A Lempel-Ziv-style Compression Method for Repetitive Texts

Markus Mauer, Timo Beller, and Enno Ohlebusch

Institute of Theoretical Computer Science
Ulm University
89069 Ulm, Germany
{Markus.Mauer,Timo.Beller,Enno.Ohlebusch}@uni-ulm.de

Abstract. In this paper, we present a compression algorithm that is based on finding repetitions in the file to be compressed. Our approach is a variant of longest-first-substitution compression that uses the suffix array and the LCP-array to find and encode long recurring substrings. We will show that our algorithm achieves very good compression ratios for repetitive texts.

Keywords: lossless data compression, longest-first-substitution compression, repetitive texts, suffix array

1 Introduction

The dictionary-based LZ-algorithms devised by Lempel and Ziv [14,21,22] are an important class of lossless compression algorithms. One can distinguish between *on-line* algorithms (in which the dictionary is dynamically built from the prefix of the text seen so far) and *off-line* algorithms (in which the dictionary is constructed from the whole text). The original LZ77-algorithm uses a window of size w and the dictionary consists of all substrings that start within the last w scanned positions of the text. In classical implementations, the LZ77-algorithm parses greedily, i.e., if $S[1..i-1]$ has already been scanned, then the next factor is the *longest* prefix of $S[i..n]$ that is in the dictionary and starts within $S[1..i-1]$. If the next factor has length ℓ and starts at position $j \leq i-1$ in S, then the LZ77-algorithm encodes the triple (d, ℓ, c), where $d = i - j < w$ is the offset and $c = S[i+\ell]$ is the character following the factor; it then continues parsing $S[i+\ell+1..n]$. If the window consists of all positions scanned so far (we will call this algorithm LZ77-compression without window), the offset d can be very large, so one should select the rightmost copy of a factor to keep d small (see [9] for an algorithm that does this with the help of the suffix tree of S). As pointed out in [16], the LZ77 compression algorithm without window that encodes the absolute position j at which the next factors starts (instead of the offset d), should be called LZ76 compression [14]. The greedy algorithm without window is optimal with respect to the number of factors, and it can be implemented in such a way that it uses only linear time and space [5] (the result of Crochemore and Ilie has been improved by many authors; see [10] and the references therein). If one encodes the factors by variable-length codes, then the greedy algorithm is in general not bit-optimal, i.e., it is not optimal in terms of the number of bits output by the compression algorithm; see [9]. Ferragina et al. [9, Theorem 5.4] use a linear-time algorithm for the single-source shortest path problem on a weighted DAG to obtain a bit-optimal algorithm for the LZ77-compression-scheme.

In this paper, we present an off-line compression algorithm that is different from the LZ-algorithms described above in that it does not try to parse the text in a

Markus Mauer, Timo Beller, Enno Ohlebusch: A Lempel-Ziv-style Compression Method for Repetitive Texts, pp. 96–107.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

missmississippimissedinmississippi$

missmississippimissedin~~mississippi~~$

missmiss~~issi~~ppimissedin~~mississippi~~$

miss~~miss~~~~issi~~ppi~~miss~~edin~~mississippi~~$

Figure 1. Consider the string S = `missmississippimissedinmississippi$`. Our compression algorithm first detects the repeat mississippi and encodes the wavy underlined occurrence (repeat of type 2). Then, it detects the periodicity `ississi` with period-length 3 and encodes the wavy underlined occurrence of `issi` (repeat of type 1). Finally, it detects the three repetitions of `miss` and encodes the wavy underlined occurrences (this repeat gets the identifier 3). In the resulting string S' = `miss##ppi#edin#$`, every occurrence of # stands for a factor and the vector $F = 3132$ contains the types (from left to right) of these factors. The factor of type 3 is $(1, 4)$, the factor of type 1 is $(3, 4)$, and the factor of type 2 is $(5, 11)$. That is, the list of factors (from left to right) is $[(1, 4), (3, 4), (5, 11)]$; see Section 4 for details.

left-to-right scan into phrases. By contrast, it identifies long repetitions in advance (prior to the compression) and then tries to greedily compress these repetitions (first the longest, then the second longest, etc.). This strategy is called longest-first-substitution. If the repetition is a periodicity (called repeat of type 1) or if it occurs only twice (called repeat of type 2), then it is stored in a list of factors and the type is stored in a vector F. However, if it occurs more than twice, then a factor is stored only for the second occurrence whereas the other occurrences are encoded by a unique identifier, which is stored in F. All occurrences of these repeats (except for the first occurrence) are replaced with a special symbol # in S, yielding a string S'. The three components (S', F, and the list of factors) are then compressed separately; see Fig. 1 for an example and Section 4 for details. Our software is available at `https://www.uni-ulm.de/in/theo/research/seqana/`

2 Related Work for DNA-sequences

When writing this paper, we were unaware of the work of Rivals et al. [20]. It turned out that they used the same basic idea as our algorithm, but they restrict their algorithm to DNA-sequences. Moreover, the details differ substantially. For example, they do not take periodicities (overlapping repeats) into account. Furthermore, they encode *one* sequence consisting of substrings, factors, and indices to the dictionary. By contrast, we separate the three types. This separation makes the three parts amenable to different compression techniques, i.e., one can apply every lossless data compression algorithm to S' and F (while the factors, which are pairs of position and length, are encoded separately; see Section 4 for details). More related work can be found in [2,15,19,7].

The problem of compressing a collection of genomes from individuals of the same species with respect to a reference genome has been extensively studied. The relative Lempel-Ziv (RLZ) algorithm devised by Kuruppu et al. [12,13] is a popular algorithm for this special case, especially when fast random access is required. The RLZ-algorithm was subsequently improved by Deorowicz and Grabowski [6], by Ferrada et al. [8], and by Cox et al. [4]. In contrast to these algorithms, our algorithm does not rely on a reference sequence: it can be applied to every (repetitive) text. On the one hand, our algorithm provides better compression than these algorithms; on the other hand, our approach does not support random access.

3 Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the sentinel character \$. In the following, S is a string of length n on Σ having the sentinel character at the end (and nowhere else). For $1 \le i \le n$, $S[i]$ denotes the *character at position* i in S. For $i \le j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 2 for an example. We refer to the overview article [18] for suffix array construction algorithms (some of which have linear run-time).

The suffix array is closely related to the Burrows and Wheeler transform [3] $\mathsf{BWT}[1..n]$, which is defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 2.

The suffix array SA is often enhanced with the LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA; see Fig. 2. Formally, the LCP-array is an array so that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n+1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \le i \le n$, where $\mathsf{lcp}(u,v)$ denotes the longest common prefix between two strings u and v. Kasai et al. [11] showed that the LCP-array can be computed in linear time from the suffix array and its inverse. Abouelhoda et al. [1] introduced the concept of lcp-intervals; see Fig. 2. An interval $[i..j]$, where $1 \le i < j \le n$, in the LCP-array is called an *lcp-interval of lcp-value* ℓ (denoted by ℓ-$[i..j]$) if

1. $\mathsf{LCP}[i] < \ell$,
2. $\mathsf{LCP}[k] \ge \ell$ for all k with $i+1 \le k \le j$,
3. $\mathsf{LCP}[k] = \ell$ for at least one k with $i+1 \le k \le j$,
4. $\mathsf{LCP}[j+1] < \ell$.

In Fig. 2, for example, the interval [9..14] is an lcp-interval of lcp-value 3.

Abouelhoda et al. [1] presented an algorithm that enumerates all lcp-intervals in a bottom-up fashion. Moreover, they showed that there is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the suffix tree of S (we assume a basic knowledge of suffix trees). Consequently, there are at most $n-1$ lcp-intervals for a string of length n.

If ℓ-$[i..j]$ is an lcp-interval, then the ℓ-length prefix ω of $S_{\mathsf{SA}[k]}$, where $i \le k \le j$, is a *repeat* because the number of occurrences of ω in S is $j-i+1 \ge 2$. If $\{\mathsf{BWT}[k] \mid i \le k \le j\}$ is not a singleton set, then ω is a *maximal* repeat. In this case, the lcp-interval $[i..j]$ is also called *maximal*. For example, the lcp-interval 3-[9..14] in Fig. 2 is maximal.

If a non-empty string ω can be written as $\omega = u^k v$, where $k \ge 2$ and v is a proper prefix of u, then it is called a *periodicity* with *period-length* $|u|$.

4 The Compression Algorithm

In this section, we first describe the basic approach of our compression algorithm (implementation details will be discussed later). The key idea is to classify repeats

i	SA	LCP	BWT	$S_{\mathsf{SA}[i]}$
1	35	-1	i	$
2	21	0	e	dinmississippi$
3	20	0	s	edinmississippi$
4	34	0	p	i$
5	15	1	p	imissedinmississippi$
6	22	1	d	inmississippi$
7	31	1	s	ippi$
8	12	4	s	ippimissedinmississippi$
9	17	1	m	issedinmississippi$
10	28	3	s	issippi$
11	9	7	s	issippimissedinmississippi$
12	25	4	m	ississippi$
13	6	10	m	ississippimissedinmississippi$
14	2	3	m	issmississippimissedinmississippi$
15	16	0	i	missedinmississippi$
16	24	4	n	mississippi$
17	5	11	s	mississippimissedinmississippi$
18	1	4	$	missmississippimissedinmississippi$
19	23	0	i	nmississippi$
20	33	0	p	pi$
21	14	2	p	pimissedinmississippi$
22	32	1	i	ppi$
23	13	3	i	ppimissedinmississippi$
24	19	0	s	sedinmississippi$
25	30	1	s	sippi$
26	11	5	s	sippimissedinmississippi$
27	27	2	s	sissippi$
28	8	8	s	sissippimissedinmississippi$
29	4	1	s	smississippimissedinmississippi$
30	18	1	i	ssedinmississippi$
31	29	2	i	ssippi$
32	10	6	i	ssippimissedinmississippi$
33	26	3	i	ssissippi$
34	7	9	i	ssissippimissedinmississippi$
35	3	2	i	ssmississippimissedinmississippi$

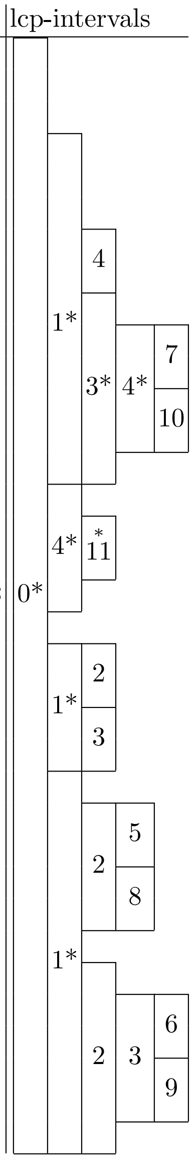

Figure 2. Suffix array, LCP-array, and the Burrows-Wheeler transform BWT of the string $S =$ `missmississippimissedinmississippi$`. If an lcp-interval is maximal, then its lcp-value is marked with an asterisk.

into different types, which are treated differently. These types are represent by the variable id. An overlapping repeat (a periodicity) is said to be of type 1 ($id = 1$). In this case, the part without the first period is encoded by a reference to the beginning of the repeat (and the period-length). If there are two non-overlapping occurrences of a repeat, then it is of type 2 ($id = 2$). In this case, the second occurrence is encoded as in LZ76 compression by a reference to the first occurrence. Finally, if there are more than two non-overlapping occurrences of a repeat, then each of the occurrences—except for the first one—is encoded by a unique identifier $id > 2$. It is important to note that for each such identifier, only one reference is stored.

4.1 The basic approach

Our basic compression algorithm works as follows:

1. Compute all maximal lcp-intervals of the LCP-array.[1] This can e.g. be done in linear time by a bottom-up traversal of the LCP-interval tree; see [17, Algorithm 5.15].
2. Store all maximal lcp-intervals with an lcp-value $\ell \geq \ell_{min}$ in a priority queue Q (the priority of an lcp-interval is its lcp-value ℓ; the higher the better), where ℓ_{min} is a threshold.[2]
3. Initialize a bit-vector B of size $n = |S|$ with zeros, initialize an empty list *list* and set $id \leftarrow 3$. In the following, a substring $S[k..m]$ of S is said to be marked if and only if $B[k..m]$ contains at least a one. An unmarked substring can be subject to compression, but a marked substring can not.
4. While Q is not empty, remove the lcp-interval ℓ-$[lb..rb]$ with the currently highest priority from Q and do:
   (a) Compute a subset *candidates* of $\{\mathsf{SA}[i] \mid lb \leq i \leq rb\}$ so that for each $k \in$ *candidates* the substring $S[k..k+\ell-1]$ is unmarked. This is the case if and only if $B[k] = 0$ and $B[k+\ell-1] = 0$.[3] During the computation, determine $occ_1 = \min\{\mathsf{SA}[i] \mid lb \leq i \leq rb\}$ and $occ_2 = \min(candidates \setminus \{occ_1\})$; note that occ_1 may or may not be a member of the set *candidates*.
   (b) Sort *candidates* and store the result in an array *sorted_candidates* of size $|candidates|$. Set $cur \leftarrow occ_1$ and $i \leftarrow 1$ and determine the subset $accepted \subseteq candidates$ as follows:
   while $i \leq |candidates|$ do
   – if $cur + \ell - 1 < sorted\_candidates[i]$, then add $sorted\_candidates[i]$ to *accepted* and set $cur \leftarrow sorted\_candidates[i]$; set $i \leftarrow i+1$
   Note that occ_1 is not a member of the set *accepted*. As a result, for each pair $j, k \in accepted$ with $j < k$ we have $j + \ell - 1 < k$ (i.e., the corresponding substrings are non-overlapping). If the set *accepted* is non-empty, then $occ_1 + \ell - 1 < occ_3$ where $occ_3 = \min(accepted)$. Note that occ_3 may or may not be equal to occ_2.
   (c) If $occ_1 \in candidates$ and $t = occ_2 - occ_1 < \ell$, then $S[occ_1..occ_1+\ell-1]$ and $S[occ_2..occ_2+\ell-1]$ overlap and $occ_2 \notin accepted$. In this case, add $(occ_1+t, 1, t, \ell)$ to *list* and set $B[occ_1+t..occ_1+t+\ell-1] \leftarrow [1..1]$ unless $S[occ_2..occ_2+\ell-1]$ overlaps with $S[occ_3..occ_3+\ell-1]$ (i.e., $occ_3 \neq \perp$ and $occ_3 - occ_2 < \ell$).[4]
   (d) Let $size = |accepted|$ be the size of the set *accepted*. If $size > 0$ and $\ell \geq a/size + b$, where a and b are constants that will be explained in Section 4.2, then proceed with (4e); otherwise take the next interval from Q. In essence, the restriction on ℓ ensures that the compression of the factors (whose starting positions are in the set *accepted*) is worthwhile.
   (e) If $size = 1$, where $size = |accepted|$, then add $(occ_3, 2, occ_1, \ell)$ to *list* and set $B[occ_3..occ_3+\ell-1] \leftarrow [1..1]$.

[1] In a previous implementation we used all lcp-intervals, but this resulted in unacceptable run-times.
[2] In our implementation, ℓ_{min} equals the constant a, which will be explained in Section 4.2.
[3] If $B[k+1..k+\ell-2]$ would contain a one while $B[k] = 0$ and $B[k+\ell-1] = 0$, then a substring of $S[k+1..k+\ell-2]$ would have been subject to compression. Consequently, an lcp-interval of lcp-value $< \ell$ must have been chosen before the current lcp-interval of lcp-value ℓ. This, however, is impossible because lcp-intervals are chosen greedily (first the longest, then the second longest, etc.).
[4] $\perp$ denotes an undefined value.

(f) If $size > 1$, then add (occ_3, id, occ_1, ℓ) to *list* and set $B[occ_3..occ_3 + \ell - 1] \leftarrow [1..1]$. Furthermore, for each $k \in accepted \setminus \{occ_3\}$, add $(k, id, \perp, \ell)$ to *list* and set $B[k..k + \ell - 1] \leftarrow [1..1]$. Finally, increment *id* by one.

We note that (a part of) the first occurrence of a repeat is subject to compression only if it overlaps with the second occurrence; see Case (4c). Cases (4e) – (4f) deal with non-overlapping occurrences of the repeat under consideration. If there is just one unmarked occurrence apart from the first occurrence, then Case (4e) applies, whereas Cases (4f) applies if there is more than one unmarked occurrence apart from the first occurrence.

5. Let *sorted* be the list obtained by sorting the elements in *list* according to their first components (in increasing order).
6. Initialize an empty vector F, an empty list *factors*, an empty string S', and set $p \leftarrow 1$.
7. While *sorted* is not empty, remove its first element (k, id, occ, ℓ) and do:
   (a) If $id = 1$ or $id = 2$, then insert *id* at the back of vector F and insert (occ, ℓ) at the back of list *factors*.
   (b) If $id > 2$, then insert *id* at the back of vector F. Furthermore, if $occ \neq \perp$, insert (occ, ℓ) at the back of list *factors*.
   (c) Concatenate S' with $S[p..k-1]\#$ and set $p \leftarrow k$. (In essence, S' is obtained from S by replacing each factor $S[k..k + \ell - 1]$ with $\#$.)
8. Compress the list *factors*, the vector F, and the string S' separately.

As an example, consider $S =$ `missmississippimissedinmississippi$`. The corresponding suffix- and LCP-arrays are shown in Fig. 2. For $\ell_{min} = 4$, the priority queue looks as follows: $Q = [(11, [16..17]), (4, [11..13]), (4, [15..18])]$. In the first iteration of the while-loop (4), case (4e) applies for the lcp-interval $(11, [16..17])$, where $occ_1 = 5$ and $occ_3 = occ_2 = 24$. Thus, the quadruple $(24, 2, 5, 11)$ is added to *list* and all the bits in $B[24..34]$ are set to 1. In the second iteration of the while-loop (4), the sets $candidates = \{6, 9\}$ and $accepted = \emptyset$ are computed in steps (4a) and (4b), respectively. Furthermore, we have $occ_1 = 6$, $occ_2 = 9$, and $occ_3 = \perp$. It is not difficult to see that case (4c) applies with $t = 3$, so that the quadruple $(6 + 3, 1, 3, 4)$ is added to *list* and all the bits in $B[9..12]$ are set to 1. In the final iteration of the while-loop (4), we have $candidates = \{1, 5, 16\}$ and $accepted = \{5, 16\}$ as well as $occ_1 = 1$ and $occ_2 = occ_3 = 5$. Now case (4f) applies, so first $(5, 3, 1, 4)$ is added to *list* and all the bits in $B[5..8]$ are set to 1 and then $(16, 3, \perp, 4)$ is added to *list* and the bits in $B[16..19]$ are set to 1. It follows as a consequence that $sorted = [(5, 3, 1, 4), (9, 1, 3, 4), (16, 3, \perp, 4), (24, 2, 5, 11)]$. Furthermore, we have $factors = [(1, 4), (3, 4), (5, 11)]$, $F = 3132$, and $S' =$ `miss##ppi#edin#$`.

Let us analyse the worst-case time complexity of the compression algorithm. The first and the third step can be done in $O(n)$ time, while the second step requires $O(n \log n)$ time. As explained in Section 3, there are at most $n - 1$ lcp-intervals (note that there are strings, e.g. the string $S = a^{n-1}\$$, for which each of its lcp-intervals is maximal). It follows as a consequence that the while-loop in Case (4) has at most $n - 1$ iterations. Clearly, each iteration deals with an lcp-interval $[lb..rb]$ of size $rb - lb + 1 < n$. For each i with $lb \leq i \leq rb$, it can be tested in constant time whether $\mathsf{SA}[i]$ belongs to the set *candidates* or not; see Case (4a). Moreover, the set *candidates* can be sorted in linear time in Case (4b) provided we use counting sort (in practice, however, a comparison based sorting algorithm will outperform counting sort). It is quite obvious that each of the Cases (4c) – (4f) takes at most $O(n)$ time. Consequently, the while-loop in Case (4) runs in $O(n^2)$ time. It is not difficult to see that $O(n^2)$ is also an upper bound for each of the remaining steps. In summary, the compression algorithm has a worst-case time complexity of $O(n^2)$.

4.2 Implementations details

First of all, we will explain how a factor (a pair consisting of a position and a length) is encoded in our approach.

- Consider two consecutive factors (occ_1, ℓ_1) and (occ_2, ℓ_2) in the list *factors*. If $\mathit{diff} = (occ_2 - occ_1)$ satisfies $|\mathit{diff}| < 2^x$, where x is a fixed natural number, then we use a Rice code plus a sign bit to encode diff. Otherwise, the position occ_2 is encoded with $\lceil \log_2 n \rceil$ bits.
- If the length ℓ of a factor satisfies $\ell < 2^y$, where y is a fixed natural number, then we use a Rice code to encode it. Otherwise it is encoded with $\lceil \log_2 \ell_{max} \rceil$ bits, where ℓ_{max} is the maximum entry in the LCP-array.

Our software contains subroutines that compute the best values of x and y for the input file (prior to the compression of the factors).

Next, we will explain the constants a and b in step (4d) of the basic compression algorithm. To this end, let S_{bits} (S'_{bits}) be the average number of bits needed to encode one symbol in S (S') with a fixed compression algorithm X. In the following, we assume that S_{bits} and S'_{bits} are approximately the same and from now on we denote the average number of bits needed to encode one symbol by k. Similarly, let F_{bits} be the average number of bits needed to encode one symbol in F and let $Factor_{bits}$ denote the average number of bits needed to encode one factor. Recall that $size = |accepted|$ denotes the size of the set *accepted*. On the one hand, if ℓ is the length of the repeat to be compressed, then we would need approximately $(size+1) * \ell * k$ bits to encode all the occurrences with the compression algorithm X ($size + 1$ many occurrences of the length ℓ repeat have to be taken into account). On the other hand, in our approach we would need

- $\ell * k$ bits to encode the first occurrence of the repeat plus $size * k$ bits to encode the extra # symbols with the compression algorithm X,
- $size * F_{bits}$ bits to encode the occurrences of the identifier (type) of the repeat in F, and
- $Factor_{bits}$ bits to encode the factor.

Our compression scheme is worthwhile whenever the following inequality holds:

$$\begin{aligned}
(size + 1) * \ell * k &\geq \ell * k + size * k + size * F_{bits} + Factor_{bits} \\
\Leftrightarrow size * \ell * k &\geq size * k + size * F_{bits} + Factor_{bits} \\
\Leftrightarrow \ell &\geq 1 + \frac{F_{bits}}{k} + \frac{Factor_{bits}}{k * size} \\
\Leftrightarrow \ell &\geq \frac{a}{size} + b
\end{aligned}$$

where $a = \frac{Factor_{bits}}{k}$ and $b = 1 + \frac{F_{bits}}{k}$. In our implementation, we use the parameters $a = 30$ and $b = 80$ as default values because these values gave the best compression ratios in our experiments.

We would like to point out two more facts to the reader, which are important in practice:

- To limit the number of lcp-intervals, our algorithm uses only maximal lcp-intervals. However, if the string $S = a^{n-1}\$$ is input, then every lcp-interval is maximal and the run-time slows down significantly. Our algorithm deals with such cases at the very beginning (i.e., when all lcp-intervals are enumerated): it detects a periodicity and its period length (in case of $S = a^{n-1}\$$, it detects that a^{n-1} is a periodicity of period length 1) and does not add lcp-intervals that belong to the same periodicity to the queue Q.
- We use the special symbol # to denote the places of factors in S'. However, if S already contains #, then the decompression algorithm will not work properly. To avoid this, whenever # appears in S, a 0 is added to the type vector F at the appropriate place.

5 The Decompression Algorithm

The basic decompression algorithm decompresses the list *factors*, the vector F, and the string S' separately. It then restores the original string S from S' with the help of a variable *cur* (points to the current factor in *factors*), a variable *pos* (current position in S), and an array $table[1..max]$ (entries initialized with $\perp$), where *max* is the maximum number (identifier) in F, as follows. If the current symbol c in S' is not #, then it is simply copied, i.e., $S[pos] \leftarrow c$ and $pos \leftarrow pos + 1$. If $c = \#$, say c is the k-th occurrence of #, then the algorithm uses a case distinction on the type $F[k]$.

- If $F[k] = 0$, then $S[pos] \leftarrow \#$. Set $pos \leftarrow pos + 1$.
- If $F[k] = 1$, then $S[pos..pos + \ell - 1] \leftarrow S[pos - t..pos - t + \ell - 1]$, where $(t, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow pos - t + \ell$.
- If $F[k] = 2$, then $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow occ + \ell$.
- If $F[k] > 2$, then
  - if $table[k] = \perp$, then $table[k] \leftarrow factors[cur]$ and $cur \leftarrow cur + 1$

  set $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow table[k]$, and $pos \leftarrow occ + \ell$.

6 An Advanced Algorithm

In addition to the basic version of our algorithm (as described in the previous two sections), we implemented a second advanced version. The advanced version takes substrings of strings from the set $candidate \setminus accepted$ into account; see e.g. [19] for a similar approach. More importantly, the advanced version uses a different labeling scheme for the F vector that is obtained by replacing step (7) in the basic compression algorithm as follows:

Initialize the variable $newId$ with the value 3.
Let *max* be the maximum value of all identifiers in *sorted*.
Initialize an array *table* of size *max*.
While *sorted* is not empty, remove its first element (k, id, occ, ℓ) and do:

1. If $id = 1$ or $id = 2$, then insert id at the back of vector F and insert (occ, ℓ) at the back of list *factors*. If $id = 2$, then increment $newId$ by 1.

2. If $id > 2$ and $occ \neq \perp$, i.e., id occurs for the first time, then insert 2 at the back of vector F, insert (occ, ℓ) at the back of list *factors*, set $table[id] \leftarrow newId$, and increment $newId$ by one.
3. If $id > 2$ and $occ = \perp$, then insert $table[id]$ at the back of vector F.
4. Concatenate S' with $S[p..k-1]\#$ and set $p \leftarrow k$. (In essence, S' is obtained from S by replacing each factor $S[k..k+\ell-1]$ with #.)

Thus, even if a repeat has several occurrences, each second occurrence is encoded by a 2 in F. Since this results in many occurrences of 2 in F, the compression ratio for F is better than before. Of course, the decompression algorithm must be able to cope with the new F vector. To this end, the following modification of the basic decompression algorithm is necessary:

Initialize the variable $newId$ with the value 3.
Initialize an array *table* of size $count + 2$, where *count* is the number occurrences of the value 2 in the new F vector.

- If $F[k] = 0$, then ... (the same as before).
- If $F[k] = 1$, then ... (the same as before).
- If $F[k] = 2$, then $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow occ + \ell$. Moreover, set $table[newId] \leftarrow factors[cur]$ and increment $newId$ by one.
- If $F[k] > 2$, then set $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow table[k]$, and $pos \leftarrow occ + \ell$.

7 Experimental Results

To test our compression method, we conducted several experiments using different state of the art compression methods. We compared the sizes of the compressed files as well as the compression and decompression times. As dataset we used four repetitive files from the Pizza & Chili corpus[5] and two from the `RLZAP` dataset.[6]

In our experiments, we used the lossless data compression methods `bzip2` Version 1.0.6, `gzip` Version 1.6, `xz`[7] Version 5.1.0alpha with the compression preset level -9 (the primary compression algorithm of `xz` is currently LZMA2), `zpaq`[8] Version 7.15 with the compression level -m5 (i.e. using a high order context mixing model), and `RLZAP`.[9] We compared these methods with both the basic and the advanced version of our compression method. Since `xz` and `zpaq` provide the best compression ratios, we used them to compress the three components S', F, and *factors*.

Table 1 shows the file sizes after compression. Both the basic and the advanced version of our method outperform the other methods in five of six cases. The poor compression ratios of gzip can be attributed to the fact that the files contain occurrences of repeats that are far apart (i.e., their distance is greater than the window size). A similar statement holds for bzip2 because it compresses blocks rather than the whole text (the default block size is 900k). In Table 2, we show exemplarily the sizes of the three components S', F, and *factors* for the file para before and after the

[5] `http://pizzachili.dcc.uchile.cl/index.html`
[6] `http://acube.di.unipi.it/rlzap-dataset/`
[7] `http://tukaani.org/xz/`
[8] `https://github.com/zpaq/zpaq`
[9] `https://github.com/farruggia/rlzap`

	world_leaders	**einstein.de**	**influenza**
Filesize	46.968	92.758	154.809
bzip2	3.261	4.010	10.197
gzip	8.288	28.797	10.637
xz	0.607	0.099	**2.068**
zpaq	0.519	0.130	2.639
basic+xz	0.552	0.096	2.203
basic+zpaq	0.476	0.540	10.051
advanced+xz	0.518	0.092	2.132
advanced+zpaq	**0.453**	**0.084**	2.491
RLZAP	-	-	-
	kernel	**e_coli**	**para**
Filesize	257.962	164.899	429.266
bzip2	56.074	44.465	112.236
gzip	69.396	46.136	116.073
xz	2.087	6.289	6.256
zpaq	3.652	30.386	87.787
basic+xz	2.037	6.158	5.709
basic+zpaq	4.088	14.458	17.821
advanced+xz	2.019	**6.073**	**5.567**
advanced+zpaq	**1.573**	8.600	8.925
RLZAP	-	22.556	11.634

Table 1. Sizes after compression in MB (10^6 bytes).

	S'		**F**		**factors**	
basic+xz	35.769	4.612	0.515	0.123	1.065	0.974
basic+zpaq	35.769	16.090	0.515	0.106	1.065	1.625
advanced+xz	34.126	4.507	0.592	0.011	1.137	1.049
advanced+zpaq	34.126	7.878	0.592	0.009	1.137	1.039

Table 2. Sizes of the different components in MB (10^6 bytes) for the file para, before and after the final compression step (8) of our algorithm. Note that in this case `zpaq` compresses S' much worse than `xz`. However, this varies from file to file.

final compression step (8). As already mentioned, our advanced method achieves a smaller size for S' by finding additional factors. While this gives a larger F vector as well as a larger *factors* list, the different naming scheme for the F vector results in better final compression ratios. Moreover, we would like to point out that S' is a lot smaller than the original string S.

The compression and decompression times are listed in Table 3. While `bzip2` and `gzip` have the fastest compression times, their compression ratios are rather poor. Apart from these two, `xz` tends to give the best compression times, but our method is not far behind. Note that `xz` gives the best decompression times for all files, but our method is also very fast if `xz` is used in the final compression step. However, if we use `zpaq` as a final compression method, both compression and decompression times are significantly higher (but the combination of our algorithm with `zpaq` is always faster than `zpaq` itself).

All in all, the results show that our method can keep up with the state of the art compression algorithms both in terms of compression ratios and in terms of compression/decompression time. Furthermore, several improvements of our method seem possible. For example, the greedy strategy could be based on a sophisticated rating of factors (instead of the simple rating based on the lengths of factors) or there may

	world_leaders		einstein.de		influenza	
bzip2	0m02s	0.604s	0m09s	1.531s	**0m17s**	2.725s
gzip	**0m2s**	0.211s	**0m05s**	0.516s	0m22s	0.504s
xz	0m08s	**0.101s**	0m10s	**0.141s**	0m36s	**0.339s**
zpaq	1m34s	93.692s	2m34s	154.737s	6m23s	381.018s
basic+xz	0m30s	0.154s	0m35s	0.165s	2m22s	0.609s
basic+zpaq	0m37s	7.254s	0m36s	1.067s	2m52s	40.123s
advanced+xz	0m33s	0.206s	0m36s	0.288s	2m43s	0.764s
advanced+zpaq	0m38s	6.211s	0m36s	1.070s	3m15s	39.745s
	kernel		**e_coli**		**para**	
bzip2	0m18s	6.415s	**0m13s**	4.956s	**0m32s**	12.788s
gzip	**0m13s**	1.479s	1m36s	0.858s	4m01s	2.218s
xz	1m18s	**0.484s**	2m08s	**0.558s**	5m46s	**0.980s**
zpaq	6m03s	365.241s	6m53s	425.736s	18m33s	1140.174s
basic+xz	1m45s	0.625s	1m55s	0.996s	13m13s	1.681s
basic+zpaq	2m06s	25.458s	2m55s	95.598s	14m17s	102.614s
advanced+xz	1m43s	0.942s	1m54s	1.127s	14m56s	1.870s
advanced+zpaq	2m04s	24.564s	2m54s	93.652s	16m02s	98.342s

Table 3. Compression/decompression times for the files from our dataset. The compression times are given in minutes and seconds. For the decompression times, we use seconds.

be other ways of building the F vector. Finally, our method is quite flexible because it can be combined with other compression methods in the final compression step.

Acknowledgements: We thank the anonymous reviewers for their helpful comments.

References

1. M. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*. Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Compression of biological sequences by greedy off-line textual substitution*, in Proc. 10th Data Compression Conference, IEEE Computer Society, 2000, pp. 143–152.
3. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, Research Report 124, Digital Systems Research Center, 1994.
4. A. Cox, A. Farruggia, T. Gagie, S. Puglisi, and J. Sirén: *RLZAP: relative Lempel-Ziv with adaptive pointers*, in Proc. 23rd International Symposium on String Processing and Information Retrieval, vol. 9954 of Lecture Notes in Computer Science, Springer, 2016, pp. 1–14.
5. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications*. Information Processing Letters, 106(2) 2008, pp. 75–80.
6. S. Deorowicz and S. Grabowski: *Robust relative compression of genomes with random access*. Bioinformatics, 27(21) 2011, pp. 2979–2986.
7. P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane: *Compression with the tudocomp framework*. CoRR, abs/1702.07577 2017.
8. H. Ferrada, T. Gagie, S. Gog, and S. Puglisi: *Relative Lempel-Ziv with constant-time random access*, in Proc. 21st International Symposium on String Processing and Information Retrieval, vol. 8799 of Lecture Notes in Computer Science, Springer, 2014, pp. 13–17.
9. P. Ferragina, I. Nitto, and R. Venturini: *On the bit-complexity of Lempel-Ziv compression*. SIAM Journal on Computing, 42(4) 2013, pp. 1521–1541.
10. J. Kärkkäinen, D. Kempa, and S. Puglisi: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in Proc. 24th Annual Symposium on Combinatorial Pattern Matching, vol. 7922 of Lecture Notes in Computer Science, Springer, 2013, pp. 189–200.

11. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Annual Symposium on Combinatorial Pattern Matching, vol. 2089 of Lecture Notes in Computer Science, Springer, 2001, pp. 181–192.
12. S. Kuruppu, S. Puglisi, and J. Zobel: *Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval*, in Proc. 17th International Symposium on String Processing and Information Retrieval, vol. 6393 of Lecture Notes in Computer Science, Springer, 2010, pp. 201–206.
13. S. Kuruppu, S. Puglisi, and J. Zobel: *Optimized relative Lempel-Ziv compression of genomes*, in Proc 34th Australasian Computer Science Conference, Australian Computer Society, 2011, pp. 91–98.
14. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 21(1) 1976, pp. 75–81.
15. R. Nakamura, S. Inenaga, H. Bannai, T. Funamoto, M. Takeda, and A. Shinohara: *Linear-time text compression by longest-first substitution.* Algorithms, 2(4) 2009, pp. 1429–1448.
16. G. Navarro: *Compact Data Structures*, Cambridge University Press, Cambridge, 2016.
17. E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag, Bremen, 2013.
18. S. Puglisi, W. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Computing Surveys, 39(2) 2007, p. Article 4.
19. S. Ristov and D. Korencic: *Using static suffix array in dynamic application: Case of text compression by longest first substitution.* Information Processing Letters, 115(2) 2015, pp. 175–181.
20. E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange: *A guaranteed compression scheme for repetitive DNA sequences*, in Proc. 6th Data Compression Conference, IEEE Computer Society, 1996, p. 453.
21. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.
22. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.