cache-friendly left-to-right manner. During the scan, we increment a counter for each symbol, and later prefix sum these counters to obtain the correct (s_c, e_c) intervals of the SA for each symbol. During the scan of X, we can also trivially compute the minimum in each interval: m_c is simply the position of the first occurrence of c in X.

A further optimization is to compute two levels of lookup tables: one for single symbols and one for symbol pairs, of which there are at most 2^{16}. This allows us to skip two rounds of binary search instead of one, and because the bigram table is still small enough to fit in cache, it does not greatly increase initialization time (the time spent scanning X to build the lookup tables). For big files, the increase in memory consumption from the bigger table is negligible.

3.2. Scanning Small Ranges

A further improvement to CPS2 makes use of the NSV/PSV values in a lazy way. During the binary search phase of CPS2, when the size $e_j - s_j + 1$ of the range drops below a

```
Procedure LZ-Factor(i, nsv, psv)
1: ℓ_nsv ← lcp(i, nsv)
2: ℓ_psv ← lcp(i, psv)
3: if ℓ_nsv > ℓ_psv then
4:       (p, ℓ) ← (nsv, ℓ_nsv)
5: else
6:       (p, ℓ) ← (psv, ℓ_psv)
7: if ℓ = 0 then p ← X[i]
8: output factor (p, ℓ)
9: return i + max(ℓ, 1)
```

```
Procedure LZ-Factor(i, nsv, psv)
1: ℓ ← lcp(nsv, psv)
2: if X[i + ℓ] = X[nsv + ℓ] then
3:       ℓ ← ℓ + 1
4:       (p, ℓ) ← (nsv, ℓ + lcp(i + ℓ, nsv + ℓ))
5: else
6:       (p, ℓ) ← (psv, ℓ + lcp(i + ℓ, psv + ℓ))
7: if ℓ = 0 then p ← X[i]
8: output factor (p, ℓ)
9: return i + max(ℓ, 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.

predefined threshold t, we stop using binary search further and instead scan the range in $O(t)$ time to find $\mathsf{NSV}_{\mathsf{text}}[i]$ and $\mathsf{PSV}_{\mathsf{text}}[i]$. We then compute (p_i, ℓ_i) by Lemma 1.

Setting $t = O(\log n)$ preserves the $O(n \log n)$ overall runtime of CPS2, but the scanning scheme only requires three random accesses and so is faster than further binary searching, which even on small ranges can still attract two or more random accesses per round when looking up characters from X to narrow the current range. In practice, we found $t = 4{,}096$ to give the best performance.

3.3. Optimized Parsing

Finally, a small additional speed-up is possible. Figure 1 shows two versions of the basic parsing procedure. The standard version treats nsv and psv as independent suffixes. The optimized version is another small contribution of this article. 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.

3.4. Experiments

Our optimizations to CPS2 are summarized in Figure 2, which shows the incremental improvement to runtime achieved by cache-sensitive single-symbol interval computation, two-symbol intervals, and finally scanning of small ranges. The right of the figure shows times for several different settings of t. By far the biggest boost comes from the improved interval computation, but the other tricks consistently improve performance. The improved version of CPS2 is called CPS2I.

4. FACTORIZATION AND THE INVERSE SUFFIX ARRAY

Our second improvement to CPS2 used Lemma 1 as a way to abandon further binary search steps in favor of fast short sequential scans of SA and the text. The family of algorithms in this section exploit Lemma 1 in a different way: they use the Inverse Suffix Array, ISA, to first locate i in SA at position $\mathsf{ISA}[i]$, and then search out in SA in either direction from that position, to locate $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$.

The simplest implementation of this scheme is to store ISA explicitly, using $4n$ bytes, and to sequentially scan SA to find $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$. We call this algorithm LZ9—it uses $9n$ bytes in total for SA, ISA, and X. To compute the LZ77 factor starting at position i, we use ISA to locate i in SA in constant time. We then scan left and right in SA to find $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$. The sum of the lengths of the scans is clearly at most n, the size of SA. Over all z factors the runtime is thus $O(nz)$ in the worst case.

We had initially hoped that a tighter analysis of LZ9 would lead to a better worst-case bound, but the following string illustrates that the algorithm may indeed run

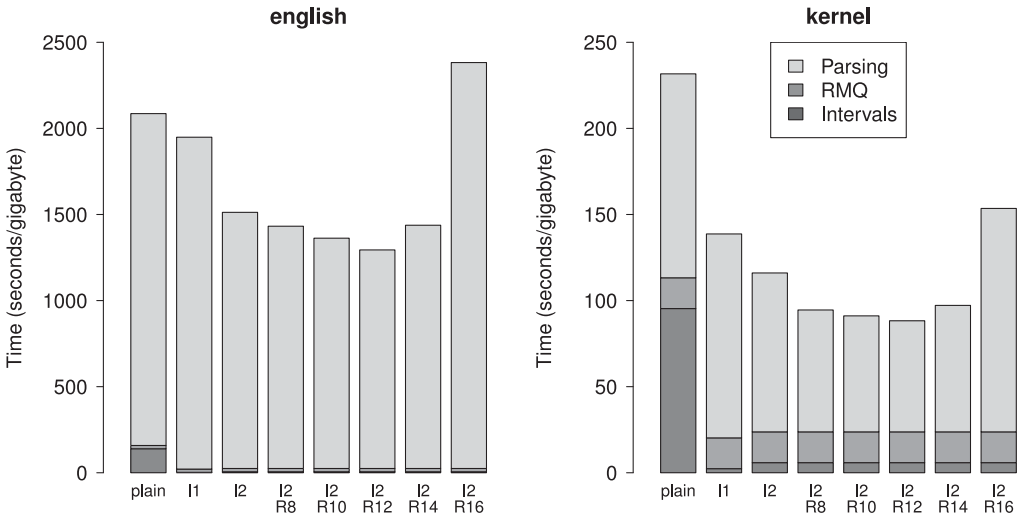

Fig. 2. Improvements to runtime for various optimizations to the CPS2 LZ77 factorization algorithm. Ix is the fast interval computation optimization using x levels of lookup tables. Ry is the small ranges scanning trick with $t = 2^y$.

in superlinear time. Let N_v be the $\log n$-bit binary code of the number $v \in [0..n)$. For example, if $\log n = 2$, $N_0 = 00$, $N_1 = 01$, $N_2 = 10$, and $N_3 = 11$. Now, let $u = \log n + 1$ and consider the following binary string:

$$\mathsf{Y} = 0^u 1 N_0 1 0^u 1 N_1 1 \ldots 0^u 1 N_j 1 \ldots 0^u 1 N_n 1.$$

The initial segment of the SA of Y contains suffixes prefixed with 0^u. There are $O(n/\log n)$ of these suffixes, and they occur in increasing order in SA, that is, the segment of SA in which they lie looks like $0, k, 2k, 3k, \ldots$ where $k = 2\log n + 3$.

Now consider the operation of LZ9 when factorizing Y. When a factor starts at a position $i = 0 \pmod k$, then the algorithm will scan SA left and right from position $\mathsf{ISA}[i]$. Because the elements in this segment of SA are increasing, the scan left for $\mathsf{PSV}_{\mathsf{text}}[i]$ will stop immediately; however, the scan right from $\mathsf{NSV}_{\mathsf{text}}[i]$ will go (at least) to the right end of the segment, and so will require $O(n/\log n)$ time. If we have to do this for every $j = 0 \pmod k$, overall runtime will be $O((n/\log n)^2)$.

Although this analysis is not rigorous, it does suggest a bad case exists, and prompted us to generate a 90 megabyte instance of string Y. CPS2 factorized the file in 86 seconds, while LZ9 labored away for 4 minutes and 23 seconds.

4.1. Adding Asymptotic Guarantees

The dismal performance of LZ9 on string Y is a result of the algorithm sequentially scanning SA from $\mathsf{ISA}[i]$ to find $\mathsf{PSV}_{\mathsf{text}}[i]$ and $\mathsf{NSV}_{\mathsf{text}}[i]$. This scanning can be avoided if we first preprocess SA and build a data structure to answer NSV/PSV queries. We found the NSV/PSV data structure of Abeliuk et al. [2013] was perfect for our needs, being space efficient, fast to answer queries, and fast to initialize. Without getting into too many details, the data structure offers a space-time trade-off, namely, it requires $4n/\hat{b}$ bytes and answers queries in $O(\hat{b} + \log(n/\hat{b}))$ time.

We call this version of LZ9 with an auxiliary NSV/PSV data structure ISA9. Setting $\hat{b} = O(\log n)$ ensures ISA9 runs in $O(n + z\log n)$ time overall. In practice, we found a higher value of $\hat{b}$ led to faster runtimes, and allowed us to reduce space overheads to a negligible level. When using the NSV/PSV data structure to find $\mathsf{PSV}_{\mathsf{text}}[i]$ and

$\mathsf{NSV}_{\mathsf{text}}[i]$, the runtime for ISA9 on string Y above is reduced to a very respectable 4.9 seconds. For brevity, from this point onward we assume $\hat{b} = \mathrm{O}(\log n)$.

4.2. Reducing Space Requirements

We now show how to reduce the space requirements of ISA9 by a more compact representation of ISA. A well known property of the LF array (see Section 2) is $\mathsf{ISA}[i-1] = \mathsf{LF}[\mathsf{ISA}[i]]$. This property is the essence of the Burrows-Wheeler Transform (BWT) inversion algorithm [Burrows and Wheeler 1994; Kärkkäinen and Puglisi 2010] and the FM-index [Navarro and Mäkinen 2007]. With this property in mind, our approach is to sparsify ISA and store only every k^{th} value in it. These *sample* values are stored in an array of n/k values and can still be accessed in constant time. Any nonsample value $i \neq 0 \pmod k$ can be recovered when needed by looking up the first sample larger than i, $j = \mathsf{ISA}[\lfloor i/k \rfloor + 1]$, and then following the LF mapping $k - i \bmod k$ times starting from $\mathsf{LF}[j]$.

The problem is now to represent LF compactly. Below we describe two approaches we found to be effective in practice. The first one implements LF with rank queries on the BWT. The second uses a sparse representation of LF and exploits the presence of SA.

Rle-LF. As mentioned in Section 2, LF can be implemented using the equation $\mathsf{LF}[i] = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}(i)$. The space needed for C is negligible but we need compact representation of rank. Data structures for supporting rank are well studied (see, e.g., Kärkkäinen and Puglisi [2010]), and we implemented and tested many of them. As with the NSV/PSV data structure, we require a solution that answers queries quickly, but is also fast to initialize and memory efficient. We found the following approach to be best for highly repetitive inputs.

A high degree of repetition in X is manifest as *runs* of equal letters in the BWT of X (see, e.g., Manzini [2001]). Let r be the number of runs in BWT. For each run, we store its starting position in BWT, say j, and the number of occurrences of $\mathsf{BWT}[j]$ before position j in BWT. To answer $\mathsf{rank}(i)$ we binary search over the starting positions of the runs, to locate the starting position of the run that i falls in, say j. The answer to $\mathsf{rank}(i)$ is then $\mathsf{rank}(j)$, which is stored earlier with j, plus $i - j + 1$, the number of occurrences of $\mathsf{BWT}[i]$ between i and j.

This solution requires $8r$ bytes assuming 4 byte integers and answers $\mathsf{rank}(i)$ in $\mathrm{O}(\log r)$ time, and so factorizes in $\mathrm{O}(n + zk \log r + z \log n)$ time overall. The version in the main experiments uses $k = 8$ so that the sparse ISA needs $0.5n$ bytes. Since this approach is effective only for highly repetitive files, we use it only when $n/r \geq 16$ so that the rank data structure fits in $0.5n$ bytes too. The total space requirement is then at most $6n$ bytes and thus we call this algorithm ISA6r.

Sparse-LF. Our second approach to computing LF makes no assumption about the repetitiveness of the input, and exploits the presence of SA. This is different from the usual contexts in which LF is computed: in inversion and indexing only the BWT is available. For each symbol c we store the position of every b^{th} occurrence of c in BWT, storing n/b integers in total over all symbols. When we want to compute $\mathsf{LF}[i]$ we first inspect $c = \mathsf{BWT}[i] = \mathsf{X}[\mathsf{SA}[i] - 1]$ (note: we do not store BWT explicitly), and then binary search symbol c's list to find the largest position in the list less than i, say j. We call j an approximate rank value—it allows us to estimate $\mathsf{LF}[i]$, and points us to a place in SA that must be within b positions of the position we seek (i.e., the true $\mathsf{LF}[i]$ value). Finally, we scan SA to the right of the approximate value of $\mathsf{LF}[i]$ until we find the suffix with value $\mathsf{SA}[i] - 1$. The position of this value is $\mathsf{ISA}[\mathsf{SA}[i] - 1]$—which is our goal. This approach avoids scanning BWT (which we would like to avoid computing

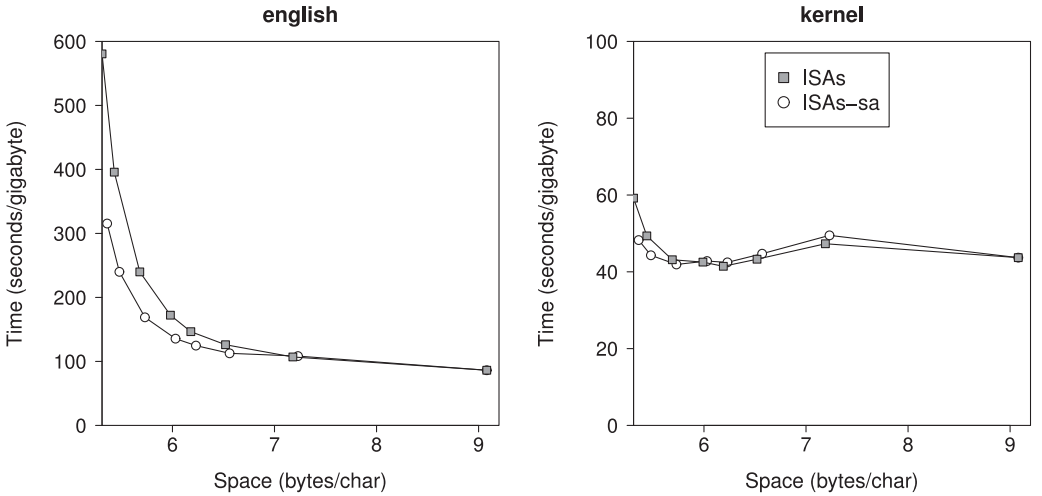

Fig. 3. Space-time trade-off for ISA*s* and ISA*s-sa* algorithms. The parameter controlling the trade-off is the ISA sampling rate.

on-the-fly because of cache misses). At query time, scanning of SA is fast, and causes no extra cache misses. This approach uses $4n/b$ bytes to represent LF and factorizes in $O(n + zkb + zk \log(n/b) + z \log n)$ time. Setting $b = O(\log n)$ yields $O(n + zk \log n)$ complexity. In practice, we set k to be a small constant (see below) and so expect the running time $O(n + z \log n)$. We found $b = 64$ to be a good value for the occurrence sampling rate. We call this algorithm ISA*s*.

As an additional optimization, we also store a compact representation of LF^2 defined by $\mathsf{LF}^2[i] = \mathsf{LF}[\mathsf{LF}[i]]$ (cf. Kärkkäinen et al. [2012]). This approximately halves the number of applications of LF or LF^2 while using another $4n/b$ bytes. To keep the space requirements about the same, we double the value of b to 128. The representation of LF^2 is essentially the same as that of LF except the C array and the lists of every b^{th} occurrence are defined for pairs of symbols instead of individual symbols. Thus, this is a kind of superalphabet version of the algorithm and we call it ISA*s-sa*. Figure 3 compares the performance of ISA*s* and ISA*s-sa*. Since ISA*s-sa* is sometimes significantly faster and never much slower, we exclude ISA*s* from further experiments.

Figure 4 shows the space-time trade-off of ISA*s-sa* in more detail. The trade-off parameter is the ISA sampling rate k. The largest space version with $k = 1$ is in fact ISA9. While on the nonrepetitive file (english) the space-time curve smoothly drops down as the available memory increases, the time for the repetitive kernel file actually increases around $7n$ when the algorithm is sampling ISA at a higher rate. This is because highly repetitive files do not benefit from having the full ISA available—the majority of parsing time is spent in NSV/PSV calculations and symbols comparisons.

For further experiments, we chose a version of ISA*s-sa* with $k = 5$. This version uses less than $6n$ bytes of space and we call it ISA6*s-sa*.

5. LINEAR-TIME ALGORITHMS

The algorithms described in the previous sections have a superlinear-time complexity in general. In this section, we describe linear-time algorithms with small space requirement and fast running time in practice.

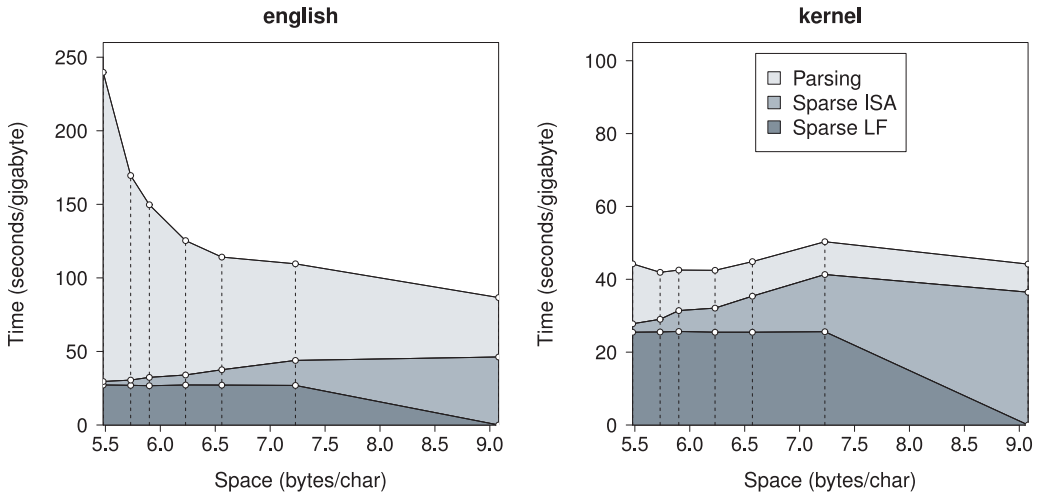

Fig. 4. Detailed space-time trade-off for the ISA*s-sa* LZ77 factorization algorithm with different ISA sampling rates. The different shaded areas show the amount of time spent in each phase. The time for initialization of the NSV/PSV data structure is negligible, hence we omit it here.

5.1. 3*n* log *n*-Bit Algorithm

Our first linear-time algorithm is closely related to the algorithms of Goto and Bannai [2013], particularly BGT and BGS. It first computes the $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ arrays and uses them for lazy LZ factorization similarly to the BGT algorithm (lines 11–13 in Figure 5). However, the NSV/PSV values are computed using the technique of the BGS algorithm, which comes originally from Crochemore and Ilie [2008].

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 Figure 5. 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 anymore (see preceding). 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.

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: $i \leftarrow$ LZ-Factor$(i, \mathsf{NSV}_{\mathsf{text}}[i], \mathsf{PSV}_{\mathsf{text}}[i])$

Fig. 5. LZ factorization using $3n \log n$ bits of working space (the arrays SA, $\mathsf{NSV}_{\mathsf{text}}$, and $\mathsf{PSV}_{\mathsf{text}}$).

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 Goto and Bannai [2013].

5.2. 2*n* log *n*-Bit Algorithm

Our second linear-time 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 [Ohlebusch and Gog 2011; Goto and Bannai 2013].

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 Figure 6. 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 $\Phi[t+1..n] = \mathsf{NSV}_{\mathsf{text}}[t+1..n]$.

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.

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

5.3. Getting Rid of the Stack

The preceding 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 [2013], is to replace the stack with $\mathsf{PSV}_{\mathsf{text}}$ pointers. As observed in Section 5.1, if j is the suffix on the top of the stack, then the next suffixes in the stack are $\mathsf{PSV}_{\mathsf{text}}[j]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{PSV}_{\mathsf{text}}[j]]$, etc. 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 Figure 5) 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 Figure 7.

The versions without an explicit stack are slightly slower because of the nonlocality 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 (e.g., 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

Algorithm KKP2n
1: $top \leftarrow 0$ // top of stack
2: **for** $i \leftarrow 1$ **to** n **do**
3: **while** $top > \mathsf{SA}[i]$ **do**
4: $top \leftarrow \Phi^{-1}[top]$ // pop from stack
5: $\Phi^{-1}[\mathsf{SA}[i]] \leftarrow top$ // $\Phi^{-1}[\mathsf{SA}[i]] = \mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[i]]$
6: $top \leftarrow \mathsf{SA}[i]$ // push to stack
7: $\Phi^{-1}[0] \leftarrow 0$
8: $next \leftarrow 1$
9: **for** $t \leftarrow 1$ **to** n **do**
10: $psv \leftarrow \Phi^{-1}[t]$
11: $nsv \leftarrow \Phi^{-1}[psv]$
12: **if** $t = next$ **then**
13: $next \leftarrow$ LZ-Factor(t, nsv, psv)
14: $\Phi^{-1}[t] \leftarrow nsv$
15: $\Phi^{-1}[psv] \leftarrow t$

Fig. 7. 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.

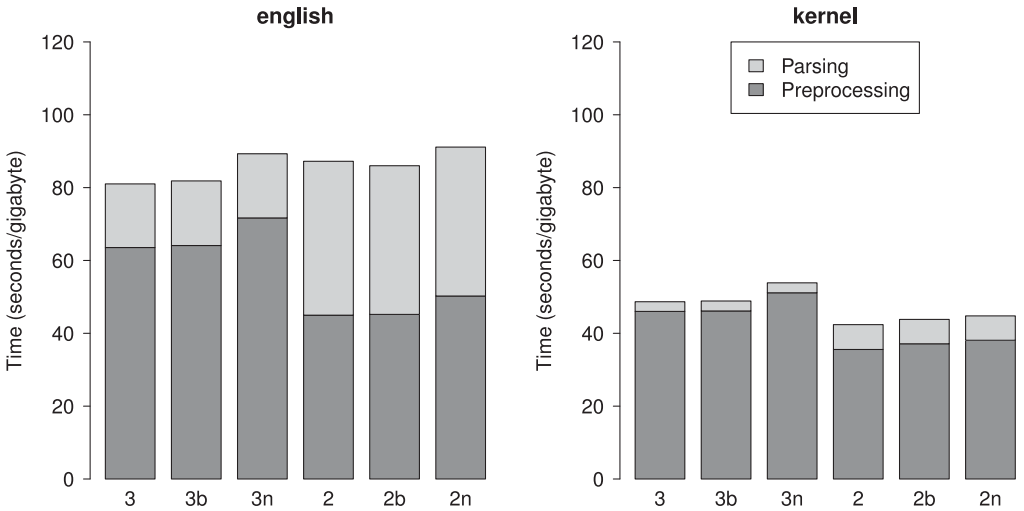

Fig. 8. Runtime breakdown into two main phases for all variants of KKP algorithm.

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.

Figure 8 shows the runtimes of our linear-time algorithms with different stack variants. In the main experimental comparison, we use the versions that overwrite the suffix array, but as the figure demonstrates, the versions with a stack buffer are equally fast.

6. EXPERIMENTAL RESULTS

We implemented the algorithms described in this article, and compared their performance in practice to algorithms of Goto and Bannai [2013] and Ohlebusch and Gog [2011]. The main experiment measured the time to compute the LZ factorization of the text. All algorithms take the text and the suffix array as input and so we omit

Table I. Files used in the Experiments

Name	Abbr.	σ	n/r	n/z	Source	Description
proteins	pro	25	1.87	9.57	S	Swissprot database
english	eng	220	2.70	13.77	S	Gutenberg Project
dna	dna	16	1.60	14.65	S	Human genome
sources	src	228	4.37	17.67	S	Linux and GCC sources
random256	r8	256	1.00	2.89	-	Random bytes
random2	r1	2	2.00	26.04	-	Random bits
coreutils	cor	236	34	110	R/R	GNU Coreutils sources
cere	cer	5	15	112	R/R	Baking yeast genomes
kernel	ker	160	61	214	R/R	Linux Kernel sources
einstein.en	ein	124	1195	3634	R/R	Wikipedia articles
tm29	tm	2	1,990K	2,912K	R/A	Thue-Morse sequence
rs.13	rs	2	2,711K	3,024K	R/A	Run-Rich String sequence

Note: The files 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 values of n/z (the average length of a phrase in the LZ factorization) and n/r (the average length of a run in a BWT) are included as a measure of repetitiveness.

Table II. Time and Space Consumption for Computing LZ Factorization/LPF Array

	Alg.	Mem	pro	eng	dna	src	r8	r1	cor	cer	ker	ein	tm	rs
LZ factorization	KKP3	13n	**74.5**	**75.7**	**81.7**	**50.5**	**103.5**	79.7	43.6	63.2	45.7	56.9	38.2	77.8
	KKP2	9n	83.9	80.6	92.7	54.7	131.9	82.7	40.2	53.3	41.6	43.6	35.1	49.0
	ISA9	9n	89.3	80.8	83.6	56.4	170.4	**70.4**	39.8	51.5	40.8	44.0	35.1	50.6
	ISA6s-sa	6n	154.4	126.2	122.7	89.5	406.4	87.8	42.7	**47.1**	40.0	34.0	38.9	37.8
	ISA6r	6n	-	-	-	-	-	-	**38.8**	48.3	**35.7**	**29.7**	33.9	34.2
	CPS2I	6n	1360	1300	2275	685.3	1002	1671	118.1	404.0	89.2	30.1	**21.4**	**22.8**
	iBGS	17n	99.8	93.2	97.5	69.3	164.0	86.2	51.5	65.5	52.9	60.0	44.1	59.5
	iBGL	17n	123.2	108.6	113.4	77.8	236.6	94.3	52.2	66.1	53.0	58.6	44.2	59.5
	iBGT	13n	171.4	153.9	188.0	99.8	258.2	168.0	55.4	84.1	56.2	52.8	44.4	56.5
	iOG	13n	215.0	192.4	249.9	124.9	301.1	233.8	67.8	105.3	66.6	61.1	50.3	63.2
LPF	KKP3-LPF	13n	**115.5**	**112.9**	**133.5**	**71.1**	**150.8**	**139.4**	56.0	88.0	58.0	63.5	49.2	82.8
	KKP2-LPF	13n	140.3	132.4	167.2	83.6	217.6	168.7	**54.6**	**82.6**	**55.6**	**51.3**	**41.1**	**58.0**
	iOG	13n	210.1	188.0	243.7	121.3	303.0	232.3	66.8	104	66.4	60.6	50.3	62.7
	LPF-online	13n	160.4	162.3	187.2	114.2	188.6	185.9	103.5	137.2	109.6	127.1	100.4	148.0

Note: 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 fastest time for each input file is shown in bold. 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.

the time to compute SA. The datasets used in experiments are described in detail in Table I. All algorithms use the optimized version of `LZ-Factor` (Figure 1), which slightly reduces the time (e.g., for KKP3 by 2% on nonrepetitive 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 II. In all cases, algorithms introduced in this article outperform the algorithms of Goto and Bannai [2013] (which are, to our knowledge, the fastest alternative LZ factorization

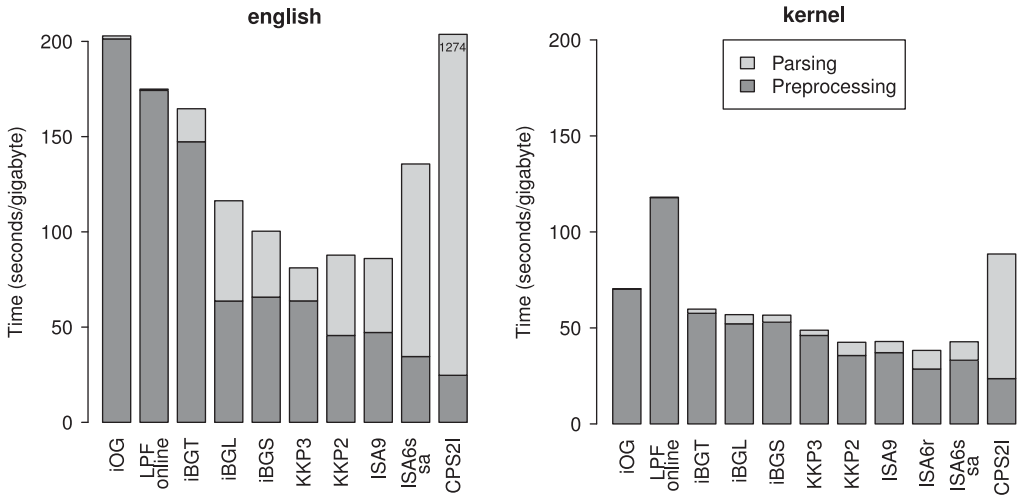

Fig. 9. Runtimes of all algorithms on example test files showing the time distribution between two main phases. We define the parsing phase to begin when algorithm starts printing LZ77 phrases.

algorithms) while using the same or less space. In particular, the KKP2 algorithm is always faster and simultaneously uses at least $n \log n$ bits less space. A notably big difference is observed for nonrepetitive data, where KKP3 significantly dominates all prior solutions.

For highly repetitive data our space-efficient algorithms (using only $6n$ bytes) are even faster, which is clearly due to the laziness of these algorithms. A particularly extreme case is CPS2I, which is the fastest on the artificially repetitive inputs. This effect is more thoroughly investigated in Figure 9 which shows a runtime breakdown of all algorithms into preprocessing and parsing stage on the example nonrepetitive (english) and repetitive (kernel) files. While the best algorithms for nonrepetitive input occupy a mid-range in the degree of laziness, for highly repetitive strings (which require little time in the parsing stage) the fastest algorithms are among the laziest.

Finally, some comments on alphabet size are in order. All our experiments use files where the alphabet size is at most 256, and thus small relative to the length of the string, which is the usual case in practice. Larger alphabets will tend to favor the linear-time algorithms, all of which have runtime and space usage independent of σ. The alphabet size mainly affects the space requirements of the superlinear algorithms. The ISA-based algorithms use the C array, which requires σ integers. The superalphabet optimization of the ISA algorithms introduces a $O(\sigma^2)$ space term, and so is unsuitable for larger alphabets. Similarly, CPS2I, tested here with two levels of lookup tables, will be unsuitable for large alphabets as these tables require $O(\sigma^2)$ space.

Full LPF Array. Our linear-time algorithms can be modified to compute the full LPF array, that is, the set of longest previous factors (p_i, ℓ_i) for all $i \in [1..n]$, 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 [2008, Figure 2]. We compared this approach to the fastest algorithms for computing the LPF array by Ohlebusch and Gog [2011] (with the interleaving optimization by Goto and Bannai [2013]) and LPF-online from Crochemore et al. [2009] (see Kempa and Puglisi [2013] and Ohlebusch and Gog [2011] for comparison). For LCP array computation in those algorithms we used the fastest version of the Φ algorithm, which consumes $13n$ bytes of space [Kärkkäinen et al. 2009].

Table III. Times for Computing LZ Factorization, Taking Into Account the Disk Reading Time

Alg.	pro	eng	dna	src	r8	r1	cor	cer	ker	ein	tm	rs
KKP1s	106.5	100.2	109.0	86.6	161.5	113.4	71.0	74.9	68.4	67.6	66.1	66.1
KKP2b	150.6	143.7	155.7	117.8	209.7	160.1	103.6	115.9	102.9	107.3	96.8	111.6

Note: 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.

As shown in Table II, the modified KKP2 algorithm consistently outperforms the older methods. When the input is not repetitive, the LPF variant of KKP3 is even faster.

Streaming. As explained in Section 5.3, our new algorithms can be implemented so that SA is only accessed sequentially in a read-only manner, allowing it to be streamed from 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 disk and disk reading time is included in the total runtime. Reading from 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 III. 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 Wu [2009]).

7. CONCLUSIONS AND FUTURE WORK

The algorithms we have described for LZ77 factorization improve on prior methods across the space-time spectrum. Our results indicate that in practice the fastest way to compute the LZ77 factorization seems to be to compute the factors in an online manner (after SA construction and some other light preprocessing), one after the other, rather than computing all LPF values and then selecting only those involved in the parsing. Computation of the LCP array also seems unnecessary: all our algorithms use the SA with alternative supporting data structures, which are smaller and faster to initialize than the LCP array.

Many interesting questions remain. All the LZ77 factorization algorithms examined here (new and old) make use of the suffix array, and so require memory at least sufficient to store n integers. An important direction for future study, especially for text indexes based on LZ77, is to develop scalable external memory and distributed factorization algorithms that avoid holding the whole suffix array in memory, or that otherwise avoid suffix sorting the entire input altogether. A recent first step in the latter direction is Kärkkäinen et al. [2013a].

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 datasets.

ACKNOWLEDGMENTS

Thanks go to Golnaz Badkobeh, Maxime Crochemore, and Travis Gagie for inspiring discussions on the topic of LZ77 factorization; to Keisuke Goto and Hideo Bannai for an early copy of their paper; to Simon Gog and German Tischler for sharing source code, and for explicating details of their experiments; and to the

anonymous referees of this and earlier versions of this work, whose comments have materially improved our manuscript.

REFERENCES

1000 Genomes Project Consortium. 2015. A global reference for human genetic variation. *Nature* 526, 7571 (2015), 68–74.

A. Abeliuk, R. Cánovas, and G. Navarro. 2013. Practical compressed suffix trees. *Algorithms* 6, 2 (2013), 319–351.

A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylova, W. F. Smyth, G. Tischler, and M. Yusufu. 2012. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.* 45, 1 (2012), 5:1–5:17.

G. Badkobeh, M. Crochemore, and C. Toopsuwan. 2012. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE'12)*, Lecture Notes in Computer Science, Vol. 7608. Springer, 61–72.

M. Burrows and D. J. Wheeler. 1994. *A Block Sorting Lossless Data Compression Algorithm*. Technical Report 124. Digital Equipment Corporation, Palo Alto, California.

M. Charikar, E. Lehman, D. Liu, R. Panigrhy, M. Prabhakaran, A. Sahai, and A. Shelat. 2005. The smallest grammar problem. *IEEE Trans. Inf. Theory* 51, 7 (2005), 2554–2576.

G. Chen, S. J. Puglisi, and W. F. Smyth. 2008. Lempel-Ziv factorization using less time and space. *Math. Comput. Sci.* 1, 4 (2008), 605–623.

M. Crochemore and L. Ilie. 2008. Computing longest previous factor in linear time and applications. *Inform. Process. Lett.* 106, 2 (2008), 75–80.

M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. 2009. LPF computation revisited. In *Proceedings of the 20th International Workshop on Combinatorial Algorithms (IWOCA'09)*, Lecture Notes in Computer Science, Vol. 5874. Springer, 158–169.

M. Crochemore, L. Ilie, and W. F. Smyth. 2008. A simple algorithm for computing the Lempel-Ziv factorization. In *Proceedings of the 2008 Data Compression Conference (DCC'08)*. IEEE Computer Society, 482–488.

J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. 2004. Linear-time computation of local periods. *Theor. Comput. Sci.* 326, 1–3 (2004), 229–240.

H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. 2014. Hybrid indexes for repetitive datasets. *Philos. Trans. R. Soc. A* 372, 2016 (2014).

P. Ferragina and G. Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.

P. Ferragina and G. Manzini. 2010. On compressing the textual web. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM'10)*. ACM, 391–400.

J. Fischer and V. Heun. 2011. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.* 40, 2 (2011), 465–492.

T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. 2012. A faster grammar-based self-index. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA'12)*, Lecture Notes in Computer Science, Vol. 7183. Springer, 240–251.

T. Gagie, P. Gawrychowski, and S. J. Puglisi. 2011. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC'11)*, Lecture Notes in Computer Science, Vol. 7074. Springer, 653–662.

K. Goto and H. Bannai. 2013. Simpler and faster Lempel Ziv factorization. In *Proceedings of the 2013 Data Compression Conference (DCC'13)*. IEEE Computer Society, 133–142.

D. Gusfield and J. Stoye. 2004. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.* 69, 4 (2004), 525–546.

C. Hoobin, S. J. Puglisi, and J. Zobel. 2011. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB* 5, 3 (2011), 265–273.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2012. Slashing the time for BWT inversion. In *Proceedings of the 2012 Data Compression Conference (DCC'12)*. IEEE Computer Society, 99–108.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2013a. Lightweight Lempel-Ziv parsing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Lecture Notes in Computer Science, Vol. 7933. Springer, 139–150.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. 2013b. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM'13)*, Lecture Notes in Computer Science, Vol. 7922. Springer, 189–200.

J. Kärkkäinen, G. Manzini, and S. J. Puglisi. 2009. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM'09)*, Lecture Notes in Computer Science, Vol. 5577. Springer, 181–192.

J. Kärkkäinen and S. J. Puglisi. 2010. Medium-space algorithms for inverse BWT. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, Lecture Notes in Computer Science, Vol. 6346. Springer, 451–462.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936.

D. Kempa and S. J. Puglisi. 2013. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the 2013 Workshop on Algorithm Engineering and Experiments (ALENEX'13)*. SIAM, 103–112.

T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. 2012. A linear time algorithm for seeds computation. In *Proceedings of the 23th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12)*. SIAM, 1095–1112.

R. Kolpakov, G. Bana, and G. Kucherov. 2003. mreps: Efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Res.* 31, 13 (2003), 3672–3678.

R. Kolpakov and G. Kucherov. 2003. Finding approximate repetitions under Hamming distance. *Theor. Comput. Sci.* 303, 1 (2003), 135–156.

S. Kreft and G. Navarro. 2010. LZ77-like compression with fast random access. In *Proceedings of the 2010 Data Compression Conference (DCC'10)*. IEEE Computer Society, 239–248.

S. Kreft and G. Navarro. 2013. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.* 483 (2013), 115–133.

V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. 2010. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.* 17, 3 (2010), 281–308.

G. Manzini. 2001. An analysis of the Burrows-Wheeler transform. *J. ACM* 48, 3 (2001), 407–430.

G. Navarro. 2012. Indexing highly repetitive collections. In *Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA'12)*, Lecture Notes in Computer Science, Vol. 7643. Springer, 274–279.

G. Navarro and V. Mäkinen. 2007. Compressed full-text indexes. *ACM Comput. Surv.* 39, 1 Article 2 (2007).

E. Ohlebusch and S. Gog. 2011. Lempel-Ziv factorization revisited. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11)*, Lecture Notes in Computer Science, Vol. 6661. Springer, 15–26.

D. Okanohara and K. Sadakane. 2008. An online algorithm for finding the longest previous factors. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, Lecture Notes in Computer Science, Vol. 5193. Springer, 696–707.

I. Pavlov. 2012. 7-zip. http://www.7-zip.org/. (2012).

J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. 2008. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE'08)*, Lecture Notes in Computer Science, Vol. 5280. Springer, 164–175.

F. Wu. 2009. Sequential file prefetching in Linux. In *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, Y. Wiseman and S. Jiang (Eds.). IGI Global, Chapter 11, 217–236.

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

Received April 2013; revised August 2013; accepted October 2014