Lempel-Ziv Factorization: Simple, Fast, Practical

Dominik Kempa†
Simon J. Puglisi†

Abstract
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 paper we describe simple and fast algorithms for computing the LZ77 factorization. These new methods consistently outperform all previous approaches in practice, use less memory, and still offer strong worst-case performance guarantees. A common feature of the new algorithms is their avoidance of the longest-common-prefix array, essential to nearly all prior art.

1 Introduction
For more than three decades the Lempel-Ziv (LZ77) factorization [33] 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 [27, 28] 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 [24, 13, 12]. Such data includes the new and rapidly growing genomic collections produced by high-throughput sequencing technology [9, 14, 25]; versioned collections of source code and multi-author documents, such as Wikipedia [32]; and web crawls [10]. Efficient index construction is stated as an open problem in both [24] and [13].

In a more traditional setting, compression of files using the 7zip tool [31], 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 [10, 16]. 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 [23]) 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.

Aside from compression and indexing, LZ77 factorization finds multifarious uses as an algorithmic tool for string processing, in particular for efficient detection of periodicities in strings [2, 8, 15, 20, 21, 22]. Periodicities in turn have diverse applications throughout computer science, in the fields of bioinformatics, data mining, and extremal combinatorics.

Our contribution. With the above applications in mind, in this paper we describe several efficient methods for computing the LZ77 factorization. Our aim was to develop fast, practical algorithms that operate in a memory range common with previous algorithms for the problem: about 6n to 13n bytes, for an input string of n symbols.

A common feature of our algorithms is their work is always related (though in different ways) to the number of factors in the resulting LZ77 factorization. This makes them particularly effective on highly repetitive inputs which have small factorizations, though we consistently outperform prior methods on all types of input, repetitive or not.

Two highlights are: an algorithm that uses 6n bytes of memory, and is 5-10 times faster than the previous fastest algorithm at that memory level; and an algorithm using 9n bytes which is faster than all other algorithms in the literature (usually by a factor of almost two). Finally, while our focus is on algorithms that are efficient in practice, the new algorithms also come with solid asymptotic guarantees on performance.

Previous work. A recent survey [1] outlines the many (mostly recent) algorithms for LZ77 factorization, nearly all of which make use of the suffix array (SA) and longest-common-prefix (LCP) array as intermediate data structures [26, 19]. The LZ77 factorization parses a string of length n into z ≤ n longest previous factors...
(we give a precise definition shortly). Almost all of the algorithms in the survey, and the ones since in [29], first compute the longest previous factor (LPF) for every position in the string, and then in a final step select just those involved in the LZ77 factorization. In many such algorithms, computing the “extra” LPF values seems unavoidable: the starting position of the \( j \)-th factor depends on the sum of the lengths of the \( j \)-1 factors prior to it, and so we cannot tell ahead of time which positions will be involved in the factorization.

Both [1] and [29] contain experimental evaluations of the various factorization algorithms described to date. We used the results from those papers to guide our experiments, in particular to select the best algorithms for comparison. Along the way we also noticed some anomalies in the performance of some algorithms, and we discuss this further in Section 5.

2 Preliminaries

Strings. Throughout we consider a string \( X = X[0..n] = X[0]X[1] \ldots X[n] \) of \( |X| = n + 1 \) symbols. The first \( n \) symbols of \( X \) are drawn from a constant ordered alphabet, of size \( \sigma \), and comprise the actual input. The final symbol \( X[n] \) is a special “end of string” symbol, \( \$ \), distinct from and lexicographically smaller than all the other characters in \( X \).

In order to account for the practical memory usage of our algorithms we assume \( \sigma \in \{0..255\} \) (corresponding to, say, an ASCII alphabet) and \( n < 2^{32} \); thus each symbol requires 1 byte of storage and the length of \( X \) and any pointers into it require 4 bytes each.

For \( i = 0, \ldots, n \) we write \( X[i..n] \) to denote the suffix of \( X \) of length \( n - i + 1 \), that is \( X[i..n] = X[i]X[i+1] \ldots X[n] \). We will often refer to suffix \( X[i..n] \) simply as “suffix \( i \)”. Similarly, we write \( X[0..i] \) to denote the prefix of \( X \) of length \( i + 1 \). We write \( X[i..j] \) to represent the substring \( X[i]X[i+1] \ldots X[j] \) of \( X \) that starts at position \( i \) and ends at position \( j \).

Suffix Arrays. We make use of several standard data structures built from \( X \). The first of these is the suffix array \( SA \) which is an array \( SA[0..n] \) containing a permutation of the integers \( 0..n \) such that \( X[SA[0..n]] < X[SA[1..n]] < \ldots < X[SA[n..n]] \). In other words, \( SA[j] = i \) iff \( X[i..n] \) is the \( j \)-th suffix of \( X \) in ascending lexicographical order. The inverse suffix array \( ISA \) is the inverse permutation of \( SA \), that is \( ISA[i] = j \) iff \( SA[j] = i \). Conceptually, \( ISA[i] \) tells us the position of suffix \( i \) in \( SA \).

The Burrows-Wheeler Transform, denoted \( BWT \), is a string \( BWT[0..n] \) is a permutation of \( X \) defined by \( SA \), such that \( BWT[i] = X[SA[i] - 1] \), except when \( SA[i] = 0 \), in which case \( BWT[i] = \$ \). None of our algorithms explicitly build the \( BWT \), but it is used implicitly in some places. We also make use of LF, the so-called last-to-first mapping. LF is usually defined in terms of \( BWT \), but it will be convenient for us to define it the following way: \( LF[i] = j \) iff \( SA[j] = SA[i] - 1 \), except when \( SA[i] = 0 \), in which case \( LF[i] = ISA[n] \).

Finally, let \( lcp(i, j) \) denote the length of the longest-common-prefix of suffix \( i \) and suffix \( j \). For example, in the string \( X = zzzzzzapzap \), \( lcp(1, 4) = 1 = |z| \), and \( lcp(4, 7) = 3 = |zap| \).

LZ77. The LZ77 factorization uses the concept of a longest previous factor (LPF). The LPF at position \( i \) in string \( X \) is a pair \((p_i, \ell_i)\) such that, \( p_i < i \), \( X[p_i..p_i+\ell_i-1] = X[i..i+\ell_i-1] \) and \( X[p_i+\ell_i] \neq X[i+\ell_i] \). In other words, \( X[i..i+\ell_i-1] \) is the longest prefix of \( X[i..n] \) which also occurs at some position \( p_i < i \) in \( X \). Note that if \( X[i] \) is the leftmost occurrence of a symbol in \( X \) then \( p_i \) does not exist. In this case we adopt the convention that \( p_i = X[i] \) and \( \ell_i = 0 \). Note also 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 \((p_i, \ell_i)\) (to represent the \( j \)-th phrase), and then the \((j+1)\)-th phrase starts at position \( i+\ell_i \), unless \( \ell_i = 0 \), in which case the next phrase starts at position \( i+1 \). We call a factor \((p_i, \ell_i)\) normal if it satisfies \( \ell_i > 0 \) and special otherwise.

The above description of LZ77 allows \( p_i+\ell_i > i \) and so \( X[i..i+\ell_i-1] \) and \( X[p_i..p_i+\ell_i-1] \) can overlap each other. This definition of LZ77 is sometimes called self-referential. The LZ77 parsing algorithms we describe can be adapted to produce non-self-referential parsing, or more exotic forms (e.g. [23, 24]), though we will assume the self-referential style throughout.

For the example string \( X = zzzzzzapzap \), the LZ77 factorization produces the pairs:

\((z, 0), (0, 4), (a, 0), (p, 0), (4, 3)\).

3 Speeding up a lightweight LZ77 algorithm

Our first contribution is a series of optimizations to a factorization algorithm due to Chen et al., called CPS2 [5]. The original algorithm has two interesting properties: firstly, it is unique among LZ77 factorization algorithms in that it avoids computation of the LCP array. For this reason it is one of the most space-efficient algorithms known, even considering algorithms that use compressed data structures [30, 29]. Secondly, it produces LZ77 in order, one factor at a time, avoiding computing longest previous factors for all \( n \) positions in the input first.
CPS2 makes use of $\text{SA}$, which it preprocesses for fast range minimum queries (RMQs) [11]. A range minimum query \text{rmq}(i,j)$ returns the position of the minimum value in $\text{SA}[i..j]$. Practical implementations of data structures supporting fast \text{rmq} are now well established.

To compute the factor starting at position $i$, CPS2 works in $\ell_i$ rounds. In round 0 it computes the range of the suffix array $\text{SA}[s_0..e_0]$ containing all the suffixes having $X[i]$ as a prefix. In a generic round $j$ CPS2 maintains the invariant that its active range, $\text{SA}[s_j..e_j]$, contains all the suffixes prefixed with $X[i..i+j]$. However, it also enforces, via \text{rmq}(s_j, e_j)$, that at least one suffix in $\text{SA}[s_j..e_j]$ begins at some position $p < i$. Of course $p$ 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.

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

### 3.1 Fast interval table computation

An important optimization to CPS2 which is not described in [5] 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 triple $(s_c, e_c, m_c)$, such that all suffixes prefixed with $c$ lie in $\text{SA}[s_c..e_c]$, and $m_c$ is the minimum value in $\text{SA}[s_c..e_c]$.

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

This leads us to our first optimization. Instead of scanning $\text{SA}$ and repeatedly accessing $X$ in $\text{SA}$ 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 $\text{SA}$ 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

---

**Figure 1:** Improvements to runtime for various optimizations to the CPS2 LZ77 factorization algorithm. We use typical repetitive (kernel) and non-repetitive (english) files (details in Section 5). Times are seconds per gigabyte. Ix is the fast interval computation optimization using $x$ levels of lookup tables. Ry is the small ranges scanning trick with $t = 2^y$. 

---

**Figure 2:** Parsing RMQ Intervals
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 $2^{16}$ entries. 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 extra table is negligible.

3.2 Scanning small ranges A further improvement to CPS2 makes use of the following easily proved lemma, due to Crochemore and Ilie [6] (and later restated by Ohlebusch and Gog [29]).

**Lemma 1.** Let $i$ be the starting position of a normal LZ77 factor and let $i_\text{<} \ (\text{resp. } i_\text{>} )$ be the first value smaller than $i$ to the left (resp. right) of $i$ in the SA.

If $\text{lcp}(i_\text{<}, i) > \text{lcp}(i_\text{>}, i)$, then $(p_i, \ell_i) = (i_\text{<}, \text{lcp}(i_\text{<}, i))$; otherwise, $(p_i, \ell_i) = (i_\text{>, \text{lcp}(i_\text{>, i))}).$

During the binary search phase of CPS2, when the size of the range drops below a predefined threshold $t$, we stop using binary search further and instead scan the range in $O(t)$ time to find $i_\text{<}$ and $i_\text{>}$.

We then compute $\text{lcp}(i_\text{<}, i)$ and $\text{lcp}(i_\text{>}, i)$, and depending on which is greater, output $i_\text{<}$ or $i_\text{>}$ as the LZ77 factor starting at $i$.

Setting $t = O(\log n)$ preserves the $O(n \log n)$ overall runtime of CPS2, but the scanning scheme only requires three cache misses and so should be faster than further binary searching, which even on small ranges can still attract two or more cache misses per round when accessing X to narrow the current range. In practice we found $t = 4096$ to give the best performance.

Our optimizations to CPS2 are summarized in Figure 1, 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.

4 Factorization and the Inverse Suffix Array

Our last improvement to CPS2 used Lemma 1 as a way to abandon further binary search steps in favour 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 ISA[$i$], and then search out in SA in either direction from that position, to locate $i_\text{<}$ and $i_\text{>}$.

**Algorithm LZ9**

1: $i \leftarrow 0$
2: while $i < n$ do
3: scan SA[ISA[$i$..n]] to find $i_\text{>}$
4: scan SA[0..ISA[$i$]] to find $i_\text{<}$
5: if $\text{lcp}(i_\text{<}, i) > \text{lcp}(i_\text{>, i})$ then
6: $(p_i, \ell_i) \leftarrow (i_\text{<}, \text{lcp}(i_\text{<}, i))$
7: else
8: $(p_i, \ell_i) \leftarrow (i_\text{>, \text{lcp}(i_\text{>, i))})$
9: output factor $(p_i, \ell_i)$
10: $i \leftarrow i + \ell_i$

**Figure 2:** The LZ9 algorithm, which uses SA, ISA, and X to compute the LZ factorization. For ease of presentation we assume both $i_\text{<}$ and $i_\text{>}$ exist for each factor. This will not always be the case (when, say, X[$i$] is the leftmost occurrence of a symbol in X) but such cases are easily handled.

The simplest implementation of this scheme is to store ISA explicitly, using $4n$ bytes, and to sequentially scan SA to find $i_\text{<}$ and $i_\text{>}$. We call this algorithm LZ9 — it uses $9n$ bytes in total for SA, ISA, and X. Pseudocode is given in Figure 2. 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 $i_\text{<}$ and $i_\text{>}$. 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 faster worst case bound, but the following string illustrates that things can indeed get quite bad for the algorithm. 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:

$$Y = 0^u 1N_0 01^u 1N_1 1 \ldots 0^u 1N_j 1 \ldots 0^u 1N_n 1.$$  

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, $k$, 2$k$, 3$k$, . . . where $k = 2 \log n + 3$.

Now consider the operation of LZ9n when factorizing Y. When a factor starts at a position $i = 0(\mod k)$ then the algorithm will scan SA left and right from position ISA[$i$]. Because the elements in this segment of SA are increasing, the scan left for $i_\text{<}$ will stop immediately, however the scan right from $i_\text{>}$ 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 $j = 0(\mod k)$, 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 laboured 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 $ISA[i]$ to find $i_<$ and $i_>$. This scanning can be avoided if we first preprocess $SA$ and build and data structure to answer next-smaller-value (NSV) and and previous-smaller-value (PSV) queries. We found the NSV/PSV data structure of Cánovas and Navarro [4] 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 tradeoff, namely: it requires $4n/b$ bytes and answers queries in $O(b + \log(n/b))$ time.

We call this version of LZ9 with an auxiliary NSV/PSV data structure ISA9. Setting $b = O(\log n)$ ensures ISA9 runs in $O(n + z \log n)$ time overall. In practice we found a higher value of $b$ led to faster runtimes, and allowed us to reduce space overheads to a negligible level. When using the NSV/PSV data structure to find $i_<$ and $i_>$, the runtime for ISA9 on string $Y$ above is reduced to a very respectable 4.9 seconds. For brevity from this points onwards we assume $b = O(\log n)$.

4.2 Reducing space requirements We now show how to reduce the space requirements of ISA9 by a more careful representation of ISA, which does not adversely affect runtime. A well known property of suffix arrays, and the Burrows-Wheeler transform, is $ISA[i − 1] = LF[ISA[i]]$. This property is the essence of the BWT inversion algorithm [3, 18] and the FM-index [28]. 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 non-sample value $i \neq 0(\mod k)$ can be recovered when needed by looking up the first sample larger than $i$, $j = ISA[(i/k) + 1]$, and then following the LF mapping $k − i \mod k$ times starting from $LF[j]$.

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.** LF can be implemented by answering rank queries on the BWT of the input string (see, e.g. [18]). In particular, $LF[i] = C[BWT[i]] + \text{rank}(i)$, where $C[c]$ is the total number of symbols less than symbol $c$ in the whole of $X$, and $\text{rank}(i)$ tells us the number of occurrences of symbol $BWT[i]$ before position $i$ in $BWT$. Data structures for supporting $\text{rank}$ are well studied, 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 $X$. 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 $BWT[j]$ before position $j$ in BWT. To answer $\text{rank}(i)$ we binary search over the starting positions of the runs, to locate the starting position of the run which $i$ falls in, say $j$. The answer to $\text{rank}(i)$ is then $\text{rank}(j)$, which is stored earlier with $j$, plus $j − i$, the number of occurrences of $BWT[i]$ between $i$ and $j$. This solution requires $8r$ bytes and answers $\text{rank}(i)$ in $O(\log r)$ time, and so factorizes in $O(n + zk \log r + z \log n)$ time overall.

**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 $LF[i]$ we first inspect $c = BWT[i] = X[SA[i] − 1]$ (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 $LF[i]$, and points us to a place in SA which must be within $b$ positions of the position we seek (i.e. the true $LF[i]$ value). Finally we scan $SA$ to the right of the approximate value of $LF[i]$ until we find the suffix with value $SA[i] − 1$. The position of this value is $ISA[SA[i] − 1]$ — which is our goal. This approach avoids scanning BWT (which we would like to avoid computing on-the-fly because of cache misses). At query time, scanning of $SA$ is fast, and causes no extra cache misses. We found this approach was almost as fast as rle-LF for repetitive data, but its space requirements are stable and tunable on all types of data. It uses $4n/b$ bytes on top of $SA$ and $X$, and factorizes in $O(n + zk \log(n/b) + z \log n)$ time. Setting $b = O(\log n)$ yields $O(n + zk \log n)$ complexity. In practice we set $k = O(1)$ and so expect the running time $O(n + z \log n)$.

We refer to the algorithm using rle-LF for rank queries as ISA6r, and the algorithm that uses sparse-LF instead as ISA8s. Figure 3 gives an overview of the performance of these algorithms relative to ISA9 on the
same files as used in Figure 1. For the ISAs algorithm we sampled ISA array at different rates to illustrate the space-time tradeoff. A version of ISAs with sampling rate set so that memory usage stays below 6n bytes in total is used in our experiments in Section 5 and is called ISA6s.

While on the non-repetitive file (english) the space-time curve smoothly drops down as the available memory increases, the time for kernel file actually increases around 6n 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. Sampling the ISA at a higher rate increases preprocessing time, but this is not repaid in the parsing phase.

5 Experiments

For testing we used the files listed in Table 1. All tests were conducted on a 3.30GHz Intel Xeon CPU with 8GB main memory and 8192K L2 Cache. Only a single thread of execution was used in all experiments. The machine had no other significant CPU tasks running. The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.2.0. The compiler was g++ (gcc version 4.6.3) executed with the -O3 -static -DNDEBUG options. The times given are the minima of three runs and were recorded with the standard C clock function. All data structures reside in main memory during computation.

To compute the suffix array we use Yuta Mori’s divsufsort algorithm and implementation (http://code.google.com/p/libdivsufsort/). In the algorithms that require the LCP array we compute it using our own implementation of the Φ algorithm [17], which is the fastest LCP array construction algorithm we know of. Φ has a memory peak of 13n bytes, which did not increase the peak memory for any algorithm that used it.

Experiments measured the time to compute the LZ factorization. Some algorithms, such as all those introduced in this paper, compute it directly, and others, as we noted earlier, must first compute all the LPF values. In the latter case we only include the time to compute the LPF values, as some of the implementations produce only the ℓ component of each LPF value, which is insufficient for full LZ factorization. Note that this slightly disadvantages our new algorithms.

The algorithms and their memory requirements are listed in Table 2. The experiments are summarized in Table 3 (runtimes) and Table 4 (memory usage). Implementations of our algorithms are available at http://www.cs.helsinki.fi/en/gsa/lz77. We have found the values b = 4096 and b = 64 to be a good compromise between space and time and use them in our experiments.

The proposed CPS2 optimizations significantly improve the runtime. A particularly big change for repetitive files can be attributed to fast interval computation. The parsing phase of CPS2 strongly benefits from long phrases (binary searching in a small range) hence the factorization of files with small z is very fast. It is therefore beneficial for total runtime if the preceding phase (computation of intervals) takes little time as well.

Our new ISA9 algorithm is consistently faster than all previous algorithms, and simultaneously use 4n bytes less space. The improvement in all cases is at least 28% (42% on average). For non-repetitive files the difference is even bigger (at least 40%). Interestingly ISA9 is always faster than LCP computation, for which we used the best available solution.

We also note a minor inconsistency with [29]: in our experiments algorithm OG is slower than algorithms based on LPF array for some files (in [29] it is always faster). This is because in [29] a slower LCP construction algorithm ([19]) was used (see the comparison in [17]) and the time to compute LCP dominates the total runtime of LPF-based algorithms.

Our ISA6s algorithm which uses ISA sampling is very competitive with ISA9 on repetitive files despite low memory usage. The explanation of this phenomena can be found in section 4.2. A very reasonable slowdown compared to ISA9 on non-repetitive files demonstrates the effectiveness of sparse-LF representation.

Lastly, note that although using the rle-LF representation (in place of sparse-LF) restricts the applicability of ISA6r to repetitive files, it allows the algorithm to outperform ISA6s and sometimes even beat ISA9.

6 Conclusions and Future Work

In this paper we have shown 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), 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.

All LZ77 factorization algorithms to date, including the ones in this paper, make use of the suffix array, and so require memory at least sufficient to store n integers. An important open problem, especially for text indexes based on LZ77, is to develop scalable factorization algorithms that completely avoid suffix
Table 1: Files used in the experiments. The files are from (S) the Pizza-Chili standard corpus (http://pizzachili.dcc.uchile.cl/texts.html) and (R) the Pizza-Chili repetitive corpus (http://pizzachili.dcc.uchile.cl/repcorpus.html). The repetitive corpus contains artificially generated sequences (A), files with several variants of the same data (R), and files created from standard corpus files by concatenating 100 copies of a 1MB prefix and mutating them randomly (PR). The values of \( n/r \) (average length of run in BWT) and \( n/z \) (average length of phrase in LZ factorization) are included as measures of repetitiveness.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space</th>
<th>Output</th>
<th>ISA s/r</th>
<th>LF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA9</td>
<td>9n+</td>
<td>LZ</td>
<td>1.0</td>
<td>n/a</td>
<td>Full ISA array</td>
</tr>
<tr>
<td>ISA6s</td>
<td>6n</td>
<td>LZ</td>
<td>0.2</td>
<td>sparse</td>
<td>ISA sampling, general</td>
</tr>
<tr>
<td>ISA6r</td>
<td>6n</td>
<td>LZ</td>
<td>0.125</td>
<td>rle</td>
<td>ISA sampling, specialized</td>
</tr>
<tr>
<td>CPS2I</td>
<td>6n</td>
<td>LZ</td>
<td></td>
<td></td>
<td>Improved version of [5]</td>
</tr>
<tr>
<td>CPS2</td>
<td>6n</td>
<td>LZ</td>
<td></td>
<td></td>
<td>Original algorithm from [5]</td>
</tr>
<tr>
<td>CI</td>
<td>13n</td>
<td>LZ</td>
<td></td>
<td></td>
<td>ComputeLPF from [6]</td>
</tr>
<tr>
<td>OG</td>
<td>13n+</td>
<td>LPF</td>
<td></td>
<td></td>
<td>Ultra-Fast algorithm from [5]</td>
</tr>
<tr>
<td>LPF1</td>
<td>13n</td>
<td>LPF</td>
<td></td>
<td></td>
<td>LPF-optimal in [7]</td>
</tr>
<tr>
<td>LPF2</td>
<td>13n+</td>
<td>LPF</td>
<td></td>
<td></td>
<td>LPF-online in [7]</td>
</tr>
</tbody>
</table>

Table 2: Algorithms and their space requirements (n = text length). Space requirements include the space for SA and text but exclude space for output, unless it is necessary during computation. ISA s/r - the fraction of ISA array that is stored. LF column gives the LF representation used. "+" in the space column marks that the algorithm requires some extra memory (stack for OG and LPF2 and PSV/NSV for ISA9), which in practice is negligible. Note that the space requirement of ISA6r holds only if the number of runs in the BWT satisfies $r \leq n/16$.

<table>
<thead>
<tr>
<th>Testfile</th>
<th>SA</th>
<th>LCP</th>
<th>ISA9</th>
<th>ISA6s</th>
<th>ISA6r</th>
<th>CPS2I</th>
<th>CPS2</th>
<th>CI</th>
<th>OG</th>
<th>LPF1</th>
<th>LPF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>dna</td>
<td>141.4</td>
<td>72.1</td>
<td>44.4</td>
<td>84.6</td>
<td>-</td>
<td>1141.4</td>
<td>1478.7</td>
<td>110.8</td>
<td>110.8</td>
<td>147.4</td>
<td>94.3</td>
</tr>
<tr>
<td>proteins</td>
<td>152.0</td>
<td>61.6</td>
<td>47.9</td>
<td>107.5</td>
<td>-</td>
<td>697.4</td>
<td>1128.1</td>
<td>112.3</td>
<td>122.9</td>
<td>83.2</td>
<td>79.7</td>
</tr>
<tr>
<td>english</td>
<td>132.5</td>
<td>61.8</td>
<td>43.4</td>
<td>81.7</td>
<td>-</td>
<td>658.8</td>
<td>1056.9</td>
<td>110.8</td>
<td>106.0</td>
<td>83.1</td>
<td>80.7</td>
</tr>
<tr>
<td>sources</td>
<td>88.5</td>
<td>45.1</td>
<td>33.8</td>
<td>58.8</td>
<td>-</td>
<td>356.3</td>
<td>611.4</td>
<td>106.0</td>
<td>106.0</td>
<td>68.5</td>
<td>64.0</td>
</tr>
<tr>
<td>dblp.xml</td>
<td>93.4</td>
<td>44.6</td>
<td>29.8</td>
<td>47.0</td>
<td>-</td>
<td>231.1</td>
<td>452.7</td>
<td>106.5</td>
<td>106.5</td>
<td>68.3</td>
<td>64.5</td>
</tr>
<tr>
<td>cere</td>
<td>131.7</td>
<td>62.9</td>
<td>33.4</td>
<td>30.3</td>
<td>27.4</td>
<td>148.4</td>
<td>251.4</td>
<td>121.7</td>
<td>56.8</td>
<td>98.3</td>
<td>94.6</td>
</tr>
<tr>
<td>coreutils</td>
<td>97.1</td>
<td>37.4</td>
<td>21.3</td>
<td>26.5</td>
<td>22.9</td>
<td>52.8</td>
<td>138.3</td>
<td>96.7</td>
<td>36.4</td>
<td>60.3</td>
<td>56.5</td>
</tr>
<tr>
<td>world_leaders</td>
<td>52.8</td>
<td>33.8</td>
<td>18.7</td>
<td>22.7</td>
<td>19.8</td>
<td>36.8</td>
<td>121.9</td>
<td>65.1</td>
<td>32.5</td>
<td>54.9</td>
<td>50.2</td>
</tr>
<tr>
<td>kernel</td>
<td>96.5</td>
<td>41.7</td>
<td>22.7</td>
<td>25.4</td>
<td>21.7</td>
<td>40.2</td>
<td>115.3</td>
<td>103.4</td>
<td>36.5</td>
<td>66.4</td>
<td>62.6</td>
</tr>
<tr>
<td>einstein.en</td>
<td>130.1</td>
<td>55.0</td>
<td>28.4</td>
<td>24.2</td>
<td>20.5</td>
<td>19.1</td>
<td>109.9</td>
<td>122.5</td>
<td>39.3</td>
<td>87.0</td>
<td>82.3</td>
</tr>
<tr>
<td>english.001.2</td>
<td>104.6</td>
<td>41.0</td>
<td>21.8</td>
<td>26.9</td>
<td>23.4</td>
<td>40.1</td>
<td>251.4</td>
<td>121.7</td>
<td>56.8</td>
<td>98.3</td>
<td>94.6</td>
</tr>
<tr>
<td>proteins.001.1</td>
<td>103.1</td>
<td>41.4</td>
<td>22.4</td>
<td>27.8</td>
<td>24.2</td>
<td>37.0</td>
<td>265.0</td>
<td>79.0</td>
<td>39.1</td>
<td>73.3</td>
<td>62.1</td>
</tr>
<tr>
<td>dblp.xml.00001.1</td>
<td>102.2</td>
<td>42.2</td>
<td>21.3</td>
<td>24.4</td>
<td>21.3</td>
<td>20.4</td>
<td>240.0</td>
<td>76.1</td>
<td>36.4</td>
<td>74.3</td>
<td>62.1</td>
</tr>
<tr>
<td>rs.13</td>
<td>254.2</td>
<td>42.3</td>
<td>20.9</td>
<td>25.6</td>
<td>22.8</td>
<td>12.1</td>
<td>110.3</td>
<td>88.3</td>
<td>29.2</td>
<td>62.4</td>
<td>61.6</td>
</tr>
<tr>
<td>tm29</td>
<td>269.4</td>
<td>42.5</td>
<td>21.1</td>
<td>26.4</td>
<td>23.3</td>
<td>12.3</td>
<td>152.1</td>
<td>93.9</td>
<td>38.2</td>
<td>84.0</td>
<td>61.1</td>
</tr>
</tbody>
</table>

Table 3: Times for computing LZ factorization. The times are seconds per gigabyte and do not include any reading from or writing to disk. The time to precompute the SA (which is a prerequisite for all algorithms) is not included in the runtime. We also separately present the time to compute the LCP array but, unlike SA, it is included in the total runtime for algorithms that use it (LPF1 and LPF2).

<table>
<thead>
<tr>
<th>Testfile</th>
<th>ISA9</th>
<th>ISA6s</th>
<th>ISA6r</th>
<th>CPS2I</th>
<th>CPS2</th>
<th>CI</th>
<th>OG</th>
<th>LPF1</th>
<th>LPF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>dna</td>
<td>9.03</td>
<td>5.91</td>
<td>-</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>proteins</td>
<td>9.03</td>
<td>5.93</td>
<td>-</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>12.01</td>
<td>13.01</td>
</tr>
<tr>
<td>english</td>
<td>9.03</td>
<td>6.03</td>
<td>-</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>sources</td>
<td>9.03</td>
<td>6.03</td>
<td>-</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>dblp.xml</td>
<td>9.03</td>
<td>5.97</td>
<td>-</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>cere</td>
<td>9.02</td>
<td>5.90</td>
<td>5.94</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>coreutils</td>
<td>9.03</td>
<td>6.04</td>
<td>5.98</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>world_leaders</td>
<td>9.05</td>
<td>5.99</td>
<td>5.85</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>kernel</td>
<td>9.02</td>
<td>6.00</td>
<td>5.78</td>
<td>5.83</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>einstein.en</td>
<td>9.02</td>
<td>5.98</td>
<td>5.65</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>english.001.2</td>
<td>9.03</td>
<td>5.98</td>
<td>5.82</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>proteins.001.1</td>
<td>9.03</td>
<td>5.93</td>
<td>5.82</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>dblp.xml.00001.1</td>
<td>9.03</td>
<td>5.97</td>
<td>5.68</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>rs.13</td>
<td>9.03</td>
<td>5.90</td>
<td>5.65</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
<tr>
<td>tm29</td>
<td>9.02</td>
<td>5.88</td>
<td>5.65</td>
<td>5.84</td>
<td>5.83</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
<td>13.01</td>
</tr>
</tbody>
</table>

Table 4: Peak memory usage in bytes per character for all algorithms.
sorting the entire input.

Acknowledgments. Thanks go to Golnaz Badkobeh, Maxime Crochemore, Julia Kärkkäinen, and Travis Gagie for inspiring discussions on the topic of LZ77 factorization; to Simon Gog and German Tischler for sharing source code, and for explicating details of their experiments; and to the anonymous referees whose comments materially improved this paper.

References


