Bidirectional Text Compression in External Memory

Patrick Dinklage
Technische Universität Dortmund, Department of Computer Science, Germany
patrick.dinklage@tu-dortmund.de

Jonas Ellert
Technische Universität Dortmund, Department of Computer Science, Germany
jonas.ellert@tu-dortmund.de

Johannes Fischer
Technische Universität Dortmund, Department of Computer Science, Germany
johannes.fischer@cs.tu-dortmund.de

Dominik Köppl
Kyushu University, Fukuoka, Japan Society for Promotion of Science, Japan
https://dkppl.de/
dominik.koeppl@inf.kyushu-u.ac.jp

Manuel Penschuck
Goethe University Frankfurt, Department of Computer Science, Germany
mpenschuck@ae.cs.uni-frankfurt.de

Abstract

Bidirectional compression algorithms work by substituting repeated substrings by references that, unlike in the famous LZ77-scheme, can point to either direction. We present such an algorithm that is particularly suited for an external memory implementation. We evaluate it experimentally on large data sets of size up to 128 GiB (using only 16 GiB of RAM) and show that it is significantly faster than all known LZ77 compressors, while producing a roughly similar number of factors. We also introduce an external memory decompressor for texts compressed with any uni- or bidirectional compression scheme.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases text compression, bidirectional parsing, text decompression, external algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.41

Related Version A full version of the paper is available at https://arxiv.org/abs/1907.03235.

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number JP18F18120
Manuel Penschuck: Deutsche Forschungsgemeinschaft (DFG) grants ME 2088/3-2, ME 2088/4-2

1 Introduction

Text compression is a fundamental task when storing massive data sets. Most practical text compressors such as gzip, bzip2, 7zip, etc., scan a text file with a sliding window, replacing repetitive occurrences within this window. Although this approach is memory and time efficient [3, 29], two occurrences of the same substring are neglected if their distance is longer than the sliding window. More advanced solutions [12, 13, 9, 19, to mention only a few examples] drop the idea of a sliding window, thereby finding also repetitions that are far apart in the text. These so-called *LZ77*-algorithms have a better compression ratio in practice [8, Sect. 6]. In recent years, these algorithms have also been transformed to the *external memory* (EM) model [17, 21, 2].

27th Annual European Symposium on Algorithms (ESA 2019).
Editors: Michael A. Bender, Ola Svensson, and Grzegorz Herman; Article No. 41; pp. 41:1–41:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this article, we present a modification of *LZ77*, called *plcpcomp*, which is based on the bidirectional compression scheme *lcpcomp* of [6], but is better suited for an efficient external memory implementation due to its memory access patterns. We can compute this scheme by scanning the text and two auxiliary arrays stored in EM (one of them being the *permuted longest common prefix array*, hence the acronym plcp). We underline the performance of our algorithm with evaluations showing that it is faster than any known *LZ77* compressor for massive non-highly repetitive data sets. We also present the first external decompressor for files that are compressed with a bidirectional scheme.

1.1 Related Work

Our work is the first to join the fields of bidirectional and external memory compression.

1.1.1 Bidirectional Schemes

First considerations started with [29] who also coined this notation. [11] proved that finding the *optimal* bidirectional parsing, i.e., a bidirectional parsing with the lowest number of factors, is NP-complete. [6] were the first to present a greedy algorithm for producing a bidirectional parsing called *lcpcomp*, which performs well in practice, but comes with no theoretical performance guarantees on its size. [25] combined the techniques for *lcpcomp* [6] and the longest-first grammar compression [26] in a compression algorithm running in $\mathcal{O}(n^2)$ time, which was subsequently improved to $\mathcal{O}(n \lg n)$ time by [27]. Recently, [10] showed an upper bound of $z = \mathcal{O}(b \lg(n/b))$ and a lower bound of $z = \Omega(b \lg n)$ for some specific strings, where b and z denote the minimal number of factors in an optimal *bidirectional* parsing and in an optimal *unidirectional* parsing, respectively. This implies that bidirectional parsing can be exponentially better than unidirectional parsing. They also proposed a bidirectional parsing based on the Burrows-Wheeler transform (BWT). [22] introduced so-called *string attractors*, showed that a bidirectional scheme is a string attractor and that every string attractor can be represented with a bidirectional scheme. Last but not least, the bidirectional scheme of [28] guarantees to produce at most as many factors as *LZ77*, but has the disadvantage of a super-quadratic running time.

1.1.2 EM Compression Algorithms

Yanovsky [30] presented a compressor called *ReCoil* that is specialized on large DNA datasets. Ferragina et al. [7] gave a construction algorithm of the Burrows-Wheeler transform in EM. For *LZ77* compression, [17] devised two algorithms called *EM-LZscan* and *EM-LPF*. The former performs well on highly-repetitive data, but gets outperformed easily by *EM-LPF* on other kinds of datasets. The *LZ77* compressed files can be decompressed with an algorithm due to [2], which also works in general for all files that have been compressed by a unidirectional scheme. Finally, [21] presented an EM algorithm for computing the *LZ-End* scheme [23], a variant of *LZ77*.

1.2 Preliminaries

Model of computation. We use the commonly accepted EM model by Aggarwal and Vitter [1]. It features two memory types, namely fast internal memory (IM) which may hold up to M data words, and slow EM of unbounded size. The measure of the performance of an algorithm is the number of input and output operations (I/Os) required, where each I/O transfers a block of B consecutive words between memory levels. Reading or writing

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	a	b	a	b	b	a	b	a	b	a	b	b	a	b	b	a	a	b	a	b	a	$
SA	22	21	16	19	17	6	1	8	13	3	10	20	15	18	5	7	12	2	9	14	4	11
ISA	7	18	10	21	15	6	16	8	19	11	22	17	9	20	13	3	5	14	4	12	2	1
Φ	6	12	13	14	18	17	5	1	2	3	4	7	8	9	20	21	19	15	16	10	22	11
LCP	0	0	1	1	3	5	4	7	2	4	5	0	2	2	4	5	3	5	6	1	3	4
$PLCP$	4	5	4	3	4	5	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0

Figure 1 Suffix array, its inverse, Φ, LCP array, and PLCP array of our running example string T.

n contiguous words from or to disk requires $\text{scan}(n) = \Theta(n/B)$ I/Os. Sorting n contiguous words requires $\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os. For realistic values of n, B, and M, we stipulate that $\text{scan}(n) < \text{sort}(n) \ll n$.

Text. Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$ for a natural number n. The alphabet Σ induces the *lexicographic order* $\prec$ on the set of strings Σ^*. Let $|T|$ denote the length of a string $T \in \Sigma^*$. We write $T[j]$ for the j-th character of T, where $1 \leq j \leq n$. Given $T \in \Sigma^*$ consists of the concatenation $T = UVW$ for $U, V, W \in \Sigma^*$, we call U, V, and W a *prefix*, a *substring*, and a *suffix* of T, respectively. Given that the substring V starts at the i-th and ends at the j-th position of T, we also write $V = T[i\,..\,j]$ and $W = T[j+1\,..]$. In the following, we take an element $T \in \Sigma^*$ with $|T| = n$, and call it *text*. We stipulate that T ends with a sentinel $T[n] = \$ \notin \Sigma$ that is lexicographically smaller than every character of Σ.

Text Data Structures. Let SA denote the *suffix array* [24] of T. The entry $SA[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[SA[i]\,..] \prec T[SA[i+1]\,..]$ for all integers i with $1 \leq i \leq n-1$. Let ISA of T be the inverse of SA, i.e., $ISA[SA[i]] = i$ for every i with $1 \leq i \leq n$. The *Burrows-Wheeler transform (BWT)* [4] of T is the string BWT with $BWT[i] = T[n]$ if $SA[i] = 1$ and $BWT[i] = T[SA[i]-1]$ otherwise, for every i with $1 \leq i \leq n$. The *LCP array* is an array with the property that $LCP[i]$ is the length of the longest common prefix (LCP) of $T[SA[i]\,..]$ and $T[SA[i-1]\,..]$ for $i = 2, \ldots, n$. For convenience, we stipulate that $LCP[1] := 0$. The array Φ is defined as $\Phi[i] := SA[ISA[i]-1]$, and $\Phi[i] := n$ in case that $ISA[i] = 1$. The *PLCP array* $PLCP$ stores the entries of LCP in text order, i.e., $PLCP[SA[i]] = LCP[i]$. Figure 1 illustrates the introduced data structures.

Idea for Using PLCP for Compression. Given a suffix $T[i\,..]$ starting at text position i, $PLCP[i]$ is the length of the longest common prefix of this suffix and the suffix $T[\Phi[i]\,..]$, which is its lexicographical predecessor among all suffixes of T. The longest common prefix of these two suffixes $T[i\,..]$ and $T[\Phi[i]\,..]$ is $T[i\,..\,i + PLCP[i] - 1]$. The longest string among all these longest common prefixes (for each i with $1 \leq i \leq n$) is one of the longest re-occurring substrings in the text. Finding this longest re-occurring substring with $PLCP$ and Φ is the core idea of our compression algorithm. This algorithm produces a bidirectional scheme, which is defined as follows.

2 Compression Scheme

A *bidirectional scheme* [29] is defined by a factorization $F_1 \cdots F_b = T$ of a text T. A factor F_x is either a *referencing* factor or a *literal* factor. A referencing factor F_x is associated with a pair (src, ℓ) such that F_x and $T[src\,..\,src + \ell - 1]$ are two different but possibly overlapping occurrences of the substring F_x in T. The pair (src, ℓ) and the text position src are called

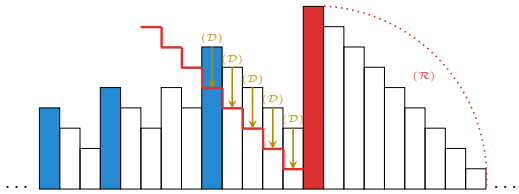


Figure 2 Visualization of Rules ($\mathcal{D}$) and ($\mathcal{R}$) being applied. Bars represent *PLCP* values.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	a	b	a	b	b	a	b	a	b	a	b	b	a	b	b	a	a	b	a	b	a	$
PLCP	4	5	4	3	4	5	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0
$PLCP^1$	4	5	4	3	3	2	1	0	0	0	0	0	0	0	2	1	3	2	1	0	0	0
$PLCP^2$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	2	1	3	2	1	0	0	0
$PLCP^4$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0
$PLCP^3$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3 Step-by-step execution of the *plcpcomp* compression scheme (see Section 2) on $T =$ `ababbababababbaababa$`. We overwrite values of *PLCP* according to Rules ($\mathcal{D}$) and ($\mathcal{R}$). Each row $PLCP^i$ shows *PLCP* after creating the i-th referencing factor starting at a position whose *PLCP* entry is surrounded by a box. Changed entries according to Rules ($\mathcal{D}$) and ($\mathcal{R}$) are underlined.

reference and *referred position*, respectively. A factorization is *cycle-free*, i.e., references are not allowed to have cyclic dependencies. A factorization is called ξ-*restricted* for an integer $\xi \geq 2$ if each referencing factor F_x is at least ξ characters long (i.e., $\ell \geq \xi$).

A *unidirectional scheme* is a special case of a bidirectional scheme, with the restriction that the referred position of a referencing factor F_x must be smaller than the starting position of F_x. The most prominent example of a unidirectional scheme is the *LZ77* factorization, whose factorization is usually designed to be 2-restricted.

2.1 Coding

A bidirectional scheme codes the factors by substituting referencing factors with their associated references while keeping literal factors as strings. By doing so, the coding is a list whose x-th element is either a string (corresponding to a literal factor) or a reference representing the x-th factor ($1 \leq x \leq b$), which is referencing.

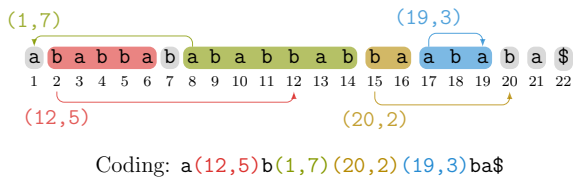
The *plcpcomp* scheme and its predecessor, the *lcpcomp* scheme [6], are bidirectional schemes. Both schemes are greedy, as they create a referencing factor equal to the longest re-occurring substring of the text that is not yet part of a factor. They differ in the selection of such a substring in case that there are multiple candidates with the same length. The *plcpcomp* scheme can be computed with a rewritable *PLCP* array and the following instructions:

1. Compute the set of *candidate positions* $\mathcal{C} := \{i \mid PLCP[i] \geq PLCP[j]$ for all text positions $j\}$.
2. Let dst be the leftmost position of all candidate positions $\mathcal{C}$. Terminate if $PLCP[dst] < \xi$.
3. Create a referencing factor by replacing $T[dst \mathinner{..} dst + PLCP[dst] - 1]$ with the reference $(\Phi[dst], PLCP[dst])$
4. Apply the following rules to ensure that we do not create overlapping factors (cf. Figure 2):

($\mathcal{D}$) Decrease $PLCP[j] \leftarrow \min(PLCP[j], dst - j)$ for every $j \in [dst - PLCP[dst], dst)$.

($\mathcal{R}$) Remove the factored positions by setting $PLCP[dst + k] \leftarrow 0$ for every $k \in [0, PLCP[dst])$.

5. Recurse with the modified *PLCP*.



dst	$src = \Phi[dst]$	length
8	1	7
2	12	5
17	19	3
15	20	2

Figure 4 Coding of *plcpcomp* with $\xi = 2$. The factorization described in Figure 3 computes four referencing factors, listed in the table on the right. These factors are coded by their references. The factorization with *PLCP* in Figure 3 already determines the starting position and the lengths of all referencing factors (columns "*dst*" and "length" in the table). The referred positions are obtained using Φ (column "*src*"). The figure on the left illustrates factors as boxed substrings and the references as arrows from the starting positions of referencing factors to their respective referred positions.

An application of the above instructions on our running example is given in Figure 3. The coding is visualized in Figure 4. There and in the following figures, we fix $\xi := 2$.

2.2 Comparison to *lcpcomp*

The difference to *lcpcomp* [6] is that we fix *dst* to be the *leftmost* of all candidate positions in $\mathcal{C}$. [6] presented an algorithm computing the *lcpcomp* scheme in $\mathcal{O}(n \lg n)$ time with a heap storing the candidate positions ranked by their *PLCP* values. We can adapt this algorithm to compute the *plcpcomp* scheme by altering the order of the heap to rank the candidate positions first by their *PLCP* values (maximal *PLCP* values first) and second (in case of equal *PLCP* values) by their values themselves (minimal text positions first).

Since *lcpcomp* is cycle-free [6, Lemma 4] regardless of the selection of $dst \in \mathcal{C}$, we conclude that *plcpcomp* is also cycle-free, i.e., the substitution of substrings by references is reversible.

3 Computing the Factorization without Random Access

In this section, we present an algorithm for computing the *plcpcomp* scheme, which linearly scans *PLCP* without changing its contents. Instead of maintaining a heap storing all text positions ranked by their *PLCP* values, we compute the factorization by scanning the text sequentially from left to right. Although the algorithm will produce the *plcpcomp* factorization, it does not compute it in the order explained previously (starting with the longest factor). Instead, it first determines a subset of those substrings that define a referencing factor according to the *plcpcomp* scheme. The starting positions of these substrings have a *PLCP* value that is relatively large compared to their neighboring positions. We call those starting positions *peaks*.

Formally, we call a text position *dst* a *peak* if $PLCP[dst] \geq \xi$ and one of the following conditions holds:

1. $dst = 1$,
2. $PLCP[dst-1] < PLCP[dst]$,[1] or
3. there is a referencing factor ending at $dst - 1$.

A peak *dst* is called *interesting* if there is no text position j with $dst \in (j, j+PLCP[j])$ and $PLCP[j] \geq PLCP[dst]$. An interesting peak *dst* is called *maximal* if there is no interesting peak j with $j \in (dst, dst + PLCP[dst])$.

[1] A subset of the so-called *irreducible PLCP* entries [20, Lemma 4] have this property.

Algorithm 1 Computation of *plcpcomp* factors.

```
1  L ← ∅                                                          // Step 1a
2  for dst = 1 to n do                                            // Step 1b
3      if dst is a maximal peak then                              // Step 2
4          create a referencing factor replacing T[dst .. dst + PLCP[dst] − 1]   // Step 3
5          apply Rule (D) to the peaks in L
6          while L contains maximal peaks do
7              j ← rightmost maximal peak in L
8              create referencing factor replacing T[j .. j + PLCP[j] − 1]
9              apply Rules (D) and (R) to the peaks in L
10             remove those elements of L that are no longer interesting peaks
11         dst ← dst + PLCP[dst]
12     if dst is an interesting peak then
13         L ← L ∪ {dst}
```

Given an interesting peak dst, there is no text position j with $PLCP[j] \geq PLCP[dst]$ that becomes the starting position of a referencing factor containing $T[dst]$ (such that $PLCP[dst]$ cannot be removed according to Rule ($\mathcal{R}$)). Given a maximal peak dst, there is additionally no text position j with $PLCP[j] > PLCP[dst]$ for which we apply Rule ($\mathcal{D}$) on $PLCP[dst]$ after factorizing $T[j \mathinner{..} j + PLCP[j] - 1]$. Informally, we can determine whether a peak is interesting by looking at the $PLCP$ values *before* this peak, whereas we need to also look *ahead* for determining whether a peak is maximal. Given that there is at least one $PLCP$ entry with a value of at least ξ, we can find a maximal peak, since the leftmost position $\min\{i \in [1 \mathinner{..} n] \mid PLCP[i] \geq PLCP[j]$ for all j with $1 \leq j \leq n\}$ among all positions with the highest $PLCP$ value is a maximal peak. The following lemma states that we can always factorize the leftmost maximal peak, regardless of whether the text has even higher peaks.

▶ **Lemma 1.** *If the text position dst is a maximal peak, then $T[dst \mathinner{..} dst + PLCP[dst] - 1]$ is a referencing factor.*

Our preliminary algorithm consists of the following steps:

1. Scan $PLCP$ for the leftmost maximal peak dst.
2. Terminate if no such peak exists.
3. Create the referencing factor $T[dst \mathinner{..} dst + PLCP[dst] - 1]$.
4. Apply Rules ($\mathcal{R}$) and ($\mathcal{D}$).
5. Interpret $T[1 \mathinner{..} dst - 1]$ and $T[dst + PLCP[dst] \mathinner{..} n]$ as two independent strings and recurse on each of them individually.

This algorithm produces the *plcpcomp* scheme, because

- $T[dst \mathinner{..} dst + PLCP[dst] - 1]$ is a referencing factor for each selected leftmost maximal peak dst according to Lemma 1, and
- the part $T[1 \mathinner{..} dst - 1]$ can be factorized independently from how $T[dst + PLCP[dst] \mathinner{..}]$ is factorized, and vice versa. That is because, having already $T[dst \mathinner{..} dst + PLCP[dst] - 1]$ factorized, we can no longer create a factor that covers a text position in the range $[dst \mathinner{..} dst + PLCP[dst] - 1]$.

Hence, we can factorize $T[1 \mathinner{..} dst - 1]$ without considering the factorization of the rest of the text to produce the correct *plcpcomp* scheme. Figure 5 illustrates the computation of the *plcpcomp* factorization with this algorithm.

However, as the algorithm overwrites entries of $PLCP$, it is not yet satisfying. A rewritable $PLCP$ array would have to be kept in RAM, costing us $n \lg n$ bits of space if we require constant time read and write access. Instead of keeping $PLCP[1 \mathinner{..} dst - 1]$ in RAM, we now

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	a	b	a	b	b	a	b	a	b	a	b	b	a	b	b	a	a	b	a	b	a	$
$PLCP$	4	5	4	3	4	5	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0
$PLCP^1$	1	0	0	0	0	0	5	7	6	5	4	3	2	1	2	1	3	2	1	0	0	0
$PLCP^1$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	2	1	3	2	1	0	0	0
$PLCP^2$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	3	2	1	0	0	0
$PLCP^3$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5 Step-by-step execution of our *plcpcomp* algorithm on $T =$ `ababbababbabbaababa$`. While the instructions of the scheme (cf. Section 2) always replace the factor starting at a position with the maximal *PLCP* value (cf. Figure 3), our algorithm described in Section 3 creates a factor at the leftmost maximal peak. Our algorithm computes the same factorization as described in the *plcpcomp* scheme, but in different order.

show that it suffices to manage only the *PLCP* values of the *interesting peaks*. For that, we enhance the search of the leftmost maximal peak by replacing the first step of the algorithm by the following instructions:

1a. Create an empty list of peaks L.

1b. Scan T from left to right until a maximal peak *dst* is found. While doing so, insert all visited interesting peaks into L.

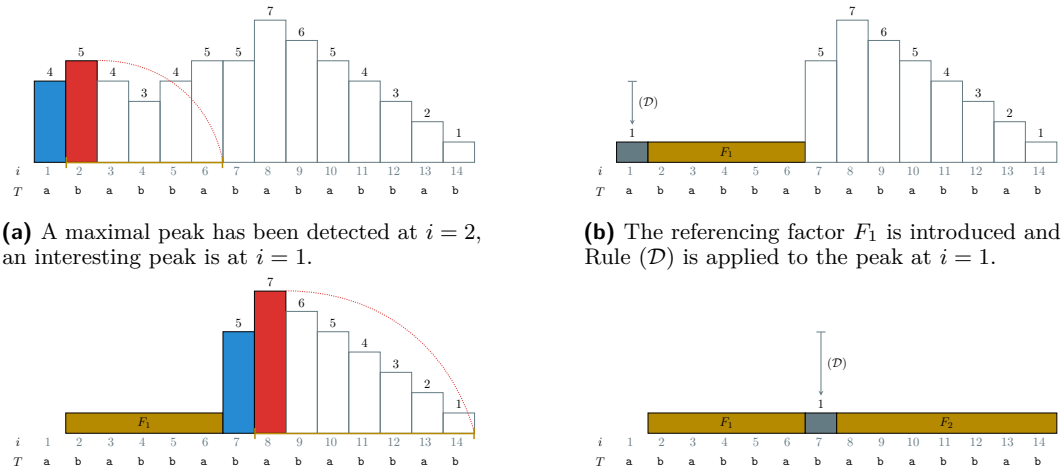
Another alternation is that we apply Step 4 only to the peaks stored in L. There, we scan L from right to left while applying Rule ($\mathcal{D}$) and removing all elements that are no longer interesting peaks. The modified algorithm is sketched as pseudo code in Algo. 1.

▶ **Example 2.** Figure 6 illustrates Algo. 1 on the prefix $T[1\,..\,14] =$ `ababbababbab` of our running example in three steps. The peaks at positions 1 and 2 are interesting. Since the peak at position 2 is the highest interesting peak, it is the maximal peak, which is detected after scanning $PLCP[1\,..\,6]$ (Figure 6a). In the second step (Figure 6b), the referencing factor F_1 is introduced, which starts at this maximal peak. As a consequence, Rule ($\mathcal{D}$) is applied to the only peak stored in L, the one at position 1. However, because the *PLCP* value 1 is below the threshold $\xi = 2$, the peak at position 1 is removed from L. Since L is then empty, we proceed with the next scan for a maximal peak starting from position 7. By definition, the peak at position 7 becomes interesting. The next maximal peak is detected at position 8 (Figure 6c). The factor F_2 (Figure 6d) is introduced, and Rule ($\mathcal{D}$) is applied to the peak at position 7. Its *PLCP* value drops below our threshold and thus it is removed from L. Finally, the prefix $T[1\,..\,14]$ has been processed.

In Algo. 1, we omit all other peaks that are not stored in L when applying Rules ($\mathcal{D}$) and ($\mathcal{R}$)). Thus, it suffices to maintain the *PLCP* value of each peak in L in an extra list instead of maintaining a complete rewritable *PLCP* array. In the following, we prove why this omission still produces the correct factorization (Lemma 5). For that, we show that we can produce the *plcpcomp* factors contained in $T[1\,..\,dst + PLCP[dst] - 1]$ only with the *PLCP* values of the peaks stored in L (first recursive call). We start with the following property of L:

▶ **Lemma 3.** *The positions stored in L are in strictly ascending order with respect to their LCP values.*

Next, we examine the result of creating the referencing factor $T[dst\,..\,dst + PLCP[dst] - 1]$ starting at the maximal peak *dst*. After creating this factor, the *PLCP* values of peaks near *dst* can be decreased. However, this causes at most one new peak as can be seen by the following lemma.



(a) A maximal peak has been detected at $i = 2$, an interesting peak is at $i = 1$.

(b) The referencing factor F_1 is introduced and Rule $(\mathcal{D})$ is applied to the peak at $i = 1$.

(c) A maximal peak has been detected at $i = 8$, an interesting peak is at $i = 7$.

(d) The referencing factor F_2 is introduced and Rule $(\mathcal{D})$ is applied to the peak at $i = 7$.

Figure 6 Execution of our algorithm of Section 3 computing the *plcpcomp* compression scheme on $T =$ `ababbababbabbaababa$`. Due to limited space, we only illustrate the processing of the prefix $T[1\,..\,14]$ in three steps (explained in Example 2). The vertical bars represent the *PLCP* array, with the corresponding values written above, in text order from left ($i = 1$) to right ($i = 14$). The shaded vertical bars represent the (current) *PLCP* value of an interesting peak. Horizontal bars represent (referencing) factors. In (b), the factor F_1, starting at position 2, is displayed as the maximal peak being *tipped over* to the right.

▶ **Lemma 4.** *Applying Rules $(\mathcal{D})$ and $(\mathcal{R})$ after creating a referencing factor F_x does not cause new peaks, with the only possible exception of the position succeeding the end of F_x.*

Since Rule $(\mathcal{D})$ decreases at most the values of $PLCP[dst - PLCP[dst]\,..\,dst - 1]$, the highest peak dst' in $PLCP[1\,..\,dst - 1]$ is an interesting peak that is either

- in the interval $[dst - PLCP[dst]\,..\,dst - 1]$, or,
- in the case that all interesting peaks in $[dst - PLCP[dst]\,..\,dst - 1]$ are no longer interesting after decreasing their *PLCP* values, the rightmost peak preceding $dst - PLCP[dst]$ (whose *PLCP* value is equal to the *PLCP* value of the last peak removed from L in Step 4).

We can locate dst' while applying Rule $(\mathcal{D})$ as a result of creating the factor starting at dst. After locating dst', we apply the following steps recursively:

1. Substitute $T[dst'\,..\,PLCP[dst'] - 1]$ with a reference, because it is the highest peak in $T[1\,..\,dst - 1]$.
2. If $dst'' := dst' + PLCP[dst']$ with $PLCP[dst''] \geq \xi$ was not a peak, then dst'' becomes an interesting peak. In this case, substitute dst' with dst'' in L to preserve the order in L. Otherwise, remove dst' from L.
3. Split L into two sub-lists:
   - one containing text positions of the range $[1\,..\,dst' - 1]$, and
   - the other containing text positions of the range $[dst' + PLCP[dst']\,..\,dst - 1]$.
4. Recurse on each of the two sub-lists, i.e., find the highest peak in each sub-list and substitute it.

This recursion is more efficient than the while-loop described in Lines 6 to 10 of Algo. 1.

▶ **Lemma 5.** *The algorithm emits a valid plcpcomp factorization of $T[1\,..\,dst + PLCP[dst] - 1]$.*

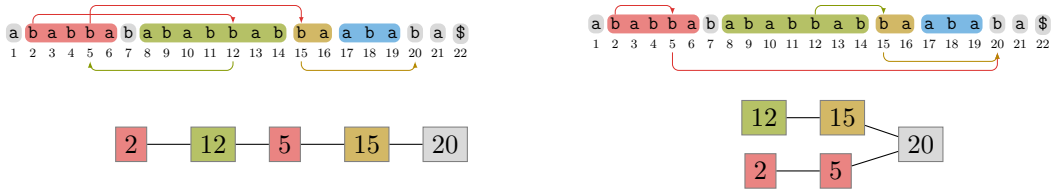


Figure 7 Pointer jumping applied to references. Suppose that our example text is represented by the coding described in Figure 4. To extract the character $T[2]$, we need to resolve the reference $(12, 5)$, which has a depth of three (*bottom left* figure). In case that we split all references into references of length one, we can reduce the depth of the reference associated with $T[2]$ by pointer jumping (*right* figure). The order in which this technique is applied to the references has an impact on the resulting references. Here, we assumed that we can apply this technique *in parallel*.

After factorizing $T[1 \mathinner{..} dst + PLCP[dst] - 1]$, we proceed with Algo. 1 on the remaining text $T[dst + PLCP[dst] \mathinner{..}]$ to compute the factorization of the entire text. It is left to explain how this algorithm can be adapted to the EM model efficiently.

3.1 Factorization in External Memory

Having the text, $PLCP$, and Φ stored as files in EM, we can compute the *plcpcomp* scheme in three sequential scans over n tuples and one sort operation:

1. Proceed with Algo. 1 to find pairs $(dst, \ell = PLCP[dst])$ representing referencing factors $T[dst \mathinner{..} dst + \ell]$ by scanning $PLCP$.
2. Sort these pairs in ascending order of their dst components (i.e., in text order).
3. Simultaneously scan this sorted list of pairs and Φ to compute triplets of the form $(dst, src = \Phi[dst], \ell)$, where the second component is the referred position of the referencing factor $T[dst \mathinner{..} dst + \ell - 1]$.
4. Finally, scan simultaneously the list of references and T to replace each substring $T[dst \mathinner{..} dst + \ell - 1]$ by the reference (src, ℓ) on reading the triplet (dst, src, ℓ).

The pairs emitted during the $PLCP$ scan (Step 1) can be stored and then sorted in EM. The references computed by the second scan can be written to disk for the final scan, which computes the *plcpcomp* scheme of T sequentially. By doing so, no random access is required on the list of references.

During the $PLCP$ scan, the list L can also be maintained on disk efficiently: until a maximal peak is found, we only append peaks to L. For our experiments, we store L in RAM, as the number of elements was much lower than the upper bound $\mathcal{O}(\min(\sqrt{n \lg n}, r))$ where r is the number of BWT runs (see the full version of this paper).

Once a maximal peak dst has been found and a reference (dst, ℓ) is emitted, we scan over L sequentially (a) to apply Rules $(\mathcal{D})$ and $(\mathcal{R})$ and (b) to find a remaining maximal peak, if any, in the process. We then repeat this process until there are no more maximal peaks in L. In practice, we scan the last elements of L linearly from right to left, since only the last interesting peaks need to be updated.

4 Decompression

The task of decompressing a bidirectional scheme is to *resolve* each reference (src_i, ℓ_i) of a referencing factor $T[dst_i \mathinner{..} dst_i + \ell_i - 1]$, i.e., to copy the characters from $T[src_i \mathinner{..} src_i + \ell_i - 1]$ to $T[dst_i \mathinner{..} dst_i + \ell_i - 1]$.

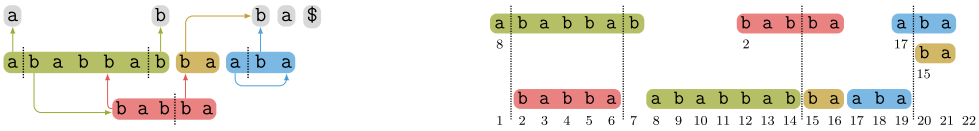


Figure 8 The dependency graph (*left*) and its EM representation (*right*) of the factorization given in Figure 4. The multi-dependent factors of length seven and five have a cyclic dependency. The EM representation of the graph described in Section 4 consists of two copies of the list of all referencing factors, sorted by their source position (*top*) as well as sorted by their destination (*bottom*).

A unidirectional scheme can be decompressed by scanning linearly over the compressed input from left to right. In that scenario, references can be resolved easily because they always refer to already decompressed parts of the text [2]. This property does not hold for a bidirectional scheme in general, as a reference can refer to a part of the text that again corresponds to a reference.

▶ **Definition 6** (Dependency Graph). *Given a bidirectional factorization $F_1 \cdots F_b = T$, we model its references as a directed graph G with $V = \{v_1, \ldots, v_b\}$ such that there is a 1-to-1 correlation between nodes v_i and factors F_i. We add a directed edge (v_i, v_j) from v_i to v_j with $i \neq j$ iff F_i refers to at least one character in the factor F_j. We put these edges into a set E to form a graph $G := (V, E)$ that has only literal factors as sinks. A node v_i can have more than one out-going edge if the referred substring is covered by multiple factors; in this case, we say v_i is* multi-dependent *and call the set of its out-going edges a* multi-dependency*. The dependency graph of our example from Figure 4 can be seen in Figure 8.*

Bidirectional decompressors face two challenges arising from this graph structure:

- The existence of multi-dependent nodes disallows efficient tree-based approaches.
- Long dependency chains may affect the time and space complexity of decompression.

Our compression scheme splits multi-dependencies into single dependencies and deploys the pointer jumping technique [14, Sect. 2.2] for dependency resolution. After the resolution we obtain a dependency graph in which each reference is single-dependent on a literal factor. Then the text can be trivially recovered with $\mathrm{sort}(n)$ I/Os. The details are as follows.

Let G be the dependency graph of the factorization $T = F_1 \cdots F_b$. For now we assume that all factors are single-dependent, i.e., each node v representing a referencing factor has exactly one outgoing edge $(v, p(v))$. For all other nodes (representing literal factors) we define $p(v) := v$. Clearly, G forms a forest in which each tree is rooted in a literal factor. When applying the pointer jumping technique, we take each referencing factor and attach it to the parent of its parent (cf. Figure 7). Given that G' is the resulting graph with $p'(v) = p(p(v))$, we thereby halve the depth, i.e., $d(G') = \lceil d(G)/2 \rceil$ if $d(G) \geq 2$, where $d(G)$ denotes the maximum depth of a tree in G. Hence, after $\Theta(\lg d(G))$ iterations all indirect references are resolved and have been replaced by direct references to literal factors.

If we allow multi-dependencies, pointer jumping is only possible for single-dependent nodes. To apply pointer jumping, we split each multi-dependent reference into the smallest possible set of single-dependent references. A split is introduced ad-hoc each time it is required for a pointer jump. The details of the splitting are discussed in the full version of this paper.

We first construct a representation of the dependency graph consisting of two EM vectors called requests and factors. Intuitively, each request (child) sends a request message to the first factor it refers to (parent). Addressing is implemented indirectly in terms of

Figure 9 Split-Strategy of *EM-PJ* applied to the first (*left* figure) and second (*right* figure) referencing factor of the factorization given in Figure 4. *EM-PJ* splits up references in a minimal number of sub-references on which the pointer jumping technique can be applied. The *left* figure shows such an application to the reference of the leftmost referencing factor that is split into two sub-references. The first and second sub-reference receive new referred positions based on the referred positions of the second and third referencing factors, respectively. In the *right* figure, we split up the next reference $(1, 7)$ in four sub-references, where the first and last sub-reference refer to literal factors.

text positions rather than factor indices. For each reference (src, ℓ) corresponding to a factor $F_i = T[dst \mathinner{..} dst + \ell - 1]$, we push $\langle dst, \ell, src\rangle$ into requests and $\langle src, \ell, dst\rangle$ into factors. We omit literal factors, since the lack of a reference in factors for a certain text position indicates the presence of a literal factor.

Subsequently, we sort[2] both vectors independently, bringing the messages in requests and the recipients in factors into the same order. On the right side of Figure 8 we see a visualization of the lists (after the initial sorting) for our running example. We augment requests with an initially empty EM priority queue PQSplit. In the following, after processing a factor F_i, we write F_i either to a vector result if it refers to literal factors, or to a vector nextRequests otherwise: Let $\langle dst, \ell, src\rangle$ be the smallest unprocessed request of a factor F_i received via requests or PQSplit. If it originates from requests, we advance requests's read pointer for the next iteration, otherwise we dequeue the top element from PQSplit. We process the read request $\langle dst, \ell, src\rangle$ depending on the following cases (cf. Figure 9):

- **Jump** The request is completely covered by parent F_j in factors. In this case, we substitute F_i's reference according to F_j and push it into nextRequests to be processed in the next iteration.
- **Finalize** No parent (partially) overlapping with F_i is available in factors. Then we know that F_i points to a substring contained in literal factors. We finalize F_i by pushing it into result.
- **Split** A prefix of F_i is contained in the parent F_j or points to literals. Let $\ell' < \ell$ be the length of the longest such prefix. Then split F_i into a prefix F_i^{P} of length ℓ' and a suffix F_i^{S} of length $\ell - \ell'$. By construction, either case "Jump" or case "Finalize" is applicable to F_i^{P}, and we execute it directly. Then we push $\langle src{+}\ell', \ell{-}\ell', dst + \ell'\rangle$ representing F_i^{S} into PQSplit to process it later within the same iteration. Observe that F_i can be split multiple times during the same iteration.

If nextRequests is not empty, we sort it and recurse by processing nextRequests and the (unaltered) factors simultaneously as before. With these steps, we obtain the final result:

▶ **Theorem 7.** *Let $F_1 \cdots F_b = T$ be a ξ-restricted bidirectional scheme, and $d(G) < b$ be the depth of T's dependency graph G. Then EM-PJ requires $\mathcal{O}(\lg(d(G))\,\mathrm{sort}(n/\xi))$ I/Os.*

[2] To sort tuples we always use lexicographic order, i.e., we order tuples by their first unequal elements.

Table 1 Empirical entropies of our data sets. The alphabet sizes of all instances are 242 and 4 for `commoncrawl` and `dna`, respectively.

`commoncrawl` prefix length	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
16 GiB	5.99165	4.26109	3.48920	2.94113	2.42738	2.01886	1.64558	1.35130
32 GiB	5.99145	4.26160	3.49006	2.94411	2.43471	2.03284	1.66737	1.37798
64 GiB	5.99119	4.26209	3.49100	2.94669	2.44088	2.04409	1.68482	1.40001
128 GiB	5.99177	4.26148	3.49055	2.94684	2.44231	2.04753	1.69087	1.40839
`dna` prefix length	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
16 GiB	1.9715	1.94676	1.93166	1.92232	1.91167	1.89491	1.87101	1.84585
32 GiB	1.97128	1.94561	1.93201	1.92421	1.91507	1.90190	1.88270	1.86160
64 GiB	1.97067	1.94506	1.93145	1.92424	1.91588	1.90445	1.88763	1.86889
128 GiB	1.97528	1.95010	1.93873	1.93273	1.92486	1.91341	1.89601	1.87634

5 Practical Evaluation

Experimental Setup. Our experiments are conducted on a machine with 16 GiB of RAM[3], eight `Hitachi HUA72302` hard drives with 1.8 TiB, two `Samsung SSD 850` SSDs with 465.8 GiB, and an `Intel Xeon CPU i7-6800K`. The operating system is a 64-bit version of Ubuntu Linux 16.04. We implemented *plcpcomp*[4] in the version 1.4.99 (development snapshot) of the STXXL library [5]. We compiled the source code with the GNU `g++ 7.4` compiler.

Text Collections. We conduct our experiments on two texts of different alphabet sizes and repetitiveness (cf. Table 1):

- COMMONCRAWL: A crawl of web pages with an alphabet size of 242 collected by the commoncrawl organization.
- DNA: DNA sequences with an alphabet size of 4 extracted from FASTA files.

Algorithms. We compare *plcpcomp* against *EM-LPF* [17] by Kärkkäinen et al., which is an EM algorithm computing the *LZ77* factorization by constructing the *LPF* array. In addition to the input text, it requires *SA* and *LCP*.

In early experiments with *EM-LZscan* [17], it became clear that its throughput on the text collection we use is nowhere near competitiveness with *EM-LPF* and *plcpcomp*. Therefore, it is not considered in our experiments. Semi-external *LZ77* algorithms like *SE-KKP* [17] storing the text or parts of the text in RAM have not been considered.

Data Structures. Currently, the fastest way to compute the data structures *PLCP* and Φ in EM is to compute *BWT* from *SA* with the parallel EM algorithm *pEM-BWT* by Kärkkäinen and Kempa[5], and use it for computing *PLCP* with the parallel EM construction algorithm of [16]. We modified the source code of the latter to also produce Φ as a side product.

[3] In order to avoid swapping, each experiment was conducted with a limit of 14 GiB of RAM.
[4] Available at `https://github.com/tudocomp/tudocomp`.
[5] `https://www.cs.helsinki.fi/u/dkempa/pem_bwt.html`

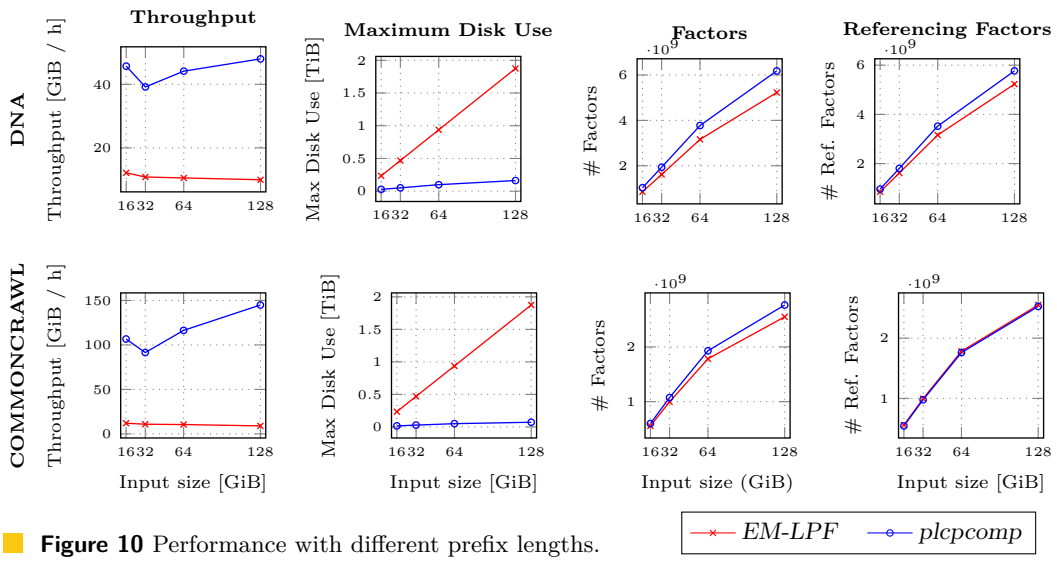


Figure 10 Performance with different prefix lengths.

For *EM-LPF*, we additionally need to convert *PLCP* to *LCP* by a scan over *SA* and a subsequent sort step. This is currently the fastest approach for obtaining *LCP*, as other approaches building *LCP* directly from *SA* like [15] are slower.

Consequently, both contestants need (directly or indirectly) *SA*. However, it takes a considerable amount of time to construct it with EM algorithms on a single machine (e.g., with pSAScan [18]). To put the focus on the comparison between *EM-LPF* and *plcpcomp*, we do not take into account the construction of *SA* and *LCP* when measuring running times.

Measurements and Results. Our experiments measure the throughput, the maximum hard disk usage, and the number of referencing factors, for *EM-LPF* and *plcpcomp* for 2^kGiB prefixes ($4 \leq k \leq 7$) of our data sets DNA and COMMONCRAWL. We collected the median of three iterations and present the results in Figure 10. The plots show that *plcpcomp* is magnitudes faster on both data sets (cf. plots "Throughput"). The reason for this could be that the disk accesses of *EM-LPF* scale much worse than those of *plcpcomp* (cf. plots "Maximum Disk Use"). We point out that *plcpcomp* is already faster than the step for computing *LCP* from *PLCP* and *SA*. Regarding the number of factors, *plcpcomp* is on par with *LZ77* (rightmost plots), producing, relatively speaking, slightly more factors. Our decompression requires multiple sorts of factor sets depending on the maximum depth of (a tree in) the dependency graph induced by the factorization. Therefore, it is not surprising that it is a lot slower than the comparatively simple compression.

Furthermore, and for the same reason, our decompression expectedly runs orders of magnitudes slower than the external memory Lempel-Ziv decoder of [2], which is why we do not do a more detailed performance comparison here.

Decompression. We ran our decompressor implementation on the *plcpcomp* codings of our datasets. Plots of the scaling experiments are shown in Figure 11. As the decompression algorithm is superlinear, the throughput is decreasing with increasing text size. However, comparing the results for the 32GiB and 64GiB commoncrawl decompression, the throughput only decreases by 1%. The throughput between the 32GiB and 64 GiB DNA decompression differs by only 5%. The maximum external memory allocation rises linearly with increasing text size.

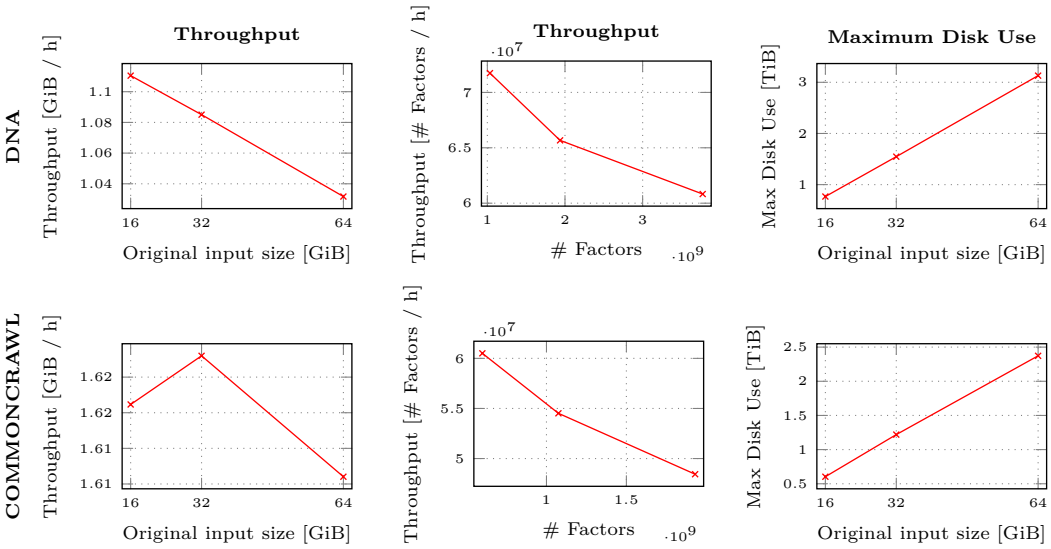


Figure 11 Performance of the decompression with different prefix lengths.

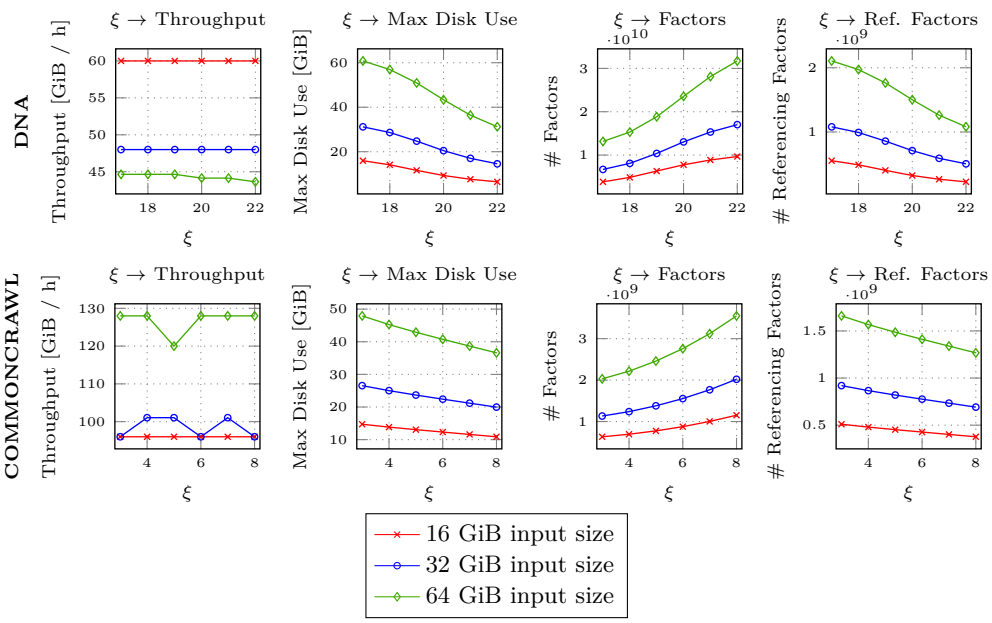


Figure 12 Evaluation of *plcpcomp* with different threshold values ξ.

In Figure 12, we measured the impact of the choice of ξ on the compressed output and the decompression algorithm of our datasets. For larger values of ξ, *plcpcomp* creates less referencing factors, but the total number of factors increases (as we obtain much more literal factors). Having less referencing factors, the decompression needs less disk space.

Our decompression requires multiple sorting steps on the factor lists such as requests (cf. Section 4). The number of these steps depend on the maximum depth of (a tree in) the dependency graph induced by the factorization. Therefore, it is not surprising that the decompressor is magnitudes slower than the comparatively simple compression algorithm.

Furthermore, and for the same reason, our decompression (expectedly) runs slower than the external memory Lempel-Ziv decoder of [2], which is why we skip a more detailed performance comparison here.

6 Conclusions

We presented *plcpcomp*, the first external memory bidirectional compression algorithm, and showed its practicality by performing experiments on very large data sets, using only very limited RAM. We also presented a decompression algorithm in external memory, which can decode the output of any bidirectional compression scheme (not only *plcpcomp*). Possible future steps include relating the number of factors of *plcpcomp* to the minimal number of factors in a bi- or unidirectional compression scheme, evaluating the whole compression chain by also experimenting on *codings* of the output of *plcpcomp* (similar to [6]), and improving the performance of the decompression algorithm.

References

1. Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. Djamal Belazzougui, Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv decoding in external memory. In *Proc. SEA*, volume 9685 of *LNCS*, pages 63–74, 2016.
3. Timothy C. Bell. Better OPM/L Text Compression. *IEEE Trans. Communications*, 34(12):1176–1182, 1986.
4. M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
5. Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Softw., Pract. Exper.*, 38(6):589–637, 2008.
6. Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane. Compression with the tudocomp Framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017. `arXiv:1702.07577`.
7. Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.
8. Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the Bit-Complexity of Lempel-Ziv Compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.
9. Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv Factorization Powered by Space Efficient Suffix Trees. *Algorithmica*, 80(7):2048–2081, 2018.
10. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. On the Approximation Ratio of Lempel-Ziv Parsing. In *Proc. LATIN*, volume 10807 of *LNCS*, pages 490–503, 2018.
11. J. K. Gallant. *String compression algorithms*. PhD thesis, Princeton University, 1982.
12. Keisuke Goto and Hideo Bannai. Simpler and Faster Lempel Ziv Factorization. In *Proc. DCC*, pages 133–142, 2013.
13. Keisuke Goto and Hideo Bannai. Space Efficient Linear Time Lempel-Ziv Factorization for Small Alphabets. In *Proc. DCC*, pages 163–172, 2014.

14 Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

15 Juha Kärkkäinen and Dominik Kempa. LCP array construction in external memory. *ACM Journal of Experimental Algorithmics*, 21(1):1.7:1–1.7:22, 2016.

16 Juha Kärkkäinen and Dominik Kempa. Engineering External Memory LCP Array Construction: Parallel, In-Place and Large Alphabet. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 17:1–17:14, 2017.

17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. DCC*, pages 153–162, 2014.

18 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel External Memory Suffix Sorting. In *Proc. CPM*, volume 9133 of *LNCS*, pages 329–342, 2015.

19 Juha Kärkkäinen, Dominik Kempa, and Simon John Puglisi. Lightweight Lempel-Ziv Parsing. In *Proc. SEA*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013.

20 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In *Proc. CPM*, volume 5577 of *LNCS*, pages 181–192, 2009.

21 Dominik Kempa and Dmitry Kosolobov. LZ-end parsing in compressed space. In *Proc. DCC*, pages 350–359, 2017.

22 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. STOC*, pages 827–840, 2018.

23 Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proc. DCC*, pages 239–248, 2010.

24 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

25 Markus Mauer, Timo Beller, and Enno Ohlebusch. A Lempel-Ziv-style Compression Method for Repetitive Texts. In *Proc. PSC*, pages 96–107, 2017.

26 Ryosuke Nakamura, Shunsuke Inenaga, Hideo Bannai, Takashi Funamoto, Masayuki Takeda, and Ayumi Shinohara. Linear-Time Text Compression by Longest-First Substitution. *Algorithms*, 2(4):1429–1448, 2009.

27 Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. $\mathcal{O}(n \log n)$-time text compression by LZ-style longest first substitution. In *Proc. PSC*, pages 12–26, 2018.

28 Takaaki Nishimoto and Yasuo Tabei. LZRR: LZ77 parsing with right reference. *arXiv 1812.04261*, 2018. `arXiv:1812.04261`.

29 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

30 Vladimir Yanovsky. ReCoil - an algorithm for compression of extremely large datasets of DNA data. *Algorithms for Molecular Biology*, 6(23):1–9, 2011.