2017 Data Compression Conference

LZ-End Parsing in Compressed Space

Dominik Kempa and Dmitry Kosolobov

Department of Computer Science,
University of Helsinki, Finland
dominik.kempa@cs.helsinki.fi, dkosolobov@mail.ru

Abstract

We present an algorithm that constructs the LZ-End parsing (a variation of LZ77) of a given string of length n in $O(n \log \ell)$ expected time and $O(z + \ell)$ space, where z is the number of phrases in the parsing and ℓ is the length of the longest phrase. As an option, we can fix ℓ (e.g., to the size of RAM) thus obtaining a reasonable LZ-End approximation with the same functionality and the length of phrases restricted by ℓ. This modified algorithm constructs the parsing in streaming fashion in one left to right pass on the input string w.h.p. and performs one right to left pass to verify the correctness of the result. Experimentally comparing this version to other LZ77-based analogs, we show that it is of practical interest.

Introduction

The growth of the amount of highly compressible data in modern applications has accelerated the development of new compression algorithms working in space comparable to the size of their compressed input. The compression schemes based on the famous LZ77 algorithm [15] have proved their extreme efficiency in compressing highly repetitive collections of genomes, logs, and repositories of version control systems. For such data, most other methods achieve significantly worse results. Unfortunately, the problem of the construction of LZ77-based schemes in small space and reasonable time is still very challenging (e.g., see [2, 5, 7, 11, 13] and references therein).

In this paper we consider a variant of LZ77 called LZ-End that was introduced in [14, 12]. This scheme is comparable in practice to LZ77 in the sense of compression quality (see [12]) but, in addition, allows to efficiently retrieve any substring of the compressed string (when equipped with an extra lightweight structure). The LZ-End construction algorithm presented in [14] builds the LZ-End parsing of a string of length n in $O(n)$ space, which is unacceptable for large inputs that do not fit in main memory. To our knowledge, there were no further improvements of this result.

We present an algorithm that constructs the LZ-End parsing of the input string of length n in $O(n \log \ell)$ time w.h.p. (throughout the paper all logarithms have base 2) and $O(z + \ell)$ space, where z is the number of phrases in the parsing and ℓ is an upper bound on the length of a phrase. Further, we modify this algorithm fixing ℓ in advance (e.g., to the size of main memory) and construct in $O(z + \ell)$ space an approximation of the LZ-End parsing in which all phrases have length less than ℓ. We implement this version and experimentally show that it is of practical interest.

Recently, in [5] an algorithm was presented that constructs an approximation of LZ77 and possesses similar space and time characteristics. However, unlike the

2375-0359/17 $31.00 © 2017 IEEE
DOI 10.1109/DCC.2017.73

algorithm of [5], ours does not require random access to the input and constructs the parsing in one left to right pass in expectation plus one right to left pass to verify that the parsing is correct, which is a good property in the external memory setting.

Preliminaries. Let s be a string of length $|s| = n$. We write $s[i]$ for the ith letter of s and $s[i..j]$ for $s[i]s[i{+}1]\cdots s[j]$. The *reversal of* s is the string $\overleftarrow{s} = s[n]\cdots s[2]s[1]$. For any i, j, the set $\{k \in \mathbb{Z}\colon i \le k \le j\}$ is denoted by $[i..j]$; denote $[i..j) = [i..j] \setminus \{j\}$ and $(i..j] = [i..j] \setminus \{i\}$. Our notation for arrays is similar: e.g., $a[i..j]$ denotes an array indexed by the numbers $[i..j]$. Let h be a hash table mapping an integer set $S \subset \mathbb{N}$ into a set T. For $x \in \mathbb{N}$, denote by $h(x)$ the image of x assuming $h(x) = \mathbf{nil}$ if $x \notin S$.

The *LZ-End parsing* [12] of a string s is a decomposition $s = f_1 f_2 \cdots f_z$ constructed as follows: if we have already processed a prefix $s[1..k] = f_1 f_2 \cdots f_{i-1}$, then $f_i[1..|f_i|{-}1]$ is the longest prefix of $s[k{+}1..|s|]$ that is a suffix of a string $f_1 f_2 \cdots f_j$ for $j < i$; the substrings f_i are called *phrases*. For instance, the string $ababaaaaaac$ has the LZ-End parsing $a.b.aba.aa.aaac$. Then, the following lemma is straightforward.

Lemma 1. *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a string. Then, for any $i \in [1..z)$, any proper prefix of length at least $|f_i|$ of the string $f_i f_{i+1} \cdots f_z$ cannot be a suffix of a string $f_1 f_2 \cdots f_j$ for $j < i$.*

Basic Observations

The definition of the LZ-End parsing easily implies the following observation suggesting a way how to perform the construction of the LZ-End parsing incrementally.

Lemma 2. *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a string s. If i is the maximal integer such that the string $f_{z-i} f_{z-i+1} \cdots f_z$ is a suffix of a string $f_1 f_2 \cdots f_j$ for $j < z - i$, then, for any letter a, the LZ-End parsing of the string sa is $f'_1 f'_2 \cdots f'_{z'}$, where $z' = z - i$, $f'_1 = f_1, f'_2 = f_2, \ldots, f'_{z'-1} = f_{z'-1}$, and $f'_{z'} = f_{z-i} f_{z-i+1} \cdots f_z a$.*

It turns out, however, that the number of phrases that might "unite" into a new phrase when a letter has been appended (as in Lemma 2) is severely restricted.

Lemma 3. *If $f_1 f_2 \cdots f_z$ is the LZ-End parsing of a string s, then, for any letter a, the last phrase in the LZ-End parsing of the string sa is 1) $f_{z-1} f_z a$ or 2) $f_z a$ or 3) a.*

Proof. By Lemma 2, the last LZ-End phrase of sa is fa, where $f = f_{z-i} f_{z-i+1} \cdots f_z$ for some i. Suppose, to the contrary, that $i > 1$. By the definition of LZ-End, there is $j < z - i$ such that f is a suffix of $f_1 f_2 \cdots f_j$. If $|f_j| \le |f_{z-1} f_z|$, then f has a proper prefix of length $|f| - |f_j| \ge |f_{z-i}|$ that is a suffix of $f_1 f_2 \cdots f_{j-1}$, which contradicts Lemma 1. If $|f_j| > |f_{z-1} f_z|$, then there is $j' < j$ (since $|f_j| > 1$) such that $f_j[1..|f_j|{-}1]$ is a suffix of $f_1 f_2 \cdots f_{j'}$ and, hence, the prefix of length $|f_{z-1} f_z| - 1$ of the string $f_{z-1} f_z$ is a suffix of $f_1 f_2 \cdots f_{j'}$, which again contradicts Lemma 1. $\square$

Let s be the input string of our algorithm and $n = |s|$. The basic idea is to read s from left to right and compute the LZ-End parsing for each prefix of s using a compressed trie storing all reversed prefixes of s ending at the phrase boundaries of the current parsing: To extend the current prefix by a letter and rebuild the parsing, we check using the trie whether the last one or two phrases have previous occurrences ending at a phrase boundary; then, according to Lemmas 2 and 3, we unite zero, one, or two last phrases with the appended letter and thus obtain a new phrase.

This approach seems promising since the trie can be stored in $O(z)$ space, where z is the number of phrases in the current parsing. Unlike LZ77, however, the LZ-End parsing of a prefix of s can have more phrases than the parsing of s (e.g., $a.b.abb.ba.bb$ and $a.b.abb.babbc$). The following lemma shows that the parsing of a prefix cannot have too many phrases. (The technical proof of this is in the full version [10].)

Lemma 4 ([10])**.** *Denote by z and z', respectively, the numbers of phrases in the LZ-End parsing of strings s and s' such that s' is a prefix of s. Then $3z \ge z'$.*

For a string t, define as $\mathsf{hash}(t) = \sum_{i=1}^{|t|} t[i]\alpha^{i-1} \bmod \mu$ the *Karp–Rabin fingerprint* (e.g., see [4]) of t, where μ is a fixed prime such that $\mu \ge n^{c+4}$ for some $c \ge 1$, and $\alpha \in [0..\mu)$ is chosen uniformly at random during the initialization of the algorithm. Denote $\mathsf{lhash}(t) = \mathsf{hash}(\overleftarrow{t})$. It is well known that the probability that two different substrings of s have the same fingerprints is less than $\frac{1}{n^c}$; such situation is called a *false positive*. Hereafter, we assume that there are no false positives to avoid repeating that the answers are correct with high probability. In the sequel we describe how to verify whether the constructed parsing really encodes the string s.

Fast Compressed Trie

Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a prefix of s that has just been calculated by our incremental algorithm. Our algorithm maintains a compressed trie T containing the reversed prefixes $\overleftarrow{f_1}, \overleftarrow{f_1 f_2}, \ldots, \overleftarrow{f_1 f_2 \cdots f_i}$ up to some specified index i. For each vertex v of T, denote by $v.par$ the parent of v (if any) and by $v.str$ the string written on the path connecting the root and v (note that $v.par$ and $v.str$ are used only in discussions). Each vertex v of T contains the following fields: $v.len$, the length of $v.str$; $v.map$, a hash table that, for any child u of v, maps the letter $a = u.str[v.len{+}1]$ to $u = v.map(a)$; $v.phr$, a number such that $v.str$ is a prefix of the string $\overleftarrow{f_1 f_2 \cdots f_{v.phr}}$.

Define $\mathsf{rst}(x, i) = x \mathbin{\&} \neg(2^i - 1)$ (resetting i least significant bits). For each non-root vertex v in T, denote $p_v = \mathsf{rst}(v.len, i)$ for the maximal i such that $\mathsf{rst}(v.len, i) > v.par.len$ and denote $h_v = \mathsf{hash}(v.str[1..p_v])$. For fast navigation in T, we maintain a hash table nav that, for each non-root vertex v, maps the pair (p_v, h_v) to $v = \mathsf{nav}(p_v, h_v)$. The table nav allows us to parse the trie T as follows (see Lemma 5):

function $\mathsf{approxFind}(pat)$
  $p \leftarrow 0,\ v \leftarrow root$;
  for $i \leftarrow \lceil \log |pat| \rceil;\ i \ge 0;\ i \leftarrow i - 1$ **do**
    if $v.len \ge p + 2^i$ **then** $p \leftarrow p + 2^i$;
    else if $\mathsf{nav}(p + 2^i, \mathsf{hash}(pat[1..p + 2^i])) \ne$ **nil then** $p \leftarrow p + 2^i,\ v \leftarrow \mathsf{nav}(p, \mathsf{hash}(pat[1..p]))$;
  if $v.map(pat[v.len + 1]) \ne$ **nil then** $v \leftarrow v.map(pat[v.len + 1])$;
  return v;

Our method resembles the so called fat binary search in z-tries introduced in [1] and the proof of its correctness in Lemma 5 is essentially the same.

Lemma 5 (see [1, 10])**.** *Denote by t the longest prefix of pat that is represented in the trie T. If $|t| = 0$, then $\mathsf{approxFind}(pat)$ returns the root of T; otherwise, it returns a vertex v such that t is a prefix of $v.str$ and either $v.par.len < |t|$ or $v.par.par.len < |t|$.*

Algorithm

Let us first describe an algorithm with a parameter ℓ such that ℓ is an upper bound on the length of a phrase in the LZ-End parsing of the input string s. The algorithm scans s from left to right and builds the LZ-End parsing for each prefix of s. We store the number of phrases in the current parsing in a variable z and encode the parsing in an array $phrs[1..z]$ containing structures defined as follows: Suppose that $f_1 f_2 \cdots f_z$ is the parsing of the current prefix; then, for $i \in [1..z]$, we have $phrs[i].c = f_i[|f_i|]$, $phrs[i].len = |f_i|$, $phrs[i].hash = \mathsf{lhash}(f_i)$, and $phrs[i].lnk$ is a number such that $f_i[1..|f_i|{-}1]$ is a suffix of $f_1 f_2 \cdots f_{phrs[i].lnk}$ ($phrs[i].lnk$ is arbitrary if $phrs[i].len = 1$).

The algorithm reads s by portions of length ℓ; the processing of one portion is called a *phase*. In the beginning of the ith phase ($i \geq 1$) $phrs[1..z]$ encodes the parsing of the string $s[1..i\ell{-}\ell]$ and the trie T contains the reversed prefixes of s ending at positions $\sum_{j=1}^{k} phrs[j].len$ for all k such that $\sum_{j=1}^{k-1} phrs[j].len \leq i\ell - 2\ell$. Since the length of any phrase is at most ℓ, this guarantees that no prefix can be deleted from T due to the changes in the array $phrs[1..z]$ during the future work of the algorithm.

Lemma 6 (see [12])**.** *Suppose that the array $phrs$ encodes the LZ-End parsing $f_1 f_2 \cdots f_z$. Then, for any $j \in [1..z]$ and k, using $phrs$, one can retrieve the suffix of length k of the string $f_1 f_2 \cdots f_j$ in $O(k)$ time.*

During the ith phase, we maintain integer arrays $lnks[i\ell - 2\ell..i\ell]$ and $lens[i\ell - 2\ell..i\ell]$ defined as follows. Let m denote the length of the current prefix ($m = i\ell - \ell$ at the beginning of the phase). For each $j \in [\max\{1, i\ell{-}2\ell\}..i\ell]$, denote by $f_{j,1} f_{j,2} \cdots f_{j,z_j}$ the LZ-End parsing of $s[1..j]$. Then, for each $j \in [\max\{1, i\ell - 2\ell\}..m]$, we have $lens[j] = |f_{j,z_j}|$ and the number $lnks[j]$ is such that $lnks[j] \in [1..z_j)$, $f_{j,z_j}[1..|f_{j,z_j}|{-}1]$ is a suffix of the string $f_{j,1} f_{j,2} \cdots f_{j,lnks[j]}$, and $\overleftarrow{f_{j,1} f_{j,2} \cdots f_{j,lnks[j]}}$ is contained in the trie T, or we have $lnks[j] = \textbf{nil}$ if there is no such number or $lens[j] = 1$ or $j \notin [1..m]$.

Define a function $\mathsf{nca}(z_1, z_2)$ that, for given z_1 and z_2, returns the nearest common ancestor of the leaves of T corresponding to $\overleftarrow{f_1 f_2 \cdots f_{z_1}}$ and $\overleftarrow{f_1 f_2 \cdots f_{z_2}}$, where $f_1 f_2 \cdots f_z$ is the LZ-End parsing of the current prefix, or returns **nil** if one of these strings is not in T. We maintain on T the structure of [3] that takes $O(z)$ space and can compute nca in $O(1)$ time using an array $N[1..z]$ such that $N[z']$ stores the leaf of T corresponding to $\overleftarrow{f_1 f_2 \cdots f_{z'}}$; N is easily modified when a leaf is inserted or deleted.

Denote $s_k{=}s[i\ell{-}3\ell..k]$. We begin the ith phase computing for the string $\overleftarrow{s}_{i\ell}$ by standard algorithms (see [4]) the *suffix array* SA, its *inverse* $\overleftarrow{SA}$, and the array $lcp[1..3\ell]$ that are defined as follows: $SA[0..3\ell]$ is a permutation of $[i\ell - 3\ell..i\ell]$ such that $\overleftarrow{s}_{SA[0]} < \overleftarrow{s}_{SA[1]} < \cdots < \overleftarrow{s}_{SA[3\ell]}$, $\overleftarrow{SA}[i\ell - 3\ell..i\ell]$ is such that $q = SA[\overleftarrow{SA}[q]]$, and $lcp[q]$ contains the length of the longest common prefix of $\overleftarrow{s}_{SA[q-1]}$ and $\overleftarrow{s}_{SA[q]}$. We equip lcp with the *range minimum query (RMQ)* structure (e.g., see [4]) that uses $O(\ell)$ space and allows us to find the minimum in any range of lcp in $O(1)$ time. Then, we build an array $hs[1..3\ell]$ such that $hs[j] = \mathsf{lhash}(s_{i\ell-j})$. All this takes $O(\ell)$ time.

In addition, we maintain a balanced tree P of size $O(\ell)$ that allows us to compute the maximum $\max\{k \in [1..z]\colon \sum_{j=1}^{k} phrs[j].len \leq x\}$ for any $x \in [i\ell{-}3\ell..i\ell]$ in $O(\log \ell)$ time. Finally, we construct a marked perfect binary tree M with leaves

Table 1: A summary of all described structures.

fields of $phrs$ elem:	$phrs[j].c$, $phrs[j].len$, $phrs[j].hash$, $phrs[j].lnk$
fields of T vertex:	$v.len$, $v.map$, $v.phr$
structures of T:	hash table $\mathsf{nav}(p, h)$, dynamic nca structure on T
additional arrays:	$lnks$, $lens$, hs, SA, $\overleftarrow{SA}$, lcp, N
miscellaneous:	RMQ on lcp, tree P, binary tree M with leaves L

$L[0..3\ell]$ in which a leaf $L[j]$ is marked iff a phrase of the current parsing ends at position $SA[j]$, and an internal node of M is marked iff it has a marked child.

The ith phase (absorbTwo2, absorbOne2, updateRecent are discussed below):

1: **for** $m \leftarrow i\ell - \ell + 1$; $m \le i\ell$; $m \leftarrow m + 1$ **do**
2: $\quad len \leftarrow phrs[z].len + phrs[z-1].len$;
3: $\quad p \leftarrow \mathsf{approxFind}(\overleftarrow{s[m - len..m-1]}).phr$;
4: $\quad lnks[m] \leftarrow$ **nil**; ▷ the global variable ptr is set by absorbOne2 and absorbTwo2
5: $\quad$ **if** $len < \ell$ **and** $\mathsf{absorbTwo}(p, m)$ **then** $z \leftarrow z - 1$, $phrs[z].len \leftarrow len + 1$, $lnks[m] \leftarrow p$;
6: $\quad$ **else if** $len < \ell$ **and** $\mathsf{absorbTwo2}(m)$ **then** $z \leftarrow z - 1$, $phrs[z].len \leftarrow len + 1$, $p \leftarrow ptr$;
7: $\quad$ **else if** $phrs[z].len < \ell$ **and** $\mathsf{absorbOne}(p, m)$ **then** $phrs[z].len \leftarrow phrs[z].len + 1$, $lnks[m] \leftarrow p$;
8: $\quad$ **else if** $phrs[z].len < \ell$ **and** $\mathsf{absorbOne2}(m)$ **then** $phrs[z].len \leftarrow phrs[z].len + 1$, $p \leftarrow ptr$;
9: $\quad$ **else** $z \leftarrow z + 1$, $phrs[z].len \leftarrow 1$;
10: $\quad lens[m] \leftarrow phrs[z].len$;
11: $\quad phrs[z].c \leftarrow s[m]$, $phrs[z].hash \leftarrow \mathsf{lhash}(s[m - phrs[z].len + 1..m])$, $phrs[z].lnk \leftarrow p$;
12: $\quad \mathsf{updateRecent}()$;

1: **function** $\mathsf{absorbTwo}(p, m)$
2: $\quad$ **return** $\mathsf{commonPart}(p, m, phrs[z].len + phrs[z-1].len)$;

1: **function** $\mathsf{absorbOne}(p, m)$
2: $\quad$ **if** $phrs[p].len < phrs[z].len$ **then return** $\mathsf{commonPart}(p, m, phrs[z].len)$;
3: $\quad$ **if** $phrs[p].c \ne phrs[z].c$ **or** $(phrs[z].len > 1$ **and** $lnks[m-1] =$ **nil**$)$ **then return false**;
4: $\quad$ **if** $phrs[z].len = 1$ **then return true**;
5: $\quad$ **return** $\mathsf{nca}(lnks[m-1], phrs[p].lnk).len + 1 \ge phrs[z].len$;

1: **function** $\mathsf{commonPart}(p, m, len)$
2: $\quad$ **if** $phrs[p].len \ge len$ **or** $phrs[p].hash \ne \mathsf{lhash}(s[m - phrs[p].len..m-1])$ **then return false**;
3: $\quad pos = m - phrs[p].len$;
4: $\quad$ **if** $lens[pos] - 1 + phrs[p].len \ne len$ **or** $lnks[pos] =$ **nil then return false**;
5: $\quad$ **return** $\mathsf{nca}(lnks[pos], p - 1).len + phrs[p].len \ge len$;

It is easy to see that we compute lhash (and no hash) only for substrings of the string $s[i\ell..i\ell{-}3\ell]$. It is well known that, using the array hs and the precomputed powers $\alpha^1, \alpha^2, \ldots, \alpha^\ell$ modulo μ, one can compute $\mathsf{lhash}(s[j..j'])$ for any substring $s[j..j']$ of $s[i\ell{-}3\ell..i\ell]$ in $O(1)$ time. Further, since the length of any phrase is less than ℓ, we have $len \le 2\ell$ and, hence, we pass only reversed substrings of the string $s[i\ell{-}3\ell..i\ell]$ to approxFind. Therefore, the calculations of hash inside approxFind can also be performed in $O(1)$ time. Evidently, $\mathsf{approxFind}(pat)$ works in $O(\log |pat|)$ time. So, the phase processing works in overall $O(\ell \log \ell)$ time plus the time required for the functions absorbTwo2, absorbOne2, and updateRecent discussed below.

Lemma 7. *Suppose that $f_1 f_2 \cdots f_z$ is the LZ-End parsing of $s[1..m{-}1]$ encoded in $phrs[1..z]$ at the beginning of an iteration of the loop 1–12. Then, the function* absorbTwo *[*absorbOne*] in line 5 [7] returns true iff the string $f_{z-1} f_z$ [f_z] is a suffix of a string $f_1 f_2 \cdots f_j$ whose corresponding reverse $\overleftarrow{f_1 f_2 \cdots f_j}$ is in the trie T.*

Proof. Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of the string $s[1..m{-}1]$. Since we have $p = \mathsf{approxFind}(\overleftarrow{f}_z \overleftarrow{f}_{z-1}).phr$ (line 3), it follows from Lemma 5 that, if $f_{z-1}f_z$ is a suffix of a string $f_1 f_2 \cdots f_j$ whose corresponding reverse $\overleftarrow{f_1 f_2 \cdots f_j}$ is contained in the trie T, then $f_{z-1}f_z$ must be a suffix of the string $f_1 f_2 \cdots f_p$.

Consider first the functions $\mathsf{absorbTwo}$ and $\mathsf{commonPart}$. Suppose that $|f_p| \geq |f_{z-1}f_z|$. If $f_{z-1}f_z$ is a suffix of $f_1 f_2 \cdots f_p$, then $f_{z-1}f_z$ has a prefix of length $|f_{z-1}f_z|{-}1$ that is a suffix of $f_1 f_2 \cdots f_{phrs[p].lnk}$, which is impossible by Lemma 1. The code in line 2 verifies the condition $|f_p| < |f_{z-1}f_z|$ and checks whether f_p is a suffix of $f_{z-1}f_z$.

It remains to check whether the string $f' = s[m - |f_{z-1}f_z|..m - |f_p| - 1]$ is a suffix of $f_1 f_2 \cdots f_{p-1}$. Obviously, the LZ-End parsing of the string $s[1..m{-}|f_{z-1}f_z|{-}1]$ is $f_1 f_2 \cdots f_{z-2}$. So, by the definition of LZ-End, if f' is a suffix of $f_1 f_2 \cdots f_{p-1}$, then the string $s[1..m{-}|f_p|]$ has the parsing $f_1 f_2 \cdots f_{z-2} f''$, where $f'' = f's[m{-}|f_p|]$; therefore, by the definition of $lens$ and $lnks$, we have $lens[pos] = |f''|$ and $lnks[pos] \neq \mathbf{nil}$, which is checked in line 4. Hence, f' is a suffix of $f_1 f_2 \cdots f_{lnks[pos]}$. It is easy to see that the length of the longest common suffix of $f_1 f_2 \cdots f_{lnks[pos]}$ and $f_1 f_2 \cdots f_{p-1}$ is equal to $a.len$, where a is the nearest common ancestor of the corresponding leaves of T. So, we have $a.len \geq |f'|$ iff f' is a suffix of $f_1 f_2 \cdots f_{p-1}$, which is tested in line 5.

Consider the function $\mathsf{absorbOne}$. Since the case $|f_p| < |f_z|$ is analogous to the case $|f_p| < |f_{z-1}f_z|$ in $\mathsf{absorbTwo}$, we omit its analysis. Suppose that $|f_p| \geq |f_z|$ (lines 2–5). The case $|f_z| = 1$ is obvious, so, assume $|f_z| > 1$. Clearly, if f_z is a suffix of f_p, then $f_z[1..|f_z|{-}1]$ is a suffix of the string $f_1 f_2 \cdots f_{phrs[p].lnk}$. Hence, we have $f_z[|f_z|] = f_p[|f_p|]$ and, by the definition of $lnks$, $lnks[m{-}1] \neq \mathbf{nil}$. We check these conditions in line 3. Then, similar to the case $|f_p| < |f_z|$, we find the nearest common ancestor a of the leaves of T corresponding to $\overleftarrow{f_1 f_2 \cdots f_{phrs[p].lnk}}$ and $\overleftarrow{f_1 f_2 \cdots f_{lnks[m-1]}}$ in line 5 and, finally, have $a.len{+}1 \geq |f_z|$ iff f_z is a suffix of f_p. $\square$

By Lemma 7, the functions $\mathsf{absorbOne}$ and $\mathsf{absordTwo}$ check whether the strings f_z and $f_{z-1}f_z$ are suffixes of a prefix contained in T. But f_z and $f_{z-1}f_z$ may have occurrences ending at the last position inside a phrase whose corresponding prefix does not belong to T. This case is processed by the functions $\mathsf{absorbTwo2}$ and $\mathsf{absorbOne2}$.

function $\mathsf{absorbOne2}(m)$
  return $\mathsf{chk}(m, phrs[z].len)$;

function $\mathsf{absorbTwo2}(m)$
  unmark leaf $L[\overleftarrow{SA}[m - phrs[z].len - 1]]$;
  $r \leftarrow \mathsf{chk}(m, phrs[z-1].len + phrs[z].len)$;
  mark leaf $L[\overleftarrow{SA}[m - phrs[z].len - 1]]$;
  return r;

function $\mathsf{chk}(m, len)$
  $(ln, x) \leftarrow \mathsf{markedLCP}(m - 1)$;
  if $ln < len$ **then return false**;
  $ptr = \max\{k\colon \sum_{j=1}^{k} phrs[j].len \leq x\}$;
  return true;

function $\mathsf{markedLCP}(q)$
  $i' \leftarrow \max\{i'\colon i' < \overleftarrow{SA}[q]$ and $L[i']$ is marked$\}$ or $+\infty$ if there is no max; ▷ use M here
  $i'' \leftarrow \min\{i''\colon \overleftarrow{SA}[q] < i''$ and $L[i'']$ is marked$\}$ or $-\infty$ if there is no min; ▷ use M here
  $y' \leftarrow \min\{lcp[j]\colon i' < j \leq \overleftarrow{SA}[q]\}$ or 0 if $i' = +\infty$; ▷ use RMQ here
  $y'' \leftarrow \min\{lcp[j]\colon \overleftarrow{SA}[q] < j \leq i''\}$ or 0 if $i'' = -\infty$; ▷ use RMQ here
  if $y' > y''$ **then return** $(y', SA[i'])$;
  else return $(y'', SA[i''])$;

Lemma 8. *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of $s[1..m{-}1]$. For any $q \in [i\ell{-}3\ell..i\ell]$, $\mathsf{markedLCP}(q)$ finds in $O(\log \ell)$ time a pair (ln, x) such that $L[\overleftarrow{SA}[x]]$ is a marked leaf of M, $x \neq q$, ln is the length of the longest common suffix of $s[i\ell - 3\ell..q]$ and $s[i\ell{-}3\ell..x]$, and any other string $s[i\ell{-}3\ell..p']$ such that $L[\overleftarrow{SA}[p']]$ is marked and $p' \neq q$ has a shorter or the same longest common suffix with $s[i\ell - 3\ell..q]$.*

Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of the string $s[1..m{-}1]$. By the definition of M, a leaf $L[\overleftarrow{SA}[j]]$ is marked iff $j \in [i\ell{-}3\ell..i\ell]$ and $j = |f_1 f_2 \cdots f_k|$ for some $k \in [1..z]$. So, by Lemma 8, if $f_1 f_2 \cdots f_z$ has a suffix of length $len \leq \ell$ that is a suffix of a string $f_1 f_2 \cdots f_k$ such that $|f_1 f_2 \cdots f_k| \geq i\ell - 2\ell$, then we obtain $ln \geq len$ in line 3 in the function $\mathsf{chk}(m, len)$. In this case the function computes this number k in $O(\log \ell)$ time using the tree P and stores k in the global variable ptr.

Thus, since the verification whether f_z is a suffix of a string $f_1 f_2 \cdots f_k$ such that $|f_1 f_2 \cdots f_k| < i\ell - 2\ell$ is performed by $\mathsf{absorbOne}$, the call to $\mathsf{absorbOne2}(m)$ in the phase processing code returns true iff f_z is a suffix of a string $f_1 f_2 \cdots f_k$ for $k \in [1..z)$ such that $|f_1 f_2 \cdots f_k| \geq i\ell - 2\ell$. Similarly, $\mathsf{absorbTwo2}(m)$ returns true iff $f_{z-1} f_z$ is a suffix of a string $f_1 f_2 \cdots f_k$ for $k \in [1..z{-}1)$ such that $|f_1 f_2 \cdots f_k| \geq i\ell - 2\ell$.

So, $\mathsf{absorbOne2}$ and $\mathsf{absorbTwo2}$ complement $\mathsf{absorbOne}$ and $\mathsf{absorbTwo}$ checking whether $\overleftarrow{f}_z$ or $\overleftarrow{f}_z \overleftarrow{f}_{z-1}$ is a prefix of a string $\overleftarrow{f_1 f_2 \cdots f_k}$ that is not contained in T. Thus, $lens[m]$ and $lnks[m]$ are filled with correct values. Finally, the function $\mathsf{updateRecent}$ performs in $O(\log \ell)$ time at most two unmarkings and one marking in the tree M according to the updated array $phrs$, and modifies the tree P appropriately.

Phase postprocessing. Once the ith phase is over, we must prepare all structures for the next phase. Let $f_1 f_2 \cdots f_z$ be the current parsing. First, we add to T the strings $\overleftarrow{f_1 f_2 \cdots f_{z'}}$, $\overleftarrow{f_1 f_2 \cdots f_{z'+1}}, \ldots,$ $\overleftarrow{f_1 f_2 \cdots f_{z''}}$, where z' and z'' are such that $\overleftarrow{f_1}, \overleftarrow{f_1 f_2}, \ldots, \overleftarrow{f_1 f_2 \cdots f_{z'-1}}$ are already in T, $\overleftarrow{f_1 f_2 \cdots f_{z'}}$ is not in T, and $|f_1 f_2 \cdots f_{z''-1}| \leq i\ell - \ell < |f_1 f_2 \cdots f_{z''}|$. The following lemma is an easy corollary of Lemma 1.

Lemma 9. *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a string. If the trie T contains the strings $\overleftarrow{f_1}, \overleftarrow{f_1 f_2}, \ldots, \overleftarrow{f_1 f_2 \cdots f_{j-1}}$ for $j < z$, then the longest prefix of the string $\overleftarrow{f_1 f_2 \cdots f_j}$ that is represented in T has length less than $|f_j|$.*

To insert $\overleftarrow{f_1 f_2 \cdots f_j}$ (for $j = z', z'{+}1, \ldots, z''$) in T, we read f_j right-to-left and traverse T from the root like a Patricia trie using $v.map$ in the traversed vertices v and skipping the strings written on edges. Let v be the deepest vertex found by this process. Then, we calculate the length of the longest common suffix of the strings f_j and $f_1 f_2 \cdots f_{v.phr}$ by Lemma 6 (it is less than $|f_j|$ by Lemma 9) thus obtaining the position in T where the new leaf must be inserted. The nca data structure [3] is modified appropriately. (It is easy to see that Lemma 9 still holds in the presence of false positives; however, if ℓ artificially restricts the length of phrases and $|f_j| = \ell$, the longest common suffix of f_j and $f_1 f_2 \cdots f_{v.phr}$ can be f_j itself, but then we can ignore f_j since the "top ℓ-part" of T, which is actually important for us, remains correct.)

Denote by u_0 and u_1 the new leaf and its parent, respectively (u_1 might also be new). In an obvious way we calculate in $O(1)$ time the numbers $p_{u_0} = \mathsf{rst}(u_0.len, k)$, for the maximal k such that $\mathsf{rst}(u_0.len, k) > u_1.len$, and $h_{u_0} = \mathsf{hash}(\overleftarrow{f_1 f_2 \cdots f_j}[1..p_{u_0}])$ using the array hs; then, we assign $\mathsf{nav}(p_{u_0}, h_{u_0}) \leftarrow u_0$.

Suppose that u_1 is a new vertex that has split the edge connecting a vertex v and the old parent of v. As above, we calculate p_{u_1} and $h_{u_1} = \mathsf{hash}(\overleftarrow{f_j}[1..p_{u_1}])$, and assign $\mathsf{nav}(p_{u_1}, h_{u_1}) \leftarrow u_1$. If p'_v, the old value of p_v, is greater than $u_1.len$, then we are done. Suppose that $p'_v \le u_1.len$. It follows from the definition of p_{u_1} that in this case $p'_v = p_{u_1}$. Then, we recalculate p_v, compute $h_v = \mathsf{hash}(\overleftarrow{f_1 f_2 \cdots f_{v.phr}}[1..p_v])$ in $O(p_v)$ time by Lemma 6, and, finally, assign $\mathsf{nav}(p_v, h_v) \leftarrow v$. By the definition of p'_v, we can have $p'_v \le u_1.len$ only if $v.len \le 2 \cdot u_1.len$, so, all this work takes $O(u_1.len)$ time. Thus, the insertions altogether take $O(|f_{z'}| + |f_{z'+1}| + \cdots + |f_{z''}|) = O(\ell)$ time.

The new strings in T require the rebuilding of $lnks$. First, we unmark in $O(\ell \log \ell)$ time all leaves $L[p]$ of M such that $SA[p] \ne |f_1 f_2 \cdots f_j|$ for any $j \in [z'..z'']$. Then, for each $q \in [i\ell - \ell..i\ell]$ such that $lnks[q] = \mathbf{nil}$, we compute $(ln, x) = \mathsf{markedLCP}(q-1)$ and, if $lens[q] \le ln$, assign the number $\max\{k\colon \sum_{j=1}^{k} phrs[j].len \le pos\}$, which is computed by P in $O(\log \ell)$ time, to $lnks[q]$. It follows from Lemma 8 and the bounding condition $lens[q] \le \ell$ that such algorithm indeed fills the array $lnks[i\ell - \ell..i\ell]$ with correct values. Finally, we assign $i \leftarrow i + 1$ and move to the next phase.

Thus, one phase including the postprocessing takes $O(\ell \log \ell)$ time and, therefore, the whole algorithm works in $O(n \log \ell)$ time and uses $O(z + \ell)$ space.

The non-fixed ℓ and verification. We maintain a variable ℓ putting $\ell = 8$ initially and proceed as above. Once we obtain a phrase of length $\ge \frac{1}{2}\ell$ during a phase processing, we put $\ell \leftarrow 4\ell$ and start a new phase from this point rebuilding all internal phase structures; we also remove a number of leaves from the trie T and modify the structures nav, nca, N appropriately according to the phase processing of the above algorithm. Obviously, such algorithm works in $O(n \log \ell)$ overall time and constructs the LZ-End parsing with high probability. Note that this version is not streaming anymore since we reread a substring of length 2ℓ each time the variable ℓ grows.

As we discussed above, the parsing is correct with high probability. To verify that possible false positives did not obscure the result, we read s right-to-left and compare with the string retrieved from the parsing with the aid of Lemma 6. If ℓ was fixed in advance and we intentionally did not produce phrases of length $>\ell$, then at this point we have a reasonable approximation of LZ-End that encodes the string s and possesses properties similar to LZ-End. (We do not provide any theoretical evidence why this parsing is an approximation in a sense; we rather rely on intuition here.)

Experimental Results

We implemented the algorithm described in this paper in C++ and compared its runtime and the size of the resulting parsing to a number of LZ77 algorithms. The experiments were performed on a machine equipped with two six-core 1.9 GHz Intel Xeon E5-2420 CPUs with 15 MiB L3 cache and 120 GiB of DDR3 RAM. The machine had 6.8 TiB of free

Table 2: Statistics of the testfiles used in experiments; z is the number of phrases in the LZ77 parsing, z' is the number of phrases in the LZ-End parsing with $\ell = 8 \times 2^{20}$.

Input	$n/2^{30}$	σ	n/z	z'/z
kernel	128	229	4547.5	1.23
geo	128	211	3147.3	1.13
chr14	128	6	5957.9	1.25

disk space striped with RAID0 across four identical local disks achieving a transfer rate of ~480 MiB/s. The OS was Ubuntu 12.04, 64bit running kernel 3.13.0. All programs were compiled using `g++` v5.2.1 with `-O3 -DNDEBUG -march=native` options. The implementations of all algorithms used in experiments are available at `https://www.cs.helsinki.fi/group/pads/`. The experiments were run using three highly repetitive testfiles (see also Table 2):

– **kernel**: a concatenation of source files from over 150 versions of Linux kernel (`http://www.kernel.org/`);
– **geo**: a concatenation of all versions (edit history) of Wikipedia articles about all countries and 10 largest cities in the XML format;
– **chr14**: multiple versions of *Homo Sapiens* chromosome 14 repeated to obtain a 128 GiB file. Each version is obtained by randomly mutating the original chromosome with rate 0.01%. See `http://hgdownload.cse.ucsc.edu/`.

Text symbols are encoded using 8 bits and all algorithms in experiments use 40-bit integers to encode text positions. The goal of our experiments is to determine: (1) how scalable is the algorithm described in this paper, and (2) whether it is competitive with the best external-memory algorithms computing LZ77 parsing.

The two fastest algorithm to compute the LZ77 parsing in external memory are EM-LZscan and EM-LPF [8]. EM-LZscan uses very little disk space and is very fast if the input is highly repetitive and many phrases are entirely contained inside each other. It gets slow, however, as the text-to-RAM ratio increases, since it needs to scan essentially the whole text n/M times, where M is the size of available RAM. EM-LPF, on the other hand, is more scalable, but since it needs the suffix and LCP arrays as input, its disk space usage is at least 10 times the size of the input text.

For experiments, we fixed $\ell = 8 \times 2^{20}$, as it is small enough to not affect the RAM usage significantly, and big enough to have essentially no effect on the parsing size. In the preliminary run we executed our new algorithm on the full 128 GiB instances of all three testfiles, we recorded the following peak RAM usages: 4161 MiB (kernel), 4557 MiB (geo), and 3605 MiB (chr14).

In the main experiment we executed all algorithms on increasing length prefixes of all testfiles and measured the runtime. As explained above, for fair comparison with the new algorithm, we allowed the LZ77 parsing algorithms to use 3.5 GiB of RAM (and we restricted the physical RAM available in the system to 4 GiB). After each run of the algorithm computing the LZ-End parsing, we run the verifier on the resulting parsing (resulting in the second scan of the input), but we never encountered any false positives. The time for the verification is not included in the runtime of LZ-End parsing. The results are given in Figure 1.

First, we observe that the algorithm to compute LZ-End scales very well with increasing input. This is not surprising, as the algorithm has linear I/O complexity. Second, the LZ-End construction is usually around two times slower than EM-LPF, and up to four times slower than EM-LZscan, making our LZ-End parser at least competitive with the existing LZ77 parsers.

It should be kept in mind, however, that because our LZ-End parser does not need any disk space and only makes one left-to-right pass over the input (two, if

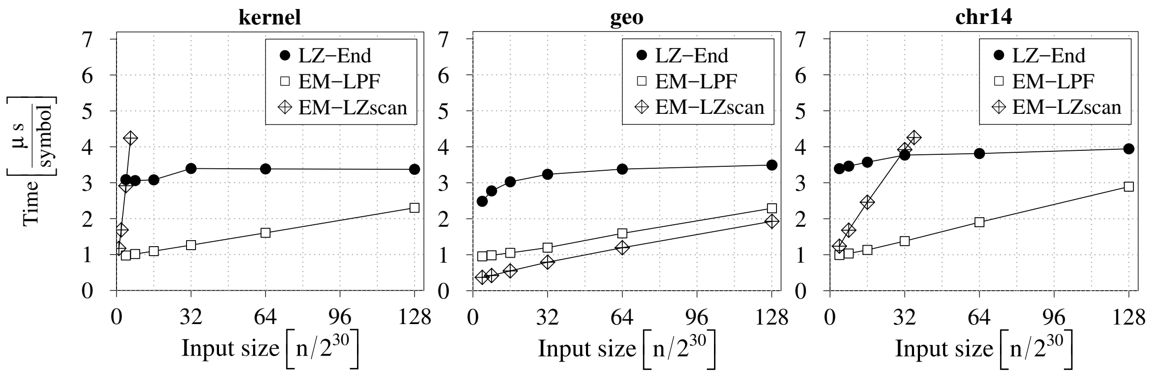

Figure 1: Runtime (in μs per input symbol) of the new algorithm compared to the fastest external-memory LZ77 parsing algorithms. EM-LPF and EM-LZscan use 3.5 GiB of RAM. EM-LPF includes the runtime for external-memory suffix [9] and LCP array construction [6].

we include the verification) the algorithm has a number of properties that none of the LZ77 algorithms have, e.g., the whole computation can be performed over the network, or by decompressing the data on-the-fly. Our algorithm only scans the input at a rate of 0.24–0.40 MB/s which is well below the typical network bandwidth, or the decompression speed of a typical modern decompressors like `gzip` of `bzip2`. Lastly, we observe that the computed LZ-End parsing is never more than 25% larger than the size of LZ77 parsing (see Table 2), showing that the LZ-End parsing is a valid replacement for LZ77 in practice.

Acknowledgement. The authors would like to thank Simon Puglisi and Juha Kärkkäinen for helpful discussions and comments that helped to improve the paper.

References

[1] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Monotone minimal perfect hashing: searching a sorted table with O(1) accesses," in *SODA*, 2009, pp. 785–794.
[2] D. Belazzougui and S. J. Puglisi, "Range predecessor and Lempel–Ziv parsing," in *SODA*, 2016, pp. 2053–2071.
[3] R. Cole and R. Hariharan, "Dynamic LCA queries on trees," in *SODA*, 1999, pp. 235–244.
[4] M. Crochemore and W. Rytter, *Jewels of stringology*. World Sci. Publishing, 2002.
[5] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka, "Approximating LZ77 via small-space multiple-pattern matching," in *ESA*, 2015, pp. 533–544.
[6] J. Kärkkäinen and D. Kempa, "Faster external memory LCP array construction," in *ESA*, 2016, pp. 61:1–61:16.
[7] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lightweight Lempel–Ziv parsing," in *SEA*, 2013, pp. 139–150.
[8] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lempel–Ziv parsing in external memory," in *DCC*, 2014, pp. 153–162.
[9] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel external memory suffix sorting," in *CPM*, 2015, pp. 329–342.
[10] D. Kempa and D. Kosolobov, "LZ-End parsing in compressed space," *arXiv*, 2016.
[11] D. Kosolobov, "Faster lightweight Lempel–Ziv parsing," in *MFCS*, 2015, pp. 432–444.
[12] S. Kreft and G. Navarro, "Self-indexing based on LZ77," in *CPM*, 2011, pp. 41–54.
[13] A. Policriti and N. Prezza, "Fast online Lempel–Ziv factorization in compressed space," in *SPIRE*, 2015, pp. 13–20.
[14] S. Kreft and G. Navarro, "LZ77-like compression with fast random access," in *DCC*, 2010, pp. 239–248.
[15] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.