Bidirectional Text Compression in External Memory

Patrick Dinklage, Jonas Ellert, Johannes Fischer, Dominik Köppl, Manuel Penschuck

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.

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, 32], two occurrences of the same substring are neglected if their distance is longer than the sliding window. More advanced solutions [10, 13, 14, 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 [9, Sect. 6]. In recent years, these algorithms have also been transformed to the external memory (EM) model [2, 20, 23].

In this article, we present a modification of LZ77, called plcpcomp, which is based on the bidirectional compression scheme lcpcomp of Dinklage et al. [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 Storer and Szymanski [32] who also coined this notation. Gallant [12] proved that finding the optimal bidirectional parsing, i.e., a bidirectional parsing with the lowest number of factors, is NP-complete. Dinklage et al. [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. Mauer et al. [28] combined the techniques for lcpcomp [6] and the longest-first grammar compression [29] in a compression algorithm running in $O(n^2)$ time, which was subsequently improved to $O(n \lg n)$ time by Nishi et al. [30]. Recently, Gagie et al. [11] showed an upper bound of $z = O(b \lg (n/b))$ 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). Kempa and Prezza [24] 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 Nishimoto and Tabei [31] 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 [33] presented a compressor called ReCoil that is specialized on large DNA datasets. Ferragina et al. [8] gave a construction algorithm of the Burrows-Wheeler transform in EM. For LZ77 compression, Kärkkäinen et al. [20] 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 Belazzougui et al. [2], which also works in general for all files that have been compressed by a unidirectional scheme. Finally, Kempa and Kosolobov [23] presented an EM algorithm for computing the BWT of datasets. The EM-LPF compression, Kärkkäinen et al. [20] 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 Belazzougui et al. [2], which also works in general for all files that have been compressed by a unidirectional scheme. Finally, Kempa and Kosolobov [23] presented an EM algorithm for computing the LZipEnd scheme [25], 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 $n$ contiguous words from or to disk requires scan($n$) = $\Theta(n/B)$ I/Os. Sorting $n$ contiguous words requires sort($n$) = $\Theta((n/B) \cdot \log M/B(n/B))$ I/Os. For realistic values of $n$, $B$, and $M$, we stipulate that scan($n$) < sort($n$) $\ll n$.

Text Let $\Sigma$ denote an integer alphabet of size $\sigma = |\Sigma| = n^{O(1)}$ for a natural number $n$. The alphabet $\Sigma$ induces the lexicographic order $\prec$ on the set of strings $\Sigma^*$. 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 $V = T[i \ldots j]$ and $W = T[j + 1 \ldots]$. 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 $\Sigma$.

Text Data Structures Let $SA$ denote the suffix array [27] of $T$. The entry $SA[i]$ is the starting position of the $i$-th lexicographically smallest suffix such that $T[SA[i] \ldots] \prec T[SA[i + 1] \ldots]$ 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 $BWT[i] = T[sa]$ 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 $T[SA[i] \ldots]$ and $T[SA[i - 1] \ldots]$ for $i = 2, \ldots, n$. For convenience, we stipulate that $LCP[1] := 0$. The array $\Phi$ 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., $PLCP[SA[i]] = LCP[i]$. Fig. [1] illustrates the introduced data structures.

| $i$ | 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   | a   | b   | 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   |
| $\Phi$ | 6   | 12  | 13  | 13  | 14  | 18  | 17  | 5   | 1   | 2   | 3   | 4   | 7   | 8   | 9   | 20  | 21  | 19  | 15  | 16  | 10  | 22  |
| LCP | 0   | 0   | 1   | 1   | 3   | 5   | 4   | 2   | 4   | 1   | 5   | 0   | 2   | 2   | 4   | 5   | 3   | 5   | 6   | 1   | 3   | 4   |
| PLCP | 4   | 5   | 4   | 3   | 4   | 5   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 2   | 1   | 3   | 2   | 1   | 0   | 0   | 0   |

Figure 1: Suffix array, its inverse, $\Phi$, LCP array, and PLCP array of our running example string $T$. 
Figure 2: Visualization of Rules $D$ and $R$ 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 | a | b | a | b | a | b | a | b | a | $\$ |
| PLCP | 4 | 5 | 4 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 1 | 3 | 2 | 1 | 0 | 0 | 0 |
| PLCP$^1$ | 4 | 5 | 4 | 3 | $\underline{3}$ | $\underline{2}$ | $\underline{1}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PLCP$^2$ | 1 | $\underline{0}$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

Figure 3: Step-by-step computation of the instructions in Section 2 computing the plcpcomp compression scheme on $T = \text{ababbabababbababa}$.

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 $T[i..i + \text{PLCP}[i] - 1]$. 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 $\Phi$ 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 $[32]$ 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 $(\text{src}, \ell)$ such that $F_x$ and $T[\text{src}..\text{src} + \ell - 1]$ are two different but possibly overlapping occurrences of the substring $F_x$ in $T$. The pair $(\text{src, } \ell)$ and the text position $\text{src}$ are called reference and referred position, respectively. A factorization is cycle-free, i.e., references are not allowed to have cyclic dependencies. A factorization is called $\xi$-restricted for an integer $\xi \geq 2$ if each referencing factor $F_x$ is at least $\xi$ 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
The factorization described in Fig. 3 computes four referencing factors, listed in the table on the right. These factors are coded by their references. The factorization with PLCP in Fig. 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’ in the table). 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.

A 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 \( C := \{ i \mid \text{PLCP}[i] \geq \text{PLCP}[j] \text{ for all text positions } j \} \).
2. Let \( dst \) be the leftmost position of all candidate positions \( C \). Terminate if \( \text{PLCP}[dst] < \xi \).
3. Create a referencing factor by replacing \( T[dst..dst + \text{PLCP}[dst] - 1] \) with the reference \( (\Phi[dst], \text{PLCP}[dst]) \).
4. Apply the following rules to ensure that we do not create overlapping factors (cf. Fig. 2):
   - (D) Decrease \( \text{PLCP}[j] \leftarrow \min(\text{PLCP}[j], dst - j) \) for every \( j \in [dst - \text{PLCP}[dst], dst) \).
   - (R) Remove the factored positions by setting \( \text{PLCP}[dst + k] \leftarrow 0 \) for every \( k \in [0, \text{PLCP}[dst]) \).
5. Recurse with the modified PLCP.

An application of the above instructions on our running example is given in Fig. 3. The coding is visualized in Fig. 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 \( C \). Dinklage et al. [6] presented an algorithm computing the lcpcomp scheme in \( O(n \log n) \) 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 \( dst \in C \), 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 \( \text{PLCP}[dst] \geq \xi \) and one of the following conditions holds:

| dst | src | length |
|-----|-----|--------|
| 8   | 1   | 7      |
| 2   | 12  | 5      |
| 17  | 19  | 3      |
| 15  | 20  | 2      |
Algorithm 1: Computation of \textit{plcpcomp} factors.

\begin{algorithm}
\caption{Computation of \textit{plcpcomp} factors.}
\begin{algorithmic}[1]
\State $L \leftarrow \emptyset$ \hfill \small{\textit{// Step 1a}}
\For{$\text{dst} = 1$ to $n$} \hfill \small{\textit{// Step 1b}}
\If{$\text{dst}$ is a maximal peak} \hfill \small{\textit{// Step 2}}
\State create a referencing factor replacing $T[\text{dst}..\text{dst} + \text{PLCP}[\text{dst}] - 1]$ \hfill \small{\textit{// Step 3}}
\State apply Rule \textit{(D)} to the peaks in $L$
\While{$L$ contains maximal peaks} \hfill \small{\textit{// Step 4}}
\State $j \leftarrow$ rightmost maximal peak in $L$
\State create referencing factor replacing $T[j..j + \text{PLCP}[j] - 1]$
\State apply Rules \textit{(D)} and \textit{(R)} to the peaks in $L$
\State remove those elements of $L$ that are no longer interesting peaks
\EndWhile
\State $\text{dst} \leftarrow \text{dst} + \text{PLCP}[\text{dst}]$
\If{$\text{dst}$ is an interesting peak} \hfill \small{\textit{// Step 5}}
\State $L \leftarrow L \cup \{\text{dst}\}$
\EndIf
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

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

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

Given an interesting peak $\text{dst}$, there is no text position $j$ with $\text{PLCP}[j] \geq \text{PLCP}[\text{dst}]$ that becomes the starting position of a referencing factor containing $T[\text{dst}]$ (such that $\text{PLCP}[\text{dst}]$ cannot be removed according to Rule \textit{(R)}). Given a maximal peak $\text{dst}$, there is additionally no text position $j$ with $\text{PLCP}[j] > \text{PLCP}[\text{dst}]$ for which we apply Rule \textit{(D)} on $\text{PLCP}[\text{dst}]$ after factorizing $T[j..j + \text{PLCP}[j] - 1]$. Informally, we can determine whether a peak is interesting by looking at the PLCP values before this peak, whereas we need to also look \textit{ahead} for determining whether a peak is maximal. Given that there is at least one PLCP entry with a value of at least $\xi$, we can find a maximal peak, since the leftmost position $\min \{i \in [1..n] | \text{PLCP}[i] \geq \text{PLCP}[j] \text{ for all } j \text{ with } 1 \leq j \leq n\}$ 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.

\begin{lemma}
If the text position $\text{dst}$ is a maximal peak, then $T[\text{dst}..\text{dst} + \text{PLCP}[\text{dst}] - 1]$ is a referencing factor.
\end{lemma}

\begin{proof}
When applying Rules \textit{(R)} and \textit{(D)} we do not change the value of $\text{PLCP}[\text{dst}]$, since $\text{dst}$ is a maximal peak. Therefore, we will eventually create a referencing factor starting with $\text{dst}$.
\end{proof}

Our preliminary algorithm consists of the following steps:

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

This algorithm produces the \textit{plcpcomp} scheme, because

\footnote{A subset of the so-called \textit{irreducible} PLCP entries \cite{irreducible} \textit{Lemma 4} have this property.}
As a consequence, Rule (D) is applied to the only peak stored in \( L \). Since \( L \) is 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 (Fig. 6c).

In the second step (Fig. 6b), the referencing factor \( F \) is introduced, which starts at this maximal peak. As a consequence, Rule (D) is applied to the only peak stored in \( L \), the one at position 1. However, because the PLCP value at position 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 (Fig. 6e). The factor \( F_2 \) (Fig. 6e) is introduced, and Rule (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.

### Example 3.2

Figure 6 illustrates Algo. 1 on the prefix \( T[1..14] = ababababababb \) 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] (Fig. 6a).

In the second step (Fig. 6b), the referencing factor \( F_1 \) is introduced, which starts at this maximal peak. As a consequence, Rule (D) is applied to the only peak stored in \( L \), the one at position 1. However, because the PLCP value at position 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 (Fig. 6e). The factor \( F_2 \) (Fig. 6e) is introduced, and Rule (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 (D) and (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 3.5). For that, we show that we can produce the plcpcmp 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 \( L \):

| \( i \) | 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 | a | b | 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\(^2\) | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 3 | 2 | 1 | 0 | 0 | 0 |
| PLCP\(^3\) | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 5: Step-by-step computation of our plcpcmp algorithm on \( T = abababababababbaba $ \). While the instructions of the scheme (cf. Section 2) always replace the factor starting at a position with the maximal PLCP value (cf. Fig. 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 plcpcmp scheme, but in different order.

- \( T[dst.. dst + PLCP[dst] - 1] \) is a referencing factor for each selected leftmost maximal peak \( dst \) according to Lemma 3.1 and
- the part \( T[1.. dst - 1] \) can be factorized independently from how \( T[dst + PLCP[dst].. ] \) is factorized, and vice versa. That is because, having already \( T[dst.. dst + PLCP[dst] - 1] \) factorized, we can no longer create a factor that covers a text position in the range \([dst.. dst + PLCP[dst] - 1]\].

Hence, we can factorize \( T[1.. dst - 1] \) without considering the factorization of the rest of the text to produce the correct plcpcmp scheme. Figure 5 illustrates the computation of the plcpcmp 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 \( n \log n \) bits of space if we require constant time read and write access. Instead of keeping PLCP[1.. dst - 1] in RAM, we now 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 alternative is that we apply Step 1 only to the peaks stored in \( L \). There, we scan \( L \) from right to left while applying Rule (D) and removing all elements that are no longer interesting peaks. The modified algorithm is sketched as pseudo code in Algo. 1.
Lemma 3.3. The positions stored in \( L \) are in strictly ascending order with respect to their LCP values.

Proof. Let \( dst \) be the leftmost maximal peak. Assume that there is an entry \( L[k] < dst \) with \( 1 \leq k \leq |L - 1| \) and \( PLCP[L[k + 1]] \leq PLCP[L[k]] \). Since \( L[k] \) is an interesting peak, there is no text position \( j \) with \( L[k] < (j, j + PLCP[j]) \) and \( PLCP[j] \geq PLCP[L[k]] \). Since \( L[k + 1] \) is the succeeding interesting peak of \( L[k] \) (with respect to text order), \( L[k + 1] < L[k] + PLCP[L[k]] \) must hold. Otherwise, \( L[k] \) would be a maximal peak, which contradicts the fact that \( dst \) is the leftmost maximal peak. However, the condition \( PLCP[L[k]] < L[k + 1] \) must hold for \( L[k + 1] \) to be interesting.

Next, we examine the result of creating the referencing factor \( T[dst \ldots dst + PLCP[dst] - 1] \) 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:

Lemma 3.4. Applying Rules \([D]\) and \([R]\) after creating a referencing factor \( F_x \) does not cause new peaks, with the only possible exception of the position succeeding the end of \( F_x \).

Proof. Let \( dst \) be the starting position of the referencing factor \( F_x \) and let \( j < dst \) be a position that is not a peak at the time before the creation of \( F_x \). Then \( PLCP[j - 1] \geq PLCP[j] \). After creating \( F_x \), it holds that

\[
PLCP'[j - 1] = \min(PLCP[j - 1], dst - j) \geq \min(PLCP[j], dst - j - 1) = PLCP'[j],
\]

where \( PLCP' \) is the modified PLCP array after applying Rules \([D]\) and \([R]\). Hence, position \( j \) did not become a peak. If \( j = dst + PLCP[dst] \) is the position succeeding the end of \( F_x \), then \( PLCP[dst + PLCP[dst] - 1] = 0 \) according to Rule \([R]\). Hence, \( j \) becomes a peak if \( PLCP[j] \geq \xi > 0 \).

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

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

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

1. Substitute \(T[\text{dst}'..\text{PLCP}[\text{dst}'] - 1]\) with a reference, because it is the highest peak in \(T[1..\text{dst} - 1]\).

2. If \(\text{dst}'' := \text{dst}' + \text{PLCP}[\text{dst}']\) with \(\text{PLCP}[\text{dst}''] \geq \xi\) was not a peak, then \(\text{dst}''\) becomes an interesting peak. In this case, substitute \(\text{dst}'\) with \(\text{dst}''\) in \(L\) to preserve the order in \(L\). Otherwise, remove \(\text{dst}'\) from \(L\).

3. Split \(L\) into two sub-lists:
   - one containing text positions of the range \([1..\text{dst}' - 1]\), and
   - the other containing text positions of the range \([\text{dst}' + \text{PLCP}[\text{dst}']..\text{dst} - 1]\).

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 3.5.** The algorithm emits a valid \text{plcpcomp} factorization of \(T[1..\text{dst} + \text{PLCP}[\text{dst}] - 1]\).

After factorizing \(T[1..\text{dst} + \text{PLCP}[\text{dst}] - 1]\), we proceed with Algo. 1 on the remaining text \(T[\text{dst} + \text{PLCP}[\text{dst}]..]\) 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 \(\Phi\) stored as files in EM, we can compute the \text{plcpcomp} scheme in three sequential scans over \(n\) tuples and one sort operation:

1. Proceed with Algo. 1 to find pairs \((\text{dst}, \ell = \text{PLCP}[\text{dst}])\) representing referencing factors \(T[\text{dst}..\text{dst} + \ell]\) by scanning PLCP.

2. Sort these pairs in ascending order of their \(\text{dst}\) components (i.e., in text order).

3. Simultaneously scan this sorted list of pairs and \(\Phi\) to compute triplets of the form \((\text{dst}, \text{src} = \Phi[(\text{dst}), \ell]),\) where the second component is the referred position of the referencing factor \(T[\text{dst}..\text{dst} + \ell - 1]\).

4. Finally, scan simultaneously the list of references and \(T\) to replace each substring \(T[\text{dst}..\text{dst} + \ell - 1]\) by the reference \((\text{src}, \ell)\) on reading the triplet \((\text{dst}, \text{src}, \ell)\).

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 \text{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 \(L\).

Once a maximal peak \(\text{dst}\) has been found and a reference \((\text{dst}, \ell)\) is emitted, we scan over \(L\) sequentially (a) to apply Rules \((D)\) and \((R)\) 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. For our experiments, we store \(L\) in RAM, as the number of elements was much lower than the following upper bound:

**Lemma 3.6.** \(|L| = \mathcal{O}(\min(\sqrt{n\ln n}, r))\), where \(r\) is the number of BWT runs.
Proof. The list $L$ stores all interesting peaks between two different maximal peaks (or between the first position and the first maximal peak). Given an interesting peak $dst$ with $PLCP[dst]$, there is no peak $j$ with $PLCP[j] ≥ PLCP[dst]$ and $j < dst < j + PLCP[j]$. In order to be added to $L$, the peak $dst$ must not be a maximal peak, i.e., there must be a text position $j$ with $dst < j < dst + PLCP[dst]$ and $PLCP[j] > PLCP[dst]$. The worst case is that $j = dst + 1$, $PLCP[j] = PLCP[dst] + 1$, and $j$ is again an interesting peak that is not maximal. By induction, we may insert $m$ interesting non-maximal peaks $(j_1, \ldots, j_m)$ into $L$ with $j_i + 1 ≤ j_{i+1}$ for $1 ≤ i < m - 1$ and $PLCP[j_i] ≥ i$ for $1 ≤ i ≤ m$.

However, $\sum_{i=1}^{m} i ≤ \sum_{i=1}^{m} PLCP[j_i] = O(n \log n)$ due to [22] Thm. 12, such that $m = O(\sqrt{n \log n})$. From the same reference [22] Sect. 4, we obtain that $m = O(r)$. □

Lemma 3.7. There are texts of length $n$ for which $|L| = Θ(\sqrt{n})$.

Proof. For the proof, we use the following definition: Given an interval $I$, we define $b(I)$ and $e(I)$ to be the starting and the ending position of $I = [b(I) .. e(I)]$, respectively.

Let $Σ := \{σ_1, \ldots, σ_m\}$ be an alphabet with $σ_1 < σ_2 < \ldots < σ_m$. Set $F_m := σ_m$, and $F_i := σ_i F_{i+1} σ_i$ for $1 ≤ i ≤ m - 1$. Then our algorithm finds $L$ with $Θ(\sqrt{n})$ interesting peaks on processing the text $T := F_m \cdots F_1$. This is due to the following:

First, $Φ[b(F_i)] = b(F_{i+1}) + 1$ for each $i$ with $1 ≤ i ≤ m - 1$, since

- $T[b(F_{i-1})] = F_i F_{i-1} \cdots F_1 σ_{i-1} σ_{i-2} \cdots σ_1 ≥ 1$ for all $j$ with $0 ≤ j ≤ i - 1$.

Hence, $T[b(F_{i-1}) + j] = T[b(F_{i-1}) + j-1] = T[b(F_{i-1})] = F_i F_{i-1} F_{i-2} \cdots F_1 σ_{i-1} σ_{i-2} \cdots σ_1 ≥ 1$ for all $j$ with $0 ≤ j ≤ i - 1$. For all positions $1 ≤ j ≤ n$, we have $lcp(T[j .. i], T[b(F_{i-1})]) ≤ lcp(T[b(F_{i-1})], T[b(F_{i-1} + 1 .. i)]) = |F_i| + 1 = 2i$. Hence, $PLCP[b(F_i)] = 2i$ for each $i$ with $1 ≤ i ≤ m - 1$. Similarly, we obtain $PLCP[b(F_i)] = 2i - j$ for each $j$ with $0 ≤ j ≤ |F_i|$ and $PLCP[e(F_i)] = 2i$ for each $i$ with $0 ≤ i ≤ m - 1$. We conclude that the text positions $b(F_i)$ are interesting peaks, for $1 ≤ i ≤ m - 1$. Moreover, $b(F_{m-1})$ is a maximum peak, since $T[b(F_{m-1})] = σ_1$ occurs only at $T[b(F_{m-1})]$ and at the last text position $e(F_1)$ such that $PLCP[b(F_{m-1})] = 1$.

Finally, the algorithm collects $m - 2$ interesting peaks before finding the maximal peak at text position $b(F_{m-1})$. Since $|F_i| = 2i - 1$, we have $\sum_{i=1}^{m} |F_i| = \sum_{i=1}^{m} (2i - 1) = n$, which holds for $m = Θ(\sqrt{n})$. □

4 Decompression

The task of decompressing a bidirectional scheme is to resolve each reference $(src_i, \ell_i)$ of a referencing factor $T[dst_i .. dst_i + \ell_i - 1]$, i.e., to copy the characters from $T[src_i .. src_i + \ell_i - 1]$ to $T[dst_i .. dst_i + \ell_i - 1]$. 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 4.1 (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 $i ≠ j$ if $F_i$ 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 Fig. 4 can be seen in Fig. 7.

Bidirectional decompressors face two challenges arising from this graph structure:

(C1) Long dependency chains (i.e., large values of $d(G)$) may affect the time and space complexity of decompression algorithms.

(C2) The existence of multi-dependent nodes disallows efficient tree-based approaches.

In the remaining of this section, we present three strategies of attacking these issues, first individually (Section 4.1 and 4.2), and then together (Section 4.3). We focus on the resolution of indirect dependencies to obtain a dependency graph in which all references are direct children of literal factors. After such a resolution, the text can be trivially recovered with sort($n$) I/Os.
Figure 7: The dependency graph (left) and its EM representation (right) of the factorization given in Fig. 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).

Figure 8: Compaction of a bidirectional scheme. Left: The factors of the input are represented by maximal consecutive blocks of the same shading. In this example, the input consists of six factors. Referencing factors store no characters, have a light shading and an out-going arrow pointing to a vertical bar representing its corresponding reference. The first factor refers to two factors and is not resolved during compaction. The third factor refers to the fifth which refers to the sixth; this chain is compacted by redirecting the third factor to the sixth directly. Middle and Right: Dotted edges indicate dependencies with no corresponding edge in the algorithms described in Section 4.2.

4.1 Decompressor scan
The decompressor scan was introduced in [6, Sect. 3.2.2] (to which we refer for a detailed description). In its main phase, scan avoids multi-dependencies by splitting each reference \((src, \ell)\) with \(\ell > 1\) into references \((src, 1), \ldots, (src + \ell - 1, 1)\), i.e., one for each character. Then any undecoded position refers to either a literal factor or another reference. Hence the underlying dependency graph becomes a forest, which can conceptionally be resolved in \(O(n)\) time using standard traversal techniques. The initial splitting may however increase the number of references by a factor of \(O(n)\) causing inefficiencies and a significant memory overhead (which scan tries to reduce heuristically by preprocessing). This strategy is also similar to the parallel LZ77 decompressor of Farach and Muthukrishnan [7, Sect. 4.2].

4.2 Optimizing the Coding for Decompression
Orthogonally, we present the novel approach IM-Compact to improve an existing bidirectional coding for decompression by shortening dependency chains (see the left sub-figure of Fig. 8). This approach neither changes the factorization nor does it convert a referencing factor into a literal. It may be used directly after the compression step to accelerate future decompression.

Given a coding, we construct its dependency graph \(G\) but omit all multi-dependencies. As a result, we obtain a forest in which each reference depends only on a unique predecessor as illustrated in Fig. 8 (middle). Using a top-down traversal (e.g., BFS) on each tree individually, we can replace all chains with direct references to the root. Building \(G\) and traversing it requires \(O(b)\) total time.

We now present EM-Compact, an I/O-optimal variant of IM-Compact:

Step 1: We first construct a representation of the dependency graph consisting of two EM vectors requests and factors. Intuitively, each reference (child) sends a request message to the first factor it refers to (parent). Addressing is implemented indirectly in terms of text positions rather than factor indices. To this end, for each reference \((src, \ell)\) corresponding to a factor \(F_i = T[dst .. dst + \ell - 1]\), we push
(i) the tuple \((\text{src}, \ell, i)\) into \text{requests}, and (ii) the tuple \((\text{dst}, \ell, i)\) into \text{factors}. Additionally, each literal factor \(F_i = T[\text{dst} \ldots \text{dst} + \ell - 1]\) contributes a tuple \((\text{dst}, \ell, i)\) to \text{factors}. Subsequently, we sort\(^1\) both vectors independently, bringing the messages in \text{requests} and the recipients in \text{factors} into the same order.

**Step 2**: We now scan through \text{factors} and \text{requests} simultaneously. By doing so, each \(F_i\) in \text{factors} can gather all its children (requests): a factor \(F_i\) with tuple \((\text{dst}, \ell, i)\) has a child \((\text{src}, \ell', i')\) if \(\text{src} \in [\text{dst}, \text{dst} + \ell]\). The factor of this child \(F_i\) is completely contained in \(F_i\) if \(\text{src} + \ell' \leq \text{dst} + \ell\). Otherwise, \(F_i\) is multi-dependent. In contrast to \text{IM-Compact}, which discards such a multi-dependency completely, \text{EM-Compact} retains one edge to obtain a connected dependency tree simplifying \text{Step 3}\(^1\). To complete the tree, we add a virtual node \(v\) and assign all literal factors as \(v\)'s children. The resulting graph is a tree rooted in \(v\) with \(b+1\) nodes as illustrated in Fig. 9 (right). Its construction requires sort\((b)\) I/Os\(^2\).

**Step 3**: Subsequently, we apply the Euler tour technique and list ranking \cite{26} Sect. 3.6] on the tree built in \text{Step 2} to calculate the depths of all nodes, triggering sort\((b)\) I/Os.

**Step 4**: With an additional tree traversal, we can finally update the referred positions. For that, we annotate each tuple in \text{factors} and \text{requests} with the depth of its corresponding node. Then the vectors are sorted by the depths of their items and, in case of equality, the order used in \text{Step 1}. Similarly to \text{Step 2}, we scan both vectors simultaneously to traverse the dependency tree.

Due to the order of both vectors, \text{EM-Compact} processes nodes layer-wise and within each layer from left-to-right. Thus, parents are processed before their children, and can inductively forward their referred-to positions to their children. Following the time-forward processing \cite{26} Sect. 3.4] technique, we transport those updates as messages in an EM priority queue \(\text{PQ}\).

When processing node \(F_i\) at depth \(d\), we check whether a message of the form \((\text{dst}, \text{src}, 1)\) is at the top of \(\text{PQ}\). If so, we dequeue it and update the referred position of \(F_i\) to \(\text{src} + \text{src} - \text{dst} + 1\), where \(\text{src}\) is the former referred position of \(F_i\) as illustrated in Fig. 8. In any case, we iterate over all non multi-dependent children: for each \(F_j\), we push the message \((d+1, j, \text{src}, 1)\) into \(\text{PQ}\).

During each step, \(O(b)\) items are sorted and scanned, triggering sort\((b)\) I/Os in total. I/O-optimality follows by a reduction to the permutation problem analogously to the construction in \cite{2} Thm. 1.

### 4.3 Decompressor EM-PJ

Our novel decompressor \text{EM-PJ} (refer to Section 4.4 for details) adapts the ideas of the coding optimizers \text{IM-Compact} and \text{EM-Compact} for decompression. While \text{EM-Compact} is \(1/\text{O-optimal}, its resolution phase relies on the fact that we can efficiently find a topological order of the dependency tree. Unfortunately, this is not the case for general DAGs induced by factorizations with multi-dependencies.

We switch to the pointer jumping technique \cite{15} Sect. 2.2] for dependency resolution. Let \(G\) be the dependency graph of the factorization \(T = F_1 \cdots F_b\). As a starter, we assume that all factors are single-dependent, i.e., each node \(v\) representing a referencing factor has exactly one outgoing

\(^2\)To sort tuples we always use lexicographic order, i.e., we order tuples as implied by the first unequal element.

\(^1\)\text{EM-Compact} keeps multi-dependent nodes despite its inability to optimize them. It does so because a subtree rooted in a multi-dependent node can contain optimizable dependency chains.

\(^3\)Depending on the encoding of the input, a scan over the content of literal factors may be necessary and trigger scan\((n)\) I/Os.
edge \((v, p(v))\). For all other nodes (representing literal factors) we define \(p(v) := v\). Clearly, like in EM-Compact, \(G\) 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. Fig. 10). 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 \(\Theta(\lfloor\log d(G)\rfloor)\) 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 Section 4.4.

Like in EM-Compact, we 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 text positions rather than factor indices. For each reference \((\text{src}, \ell)\) corresponding to a factor \(F_i = T[\text{dst} \ldots \text{dst} + \ell - 1]\), we push \((\text{dst}, \ell, \text{src})\) into requests and \((\text{dst}, \ell, \text{dst})\) 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 both vectors independently, bringing the messages in requests and the recipients in factors into the same order. On the right side of Fig. 7 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 \((\text{dst}, \ell, \text{src})\) 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 \((\text{dst}, \ell, \text{src})\) depending on the following cases (cf. Fig. 11):

**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.
Theorem 4.2. Let the list of factors represented by their coding. We process the references to resolve all references, where \( src \) refers to a substring that can be set to the referred position of the reference it refers to. Suppose that a number of sub-references, where each sub-reference either (a) can be immediately decoded or (b) has previously explained, this strategy is based on the optimization technique for improving the decompression \( EM-PJ \). For completeness, we present a more technical representation of our decompression strategy.

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

**Theorem 4.2.** Let \( F_1 \cdots F_b = T \) be a \( \xi \)-restricted bidirectional scheme, and \( d(G) < b \) be the depth of \( T \)'s dependency graph \( G \). Then \( EM-PJ \) requires \( O(\log d(G)) \) \( \text{sort}(n/\xi) \) I/Os.

**Proof.** As pointer jumping halves the depth of the dependency graph \( G \), \( EM-PJ \) performs \( O(\log d(G)) \) iterations. While \( G \) changes in each step, it remains valid in terms of Definition 3.1. Despite \( EM-PJ \) introducing new nodes by splitting a factor, the number \( |V| \) of nodes of \( G \) is bounded by the maximal number \( k := \Theta(n/\xi) \) of factors, i.e., \( |V| = O(k) \).

Hence the size of the vectors involved is bounded as follows: factors is filled once where each factor contributes at most one element, thus \( |\text{factors}| = O(b) = O(k) \). \( \text{requests} \) is reproduced in each iteration and may reach up to \( |\text{requests}| = O(|V|) = O(k) \) items, which directly translates into an upper bound on the number of items in \( PQSplit \). Analogously, \( |\text{result}| \) is bounded from above by \( O(k) \). Thus in each round \( O(k) \) items are sorted and scanned a constant number of times, resulting in the claimed bound. \( \square \)

### 4.4 Detailed Description of \( EM-PJ \)

For completeness, we present a more technical representation of our decompression strategy \( EM-PJ \). As previously explained, this strategy is based on the optimization technique for improving the decompression of a coding described in Section 4.2. The difference is that \( EM-PJ \) splits references up in a minimal number of sub-references, where each sub-reference either (a) can be immediately decoded or (b) has a referred position that can be set to the referred position of the reference it refers to. Suppose that a reference \( r \) refers to a substring \( S \) that is not completely decompressed. We split \( r \) into sub-references such that a sub-reference \( (src_0, \ell_0) \) refers to a substring \( T[\text{src}_0 \ldots \text{src}_0 + \ell_0 - 1] \) that is either

- already decompressed, or
- contained in a substring \( T[\text{dst}_1 \ldots \text{dst}_1 + \ell_1 - 1] \) with \( \text{dst}_1 \leq \text{src}_0 \leq \text{src}_0 + \ell_0 - 1 \leq \text{dst}_1 + \ell_1 - 1 \) substituted by a reference \( (\text{src}_1, \ell_1) \), cf. Fig. 9.

In the former case, we can resolve the sub-reference. In the latter case, we exchange \( (\text{src}_0, \ell_0) \) with \( (\text{src}_1 + \text{src}_0 - \text{dst}_1, \ell_0) \). Due to the pointer jumping technique, we need \( O(\log d) = O(\log n) \) scans of the references to resolve all references, where \( d \leq n \) is the maximal depth a reference can have.

The strategy \( EM-PJ \) maintains the following lists in external memory:

- the list of requests \( L_{\text{req}} \) storing tuples \( (\text{src}, \text{dst}, \ell) \) to maintain the information that we request the substring \( T[\text{src} \ldots \text{src} + \ell - 1] \) to restore \( T[\text{dst} \ldots \text{dst} + \ell - 1] \),
- the list of references \( L_{\text{ref}} \) storing tuples \( (\text{dst}, \text{src}, \ell) \) corresponding to unresolved referencing factors for applying the pointer jumping technique, and
- the list of resolutions \( L_{\text{res}} \) storing tuples \( (\text{dst}, S) \) with \( S \in \Sigma^* \) for the instruction to copy the string \( S \) to \( T[\text{dst} \ldots \text{dst} + |S| - 1] \).

**Initial Step** We create an external file \( T \) with the length of the original text and scan sequentially the list of factors represented by their coding. We process the \( x \)-th factor \( F_x \) as follows:

- If \( F_x \) is a literal factor, copy its contents to \( T[1 + |F_1 \cdots F_{x-1}|] \).
- Otherwise, \( F_x \) is a referencing factor. Given its reference is \( (\text{src}, \ell) \), store \( (\text{dst}, \text{src}, \ell) \) in \( L_{\text{ref}} \), and \( (\text{src}, \text{dst}, \ell) \) in \( L_{\text{req}} \), where \( \text{dst} \) is the starting position of \( F_x \).
Case 1: \( \text{src}_0 < \text{dst}_1 \). In this case, \( T[\text{src}_0 \ldots \text{dst}_1] \) is already resolved, and we insert \( (\text{dst}_0, T[\text{src}_0 \ldots \text{dst}_1]) \) into the new resolution list \( L_{\text{res}}^{\text{new}} \). We update the request \( (\text{src}_0, \text{dst}_0, \text{src}_0 - \text{dst}_0, \ell_0) \) to \( (\text{dst}_1, \text{dst}_0 + \text{dst}_1 - \text{src}_0, \ell_0 - \text{dst}_1 + \text{src}_0) \), and proceed with Case 2.

Case 2: \( \text{src}_0 \geq \text{dst}_1 \). If \( \text{dst}_1 + \ell_1 - \text{src}_0 \geq \ell_0 \), then we can pointer jump the request \( (\text{src}_0, \text{dst}_0, \ell_0) \) to \( (\text{src}_1 + \text{src}_0 - \text{dst}_1, \text{dst}_0, \ell_0) \). Otherwise, we split the request in two requests \( (\text{src}_0, \text{dst}_0, \text{dst}_1 + \ell_1 - \text{src}_0, \ell_0 - \text{dst}_1 + \ell_1 - \text{src}_0) \) and \( (\text{dst}_1 + \ell_1, \text{dst}_0 + \text{dst}_1 + \ell_1 - \text{src}_0, \ell_0 - \text{dst}_1 - \ell_1 + \text{src}_0) \). We can pointer jump the first request to \( (\text{src}_1 + \text{src}_0 - \text{dst}_1, \text{dst}_0 + \text{dst}_1 + \ell_1 - \text{src}_0) \).

In both cases, when creating a new request \( (\text{src}, \text{dst}, \ell) \), we insert it into the new request list \( L_{\text{req}}^{\text{new}} \) and insert \( (\text{dst}, \text{src}, \ell) \) into the new reference list \( L_{\text{ref}}^{\text{new}} \).

After the scan, we move the contents of the new lists \( L_{\text{req}}^{\text{new}}, L_{\text{ref}}^{\text{new}}, \) and \( L_{\text{res}}^{\text{new}} \) to their corresponding lists \( L_{\text{req}}, L_{\text{ref}}, \) and \( L_{\text{res}} \), respectively. We repeat this process until the list of references \( L_{\text{ref}} \) becomes empty.

Due to practical issues, we did not implement the lists of resolutions \( L_{\text{res}} \) and \( L_{\text{res}}^{\text{new}} \) as explained, since an entry of these lists would hold a string of arbitrary length. Instead, we use a single list \( \hat{L}_{\text{res}} \) storing tuples \( (\text{src}, \text{dst}, \ell) \) saying that the substring \( T[\text{src} \ldots \text{src} + \ell - 1] \) is already decoded and can be copied to \( T[\text{dst} \ldots \text{dst} + \ell - 1] \). To avoid random I/O, we first process a request \( (\text{src}, \text{dst}, \ell) \in L_{\text{req}} \) completely with Case 1 and Case 2 before selecting the next request \( (\text{src}', \text{dst}', \ell') \in L_{\text{req}} \). However, both requests can overlap with \( \text{src} \leq \text{src}' \leq \text{src} + \ell \) such that \( \hat{L}_{\text{res}} \) can become unsorted (cf. Fig. 13). We sort \( \hat{L}_{\text{res}} \) after a scan of all requests according to its first component. Subsequently, we scan the text and \( \hat{L}_{\text{res}} \) to produce, given a tuple \( (\text{src}, \text{dst}, \ell) \in \hat{L}_{\text{res}} \), the tuples \( (\text{dst} + i, T[\text{src} + i]) \) for all integers \( i \) with \( 0 \leq i \leq \ell - 1 \). These tuples are sorted by their first component. With a linear scan on \( T \), we set \( T[\text{dst}] \leftarrow c \) for each such tuple \( (\text{dst}, c) \).
Figure 13: Unordered insertion into $\hat{L}_{\text{req}}$. Suppose that the first tuples in the request list $L_{\text{req}}$ are $(\text{src}_0, \text{dst}_0, \ell_0)$ and $(\text{src}', \text{dst}', \ell')$ with $\text{src}_0 \leq \text{src}'$ and that the first tuples in the reference list $L_{\text{ref}}$ are $(\text{dst}_1, \text{src}_1, \ell_1)$ and $(\text{dst}_2, \text{src}_2, \ell_2)$ with $\text{src}' < \text{dst}_1 \leq \text{src}' + \ell' - 1$. Since we first process $(\text{src}_0, \text{dst}_0, \ell_0)$ and its resulting sub-requests by Cases 1 and 2, we produce the resolution to copy $T[\text{dst}_1 + \ell_1 .. \text{dst}_2 - 1]$ to $T[\text{dst}_0 + \text{dst}_1 + \ell_1 - \text{src}_0 .. \text{dst}_0 + \text{dst}_2 - 1 - \text{src}_0]$ prior to producing the resolution to copy $T[\text{src}' .. \text{dst}_1 - 1]$ to $T[\text{dst}' .. \text{dst}' + \text{dst}_1 - 1 - \text{src}'].$

| commoncrawl | | | | | | |
|-------------|----|----|----|----|----|----|
| prefix length | $H_0$ | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ |
| 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 | 5.99172 | 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 |

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

5 Practical Evaluation

We embedded our algorithms in the C++ framework tudocomp, available at https://github.com/tudocomp/tudocomp. We collected the used EM algorithms for constructing the needed text data structures in the repository https://github.com/tudocomp/emtools.

Experimental Setup Our experiments are conducted on a machine with 16 GiB of RAM, 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 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 with the compile flags -O3 -march=native -DNDEBUG.

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 [20] 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.

\
\footnote{In order to avoid swapping, each experiment was conducted with a limit of 14 GiB of RAM.}
In early experiments with EM-LZscan [20], 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 [20] 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 and use it for computing PLCP with the parallel EM construction algorithm of Kärkkäinen and Kempa [17]. We modified the source code of the latter to also produce Φ as a side product. This chain of algorithms is illustrated in Fig. 14.

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 [16] 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 [21]). 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^k GiB prefixes (4 ≤ k ≤ 7) of our data sets DNA and COMMONCRAWL. We collected the median of three iterations and present the results in Fig. 15. 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.

6 https://www.cs.helsinki.fi/u/dkempa/pem_bwt.html
Decompression  We ran our decompressor implementation on the plcpcomp codings of our datasets. Plots of the scaling experiments are shown in Fig. 16. 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.

In Fig. 17, we measured the impact of the choice of $\xi$ on the compressed output and the decompression algorithm of our datasets. For larger values of $\xi$, 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 Belazzougui et al. [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 [4]), 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.
Figure 17: Evaluation of plcpcomp with different threshold values $\xi$.

[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.

[7] Martin Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In Proc. SPAA, pages 244–253, 1995.

[8] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. Algorithmica, 63(3):707–730, 2012.

[9] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. SIAM J. Comput., 42(4):1521–1541, 2013.

[10] Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. Algorithmica, 80(7):2048–2081, 2018.

[11] 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.

[12] J. K. Gallant. String compression algorithms. PhD thesis, Princeton University, 1982.

[13] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In Proc. DCC, pages 133–142, 2013.
[14] Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In Proc. DCC, pages 163–172, 2014.

[15] Joseph Jájá. An Introduction to Parallel Algorithms. Addison-Wesley, 1992.

[16] 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.

[17] 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.

[18] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permutated longest-common-prefix array. In Proc. CPM, volume 5577 of LNCS, pages 181–192, 2009.

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

[21] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In Proc. CPM, volume 9133 of LNCS, pages 329–342, 2015.

[22] Juha Kärkkäinen, Dominik Kempa, and Marcin Piatkowski. Tighter bounds for the sum of irreducible LCP values. Theor. Comput. Sci., 656:265–278, 2016.

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

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

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

[26] Anil Maheshwari and Norbert Zeh. A survey of techniques for designing I/O-efficient algorithms. In Algorithms for Memory Hierarchies, Advanced Lectures, pages 36–61, 2002.

[27] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. SIAM J. Comput., 22(5):935–948, 1993.

[28] Markus Mauer, Timo Beller, and Enno Ohlebusch. A Lempel-Ziv-style compression method for repetitive texts. In Proc. PSC, pages 96–107, 2017.

[29] 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.

[30] Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. O(n log n)-time text compression by LZ-style longest first substitution. In Proc. PSC, pages 12–26, 2018.

[31] Takaaki Nishimoto and Yasuo Tabei. LZRR: LZ77 parsing with right reference. arXiv 1812.04261, 2018.

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

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