Published in final edited form as:

Lebniz Int Proc Inform. 2021 ; 204: . doi:10.4230/LIPIcs.ESA.2021.56.

Fast and Space-Efficient Construction of AVL Grammars from the LZ77 Parsing

Dominik Kempa,

Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Ben Langmead

Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Abstract

Grammar compression is, next to Lempel–Ziv (LZ77) and run-length Burrows–Wheeler transform (RLBWT), one of the most flexible approaches to representing and processing highly compressible strings. The main idea is to represent a text as a context-free grammar whose language is precisely the input string. This is called a straight-line grammar (SLG). An AVL grammar, proposed by Rytter [Theor. Comput. Sci., 2003] is a type of SLG that additionally satisfies the AVL property: the heights of parse trees for children of every nonterminal differ by at most one. In contrast to other SLG constructions, AVL grammars can be constructed from the LZ77 parsing in compressed time: $\mathcal{O}(z \log n)$ where z is the size of the LZ77 parsing and n is the length of the input text. Despite these advantages, AVL grammars are thought to be too large to be practical.

We present a new technique for rapidly constructing a small AVL grammar from an LZ77 or LZ77-like parse. Our algorithm produces grammars that are always at least five times smaller than those produced by the original algorithm, and usually not more than double the size of grammars produced by the practical Re-Pair compressor [Larsson and Moffat, Proc. IEEE, 2000]. Our algorithm also achieves low peak RAM usage. By combining this algorithm with recent advances in approximating the LZ77 parsing, we show that our method has the potential to construct a run-length BWT in about one third of the time and peak RAM required by other approaches. Overall, we show that AVL grammars are surprisingly practical, opening the door to much faster construction of key compressed data structures.

Keywords

Theory of computation → Data compression; Theory of computation → Pattern matching; grammar compression; straight-line program; SLP; AVL grammar; Lempel–Ziv compression; LZ77; dictionary compression

Supplementary Material *Software*: https://github.com/dominikkempa/lz77-to-slp

1 Introduction

The increase in the amount of highly compressible data that requires efficient processing in the recent years, particularly in the area of computational genomics [3, 4], has caused a spike of interest in dictionary compression. Its main idea is to reduce the size of the representation of data by finding repetitions in the input and encoding them as references to other occurrences. Among the most popular methods are the Lempel–Ziv (LZ77) compression [32], run-length Burrows–Wheeler transform (RLBWT) [5, 16], and grammar compression [6]. Although in theory, LZ77 and RLBWT are separated by at most a factor of $\mathcal{O}\left(\log^2 n\right)$ (where n is the length of the input text) [15, 20], the gap in practice is usually noticeable (as also confirmed by our experiments). RLBWT is the largest of the three representations in practice, but is also the most versatile, supporting powerful suffix array and suffix tree queries [16]. LZ77, on the other hand, is the smallest, but its functionality includes only the easier longest common extension (LCE), random-access, and pattern matching queries [1, 7, 12, 13, 21]. Grammar compression occupies the middle ground between the two, supporting queries similar to LZ77 [26]. Navarro gives a comprehensive overview of these and related representations [26, 27].

A major practical concern with these representations – RLBWT in particular – is how to construct them efficiently. Past efforts have focused on engineering efficient *general* algorithms for constructing the BWT and LZ77 [10, 18, 2, 17], but these are not applicable to the terabyte-scale datasets routinely found, e.g., in modern genomics [4]. Specialized algorithms for highly repetitive datasets have only been investigated recently. Boucher et al. [4] proposed a method for the efficient construction of RLBWT using the concept of prefix-free parsing. The same problem was approached by Policriti and Prezza, and Ohno et al. [29, 28], using a different approach based on the dynamic representation of RLBWT. These methods represent the state of the art in the practical construction of RLBWT.

A different approach to the construction of RLBWT was recently proposed in [20]. The idea is to first compute the (exact or approximate) LZ77 parsing for the text, and then convert this representation into an RLBWT. Crucially, the LZ77 → RLBWT conversion takes only $\mathcal{O}(z \operatorname{polylog} n)$ time (where z is the size of the LZ77 parsing), i.e., it runs not only in the compressed space but also in *compressed time*.[1] The computational bottleneck is therefore shifted to the easier problem of computing or approximating the LZ77, which is the only step taking $\Omega(n)$ time. Internally, this new pipeline consists of three steps: text → (approximate) LZ77 → grammar → RLBWT, unsurprisingly aligning with the gradual increase in the size and complexity of these representations. Kosolobov et al. [24] recently proposed a fast and space-efficient algorithm to approximate LZ77, called Re-LZ. The second and third steps in the pipeline, from the LZ77 parse to the RLBWT, have not been implemented. The only known algorithm to convert LZ77 into a grammar in compressed time was proposed by Rytter [30], and is based on the special type of grammars called *AVL grammars*, whose distinguishing feature is that all subtrees in the parse tree satisfy the AVL

[1] $\operatorname{polylog} n = \log^c n$ for some constant $c > 0$.

property: the tree-heights for children of every nonterminal do not differ by more than one. The algorithm is rather complex, and until now has been considered impractical.

Our Contribution.

Our main contribution is a series of practical improvements to the basic variant of Rytter's algorithm, and a fast and space-efficient implementation of this improved algorithm. Compared to the basic variant, ours produces a grammar that is always five times smaller, and crucially, the same holds for all intermediate grammars computed during the algorithm's execution, yielding very low peak RAM usage. The resulting grammar is also no more than twice of the smallest existing grammar compressors such as Re-Pair [25]. We further demonstrate that combining our new improved algorithm with Re-LZ opens up a new path to the construction of RLBWT. Our preliminary experiments indicate that at least a three-fold speedup and the same level of reduction in the RAM usage is possible.

The key algorithmic idea in our variant is to delay the merging of intermediate AVL grammars as much as possible to avoid creating nonterminals that are then unused in the final grammar. We dub this variant *lazy AVL grammars*. We additionally incorporate Karp–Rabin fingerprints [19] to re-write parts of the grammar on-the-fly and further reduce the grammar size. We describe two distinct versions of this technique: greedy and optimal, and demonstrate that both lead to reductions in the grammar size. As a side-result of independent interest, we describe a fast and space-efficient data structure for the dynamic predecessor problem, in which the inserted key is always larger than all other elements currently in the set.

2 Preliminaries

Strings.

For any string S, we write $S[i...j]$, where $1 \leq i, j \leq |S|$, to denote a substring of S. If $i>j$, we assume $S[i...j]$ to be the empty string ε. By $[i...j)$ we denote $[i \ldots j-1]$. Throughout the paper, we consider a string (text) $T[1...n]$ of $n \geq 1$ symbols from an integer alphabet $\Sigma = [0...\sigma)$. By $\mathrm{LCE}(i, i')$ we denote the length of the longest common prefix of suffixes $T[i...n]$ and $T[i'...n]$.

Karp–Rabin Fingerprints.

Let q be a prime number and let $r \in [0...q)$ be chosen uniformly at random. The *Karp–Rabin fingerprint* of a string S is defined as

$$\Phi(S) = \sum_{i=1}^{|S|} S[i] \cdot r^{|S|-i} \bmod q .$$

Clearly, if $T[i...i+\ell) = T[j...j+\ell)$ then $\Phi(T[i...i+\ell)) = \Phi(T[j...j+\ell))$. On the other hand, if $T[i...i+\ell) \neq T[j...j+\ell)$ then $\Phi(T[i...i+\ell)) \neq \Phi(T[j...j+\ell))$ with probability at least $1-\ell/q$ [9]. In our algorithm we are comparing only substrings of T of equal length. Thus, the number of different possible substring comparisons is less than n^3, and hence for any

positive constant c, we can set q to be a prime larger than n^{c+4} (but still small enough to fit in $\mathcal{O}(1)$ words) to make the fingerprint function perfect with probability at least $1 - n^{-c}$.

LZ77 Compression.

An *LZ77-like factorization* of T is a factorization $T = F_1 \cdots F_f$ into non-empty *phrases* such that every phrase F_j with $|F_j| > 1$ has an earlier occurrence in T, i.e., letting $i = 1 + |F_1 \cdots F_{j-1}|$ and $\ell = |F_j|$, there exists $p \in [1...i)$ satisfying $\mathrm{LCE}(p, i) \geq \ell$. The phrase $F_j = T[i...i + \ell)$ is encoded as a pair (p, ℓ). If there are multiple choices for p, we choose one arbitrarily. The occurrence $T[p...p + \ell)$ is called the *source* of F_j. If $\ell = 1$, the phrase $F_j = T[i]$ is encoded as a pair $(T[i], 0)$. The LZ77-like parsing, in which we additionally require the phrase to not overlap its source, i.e., $p + \ell \leq i$, is called *non-self-referential*.

The LZ77 factorization [32] (or the LZ77 parsing) of a string T is an LZ77-like factorization constructed by greedily parsing T from left to right into longest possible phrases. More precisely, the jth phrase F_j is the longest substring starting at position $i = 1 + |F_1 \cdots F_{j-1}|$ that has an earlier occurrence in T. If there is no such substring, then $F_j = T[i]$. We denote the number of phrases in the LZ77 parsing by z. For example, the text bbabaababababaababa has the LZ77 parsing b · b · a · ba · aba · bababa · ababa with $z = 7$ phrases, and is encoded as a sequence (b,0),(1,1),(a,0),(2,2),(3,3),(7,6),(10,5).

Grammar Compression.

A context-free grammar is a tuple $G = (N, \Sigma, R, S)$, where N is a finite set of *nonterminals*, Σ is a finite set of *terminals*, and $R \subseteq N \times (N \cup \Sigma)^*$ is a set of *rules*. We assume $N \cap \Sigma = \emptyset$ and $S \in N$. The nonterminal S is called the *starting symbol*. If $(A, \gamma) \in R$ then we write $A \to \gamma$. The *language* of G is set $L(G) \subseteq \Sigma^*$ obtained by starting with S and repeatedly replacing nonterminals with their expansions, according to R.

A grammar $G = (N, \Sigma, R, S)$ is called a *straight-line grammar* (SLG) if for any $A \in N$ there is exactly one production with A of the left side, and all nonterminals can be ordered $A_1, ..., A_{|N|}$ such that $S = A_1$ and if $A_i \to \gamma$ then $\gamma \in (\Sigma \cup \{A_{i+1}, ..., A_{|N|}\})^*$, i.e., the graph of grammar rules is acyclic. The unique γ such that $A \to \gamma$ is called the *definition* of A and is denoted $\mathrm{rhs}(A)$. In any SLG, for any $u \in (N \cup \Sigma)^*$ there exists exactly one $w \in \Sigma^*$ that can be obtained from u. We call such w the *expansion* of u, and denote it by $\exp(u)$. We define the *parse tree* of $A \in N \cup \Sigma$ as a rooted ordered tree $\mathcal{T}(A)$, where each node v is associated to a symbol $\mathrm{sym}(v) \in N \cup \Sigma$. The root of $\mathcal{T}(A)$ is a node v such that $\mathrm{sym}(v) = A$. If $A \in \Sigma$ then v has no children. If $A \in N$ and $\mathrm{rhs}(A) = B_1 \cdots B_k$, then v has k children and the subtree rooted at the ith child is a copy of $\mathcal{T}(B_i)$. The parse tree $\mathcal{T}(G)$ is defined as $\mathcal{T}(S)$. The *height* of any $A \in N$ is defined as the height of $\mathcal{T}(A)$, and denoted $\mathrm{height}(A)$.

The idea of *grammar compression* is, given a text T, to compute a small SLG G such that $L(G) = \{T\}$. The *size* of the grammar is measured by the total length of all definitions, and denoted $|G| := \Sigma_{A \in N} |\mathrm{rhs}(A)|$. Clearly, it is easy to encode any G in $\mathcal{O}(|G|)$ space: pick an ordering of nonterminals and write down the definitions of all nonterminals, replacing nonterminal symbols with their numbers in the order.

3 AVL Grammars and the Basic Algorithm

An SLG $G = (N, \Sigma, R, S)$ is said to be in *Chomsky normal form*, if for every $A \in N$, it holds $\mathrm{rhs}(A) \in \Sigma$ or $\mathrm{rhs}(A) = XY$, where $X, Y \in N$. An SLG in Chomsky normal form is called a *straight-line program* (SLP). Rytter [30] defines an *AVL grammar* as an SLP $G = (N, \Sigma, R, S)$ that additionally satisfies the *AVL property*: for every $A \in N$ such that $\mathrm{rhs}(A) = XY$, it holds $|\mathrm{height}(X)| - |\mathrm{height}(Y)| \leq 1$. This condition guarantees that for every $A \in N$ (in particular for $S \in N$), it holds $\mathrm{height}(A) = \mathcal{O}(\log|\exp(A)|)$ [30, Lemma 1].

The main result presented in [30] is an algorithm that given a non-self-referential LZ77-like parsing of a length-n text T consisting of f phrases, computes in $\mathcal{O}(f \log n)$ time an AVL grammar G generating T and satisfying $|G| = \mathcal{O}(f \log n)$. Rytter's construction was extended to allow self-references in [20, Theorem 6.1]. Our implementation of the basic as well as improved Rytter's algorithm works for the self-referential variant, but for simplicity here and in Section 4 we describe the algorithm only for the non-self-referential variant.

The algorithm in [30] works in f steps. It maintains the dynamically changing AVL grammar G such that after the kth step is complete, there exists a nonterminal P_k in G such that $\exp(P_k) = F_1 \cdots F_k$, where $T = F_1 \cdots F_f$ is the input LZ77-like factorization of the input. This implies that at end there exist a nonterminal expanding to T. The algorithm does not delete any nonterminals between steps. At the end, it may perform an optional pruning of the grammar to remove the nonterminals not present in the parse tree $\mathcal{T}(P_f)$. This reduces the grammar size but not the peak memory usage of the algorithm.

The algorithm uses the following three procedures, each of which adds a nonterminal A with a desired expansion $\exp(A)$ to the grammar G, along with a bounded number of extra nonterminals:

1. AddSymbol(c): Given $c \in \Sigma$, add a nonterminal A with $\mathrm{rhs}(A) = c$ to the grammar G.
2. AddMerged(X, Y): Given the identifiers of nonterminals X, Y existing in G, add a nonterminal A to G that satisfies $\exp(A) = \exp(X)\exp(Y)$. The difficulty of this operation is ensuring that the updated G remains an AVL grammar. Simply setting $\mathrm{rhs}(A) = XY$ would violate the AVL condition in most cases. Instead, the algorithm performs the procedure similar to the concatenation of AVL trees [22, p. 474], taking $\mathcal{O}(1 + |\mathrm{height}(X) - \mathrm{height}(Y)|)$ time, and introducing $\mathcal{O}(|\mathrm{height}(X) - \mathrm{height}(Y)|)$ extra nonterminals.
3. AddSubstring(A, i, j): Given the identifier of a nonterminal A existing in G, and two positions satisfying $1 \leq i \leq j \leq |\exp(A)|$, add a nonterminal B to G that satisfies $\exp(B) = \exp(A)[i...j]$. To explain how this is achieved, we define the auxiliary procedure $\mathrm{Decompose}(A, i, j)$ that given the same parameters as above, returns the sequence $B_1, ..., B_q$ of nonterminals satisfying $\exp(A)[i...j] = \exp(B_1) \cdots \exp(B_q)$. The nonterminals B_i are found by performing two root-to-leaf traversals in the parse tree $\mathcal{T}(A)$. This takes

$\mathcal{O}(1+\text{height}(A)) = \mathcal{O}(1+\log|\exp(A)|)$ time and ensures $q = \mathcal{O}(\log|\exp(A)|)$. It is easy to see that given $B_1, ..., B_q$, we can now obtain B in $\mathcal{O}\left(1+\log^2|\exp(A)|\right)$ time using AddMerged. In [30], it was however shown that if we always choose the shortest nonterminal to merge with its neighbor, the total runtime is $\mathcal{O}(1+\log|\exp(A)|)$ and only $\mathcal{O}(\log|\exp(A)|)$ extra nonterminals are introduced.

Using the above three procedures, the algorithm in [30] works as follows. Suppose we have already processed the leftmost $k-1$ phrases. The step begins by creating a nonterminal A_k satisfying $\exp(A_k) = F_k$. If $|F_k| = 1$, it uses the procedure AddSymbol(F_k). Otherwise, A_k is obtained as the output of AddSubstring($P_{k-1}, p, p+\ell-1$), where $\ell = |F_k|$ and $T[p...p+\ell)$ is the source of phrase F_k. Finally, P_k is obtained as the output of AddMerged(P_{k-1}, A_k). A single iteration thus takes $\mathcal{O}(1+\log|F_1\cdots F_{k-1}|) = \mathcal{O}(\log n)$ time and adds $\mathcal{O}(\log n)$ extra nonterminals, for total of $\mathcal{O}(f\log n)$ nonterminals over all steps.

4 Modified Algorithm

Lazy Merging.

We start by observing that the main reason responsible for the large final grammar produced by the algorithm in Section 3 is the requirement that at the end of each step $k \in [1...f]$, there exist a nonterminal P_k satisfying $\exp(P_k) = F_1\cdots F_k$. We relax this requirement, and instead require only that at the end of step k, there exists a sequence of nonterminals $R_1, ..., R_m$ such that $\exp(R_1)\cdots\exp(R_m) = F_1\cdots F_k$. The algorithm explicitly maintains these nonterminals as a sequence of pairs $(\ell_1, R_1), ..., (\ell_m, R_m)$, where $\ell_j = \sum_{i=1}^{j}|\exp(R_i)|$. The modified algorithm uses the following new procedures:

1. MergeEnclosed (i, j): Given two positions satisfying $1 \le i \le j \le |F_1\cdots F_{k-1}|$, add to G a nonterminal R satisfying $\exp(R) = \exp(R_x)\cdots\exp(R_y)$, where $x = \min\{t \in [1...m]: \ell_{t-1} \ge i-1\}$ and $y = \max\{t \in [1...m]: \ell_t \le j\}$. The positions x and y are found using a binary search. The pairs of the sequence $(\ell_1, R_1), ..., (\ell_m, R_m)$ at positions between x and y are then removed and replaced with a pair (ℓ_y, R). In other words, this procedure merges all the nonterminals from the current root sequence whose expansion is entirely inside the given interval $[i...j]$. Merging of $R_x, ..., R_y$ is performed pairwise, using the AddMerged procedure. Note, however, that there is no dependence between heights of the adjacent nonterminals (in particular, they do not form a bitonic sequence, like in the algorithm in Section 3), and moreover, their number $\widehat{m} := y - x + 1$ is not b bounded. To minimize the number of newly introduced nonterminals, we thus employ a greedy heuristic, that always chooses the nonterminal with the smallest height among the remaining elements, and merges it with a shorter neighbor. We use a list to keep the pointers between neighbors and a binary heap to maintain heights. The merging thus runs in $\mathcal{O}(\widehat{m}\log n)$ time.

2. DecomposeWithRoots (i, j): Given two positions satisfying $1 \le i \le j \le |F_1\cdots F_{k-1}|$, this procedure returns a sequence of nonterminals $A_1, ..., A_q$ satisfying $(F_1\cdots F_{k-1})[i...j] = \exp(A_1)\cdots\exp(A_q)$. First, it computes positions x and y, as defined

in the description of MergeEnclosed above. It then returns the result of Decompose $(R_{x-1}, i-\ell_{x-2}, \ell_{x-1}-\ell_{x-2})$, followed by $R_x, ..., R_y$, followed by the result of Decompose $(R_{y+1}, 1, j-\ell_y)$ (appropriately handling the boundary cases, which for clarity we ignore here). In other words, this procedure finds the sequence of nonterminals that uses as many roots from the sequence $R_1, ..., R_m$, as possible, and runs the standard Decompose for the boundary roots. Letting $\widehat{m} = y - x + 1$, it runs in $\mathcal{O}(\widehat{m} + \log n)$ time.

Using the above additional procedures, our algorithm works as follows. Suppose that we have already processed the first $k-1$ phrases. The step begins by computing the sequence of nonterminals $A_1, ..., A_q$ satisfying $\exp(A_1)\cdots\exp(A_q) = F_k$. If $|F_k| = 1$, we proceed as in Section 3. Otherwise, we first call MergeEnclosed $(p, p+\ell-1)$, where $\ell = |F_k|$ and $T[p...p+\ell)$ is the source of F_k. The sequence $A_1, ..., A_q$ is then obtained as a result of DecomposeWithRoots $(p, p+\ell-1)$. Finally, $A_1, ..., A_q$, is appended to the roots sequence.

The above algorithm runs in $\mathcal{O}\left(f \log^2 n\right)$ time. To see this, note that first calling MergeEnclosed ensures that the output size of DecomposeWithRoots is $\mathcal{O}(\log n)$. Thus, each step appends only $\mathcal{O}(\log n)$ nonterminals to the roots sequence. The total time spend in MergeEnclosed is thus bounded by $\mathcal{O}\left(f \log^2 n\right)$, dominating the time complexity.

To prove the correctness of the modified algorithm, we need to prove that: (1) every nonterminal in the new algorithm satisfies the AVL property, and (2) after iteration $k \in [1...f]$, the invariant $\exp(R_1)\cdots\exp(R_m) = F_1\cdots F_k$ holds. To show (1), we note that in the above algorithm, the nonterminals are only created by the MergeEnclosed procedure. Internally, this procedure calls AddMerged, which guarantees that the newly created nonterminal satisfies the AVL property (see Section 3). To show (2), we first note that, by definition, MergeEnclosed does not change $\exp(R_1)\cdots\exp(R_m)$ (although it may change m). The expansion of the roots sequence changes only after appending the sequence of nonterminals $A_1, ..., A_q$ returned by DecomposeWithRoots $(p, p+\ell-1)$. Since $T[p..p+\ell)$ is the source of F_k, we thus have $\exp(A_1)\cdots\exp(A_q) = (F_1\cdots F_{k-1})[p...p+\ell) = T[p...p+\ell) = F_k$.

Utilizing Karp–Rabin Fingerprints.

Our second technique is designed to detect the situation in which the algorithm adds a nonterminal A to G, when there already exists some $B \in N$ such that $\exp(A) = \exp(B)$. For any $u \in (N \cup \Sigma)^*$, we define $\Phi(u) = \Phi(\exp(u))$. Let us assume that there are no collisions between fingerprints.[2] During the algorithm, we maintain a collection of fingerprints $\{\Phi(A): A \in N'\}$, where $N' \subseteq N$ is some subset of nonterminals. Assume, that given a nonterminal $A \in N$, we can quickly compute $\Phi(A)$, and that given some $x \geq 0$, we can check, if there exists $B \in N'$ such that $\Phi(B) = x$. There are two places in the above algorithm (using lazy merging) where we utilize this to reduce the number of nonterminals:

[2]Although such an assumption can be ensured with probability $1 - n^{-c}$ for any constant $c > 0$, it cannot be easily guaranteed. This turns our algorithm into a Monte Carlo randomized algorithm.

1. Whenever during the greedy merge in MergeEnclosed, we are about to call AddMerged for the pair of adjacent nonterminals X and Y, we instead first compute the fingerprint $x = \Phi(XY)$ of their concatenation, and if there already exists $A \in N'$ such that $\Phi(A) = x$, we use A instead, avoiding the call to AddMerged and the creation of extra nonterminals.

2. Before appending the nonterminals $A_1, ..., A_q$ to the roots sequence at the end of the step, we check if there exists an equivalent but shorter sequence $B_1, ..., B_{q'}$, i.e., such that $\exp(A_1)\cdots\exp(A_q) = \exp(B_1)\cdots\exp(B_{q'})$ and $q' < q$. We utilize that $q = \mathcal{O}(\log n)$, and run a quadratic algorithm (based on dynamic programming) to find the optimal (shortest) equivalent sequence. We then use that equivalent sequence in place of $A_1, ..., A_q$.

Observe that the above techniques cannot be applied during AddMerged, as the equivalent nonterminal could have a much shorter/taller parse tree, violating the AVL property. Note also that the size and contents of N' do not affect the correctness or the time complexity of the algorithm. To determine the set N', our implementation uses a parameter $p \in [0...1]$, which is the probability of adding A to N', whenever a new nonterminal A is created. The value of p is one of the main parameters controlling the time-space trade-off of the algorithm, as well as the size of the final grammar. To check if there exists $A \in N'$ such that $\Phi(A) = x$, for a given $x \geq 0$, we maintain a hash table that maps the values from the set $\{\Phi(A): A \in N'\}$ to the corresponding nonterminals (each nonterminal is assigned a unique integer identifier).

5 Implementation Details

Storing Sequences.

Our implementation stores many sequences, where the insertion only happens at the end (e.g., the sequence of nonterminals, which are never deleted in the algorithm). A standard approach to this is to use a *dynamic array*, which is a plain array that doubles its capacity, once it gets full. Such implementation achieves an amortized $\mathcal{O}(1)$ insertion time, but suffers from a high peak RAM usage. On all systems we tried, the reallocation call that extends the array is not in-place. Since the peak RAM usage is critical in our implementation, we implemented our own dynamic array that instead of a single allocated block, keeps a larger number of blocks (we use 32). This significantly reduces the peak RAM usage. We found the slowdown in the access-time to be negligible.

Implementation of the Roots Sequence.

The roots sequence $(\ell_1, R_1), ..., (\ell_m, R_m)$ undergoes predecessor queries, deletions (at arbitrary positions), and insertions (only at the end). Rather than using an off-the-shelf dynamic predecessor data structure (such as balanced BST), we exploit as follows the fact that insertions happen only at the end.

All roots are stored as a static sequence that only undergoes insertions at the end (using the space efficient dynamic array implementation described above). Deleted elements are

marked as deleted, but remain physically in the array. The predecessor query is implemented using a binary search, with skipping of the elements marked as deleted. To ensure that the predecessor queries are efficient, we keep a counter of accesses to the deleted elements. Once it reaches the current array size, we run the "garbage collector" that scans the whole array left-to-right, and removes all the elements marked as deleted, eliminating all gaps. This way, the predecessor query is still efficient, except the complexity becomes amortized.

Computing $\Phi(A)$ and $|\exp(A)|$ for $A \in N$.

During the algorithm, we often need to query the value of $\Phi(A)$ for some nonterminal $A \in N$. In our implementation we utilize 64-bit fingerprints, and hence storing the value $\Phi(A)$ for every nonterminal is expensive. We thus only store $\Phi(A)$ for $A \in N$ satisfying $|\exp(A)| \geq 255$. The number of such elements in N is relatively small. To compute $\Phi(A)$ for any other $A \in N$, we first obtain $\exp(A)$, and then compute $\Phi(A)$ from scratch. This operation is one of the most expensive in our algorithm, and hence whenever possible we avoid doing repeated Φ queries.

As for the values $|\exp(A)|$, we observe that in most cases, it fits in a single byte. Thus, we designate only a single byte, and whenever $|\exp(A)| \geq 255$, we lookup $|\exp(A)|$ in an array ordered by the number of nonterminal, with access implemented using binary search.

6 Experimental Results

Algorithms.

We performed experiments using the following algorithms:

- Basic-AVLG, our implementation of the algorithm to convert an LZ-like parsing to an AVL grammar proposed by Rytter [30], and outlined in Section 3. Self-referential phrases are handled as in the full version of [20, Theorem 6.1]. The implementation uses space-efficient dynamic arrays described in Section 5. This implementation is our baseline.
- Lazy-AVLG, our implementation of the improved version of Basic-AVLG, utilizing lazy merging and Karp–Rabin fingerprints, as described in Section 4. This is the main contribution of our paper. In some of the experiments below, we consider the algorithm with the different probability p of sampling the Karp–Rabin hash of a nonterminal, but our default value (as discussed below) is $p = 0.125$. Our implementation (including also Basic-AVLG) is available at https://github.com/dominikkempa/lz77-to-slp.
- Big-BWT, a semi-external algorithm constructing the RLBWT from the input text in $\Omega(n)$ time, proposed by Boucher et al. [4]. As shown in [4], if the input text is highly compressible, the working space of Big-BWT is sublinear in the text length. We use the implementation from https://gitlab.com/manzai/Big-BWT.
- Re-LZ, an external-memory algorithm due to Kosolobov et al. [24] that given a text on disk, constructs its LZ-like parsing in $\Omega(n)$ time [24]. The algorithm

is faster than the currently best algorithms to compute the LZ77 parsing. In practice, the ratio f/z between the size f of the resulting parsing and the size z of the LZ77 parsing usually does not exceed 1.5. Its working space is fully tunable and can be specified arbitrarily. We use the implementation from https://gitlab.com/dvalenzu/ReLZ.

- Re-Pair, an $\mathcal{O}(n)$-time algorithm to construct an SLG from the text, proposed by Larsson and Moffat [25]. Although no upper bound is known on its output size, Re-Pair produces grammars that in practice are smaller than any other grammar compression method [25]. Its main drawback is that most implementations need $\Theta(n)$ space [11], and hence are not applicable on massive datasets. The only implementation using $o(n)$ space is [23], but as authors note themselves, it is not practical. There is also work on running Re-Pair on the compressed input [31], but since it already requires the text as a grammar, it is not applicable in our case. In our experiments we therefore use Re-Pair only as a baseline for the achievable grammar size. We note that there exists recent work on optimizing Re-Pair by utilizing maximal repeats [11]. The decrease in the grammar size, however, requires a potentially more expensive nonterminal encoding that includes the length of the expansion. For simplicity, we therefore use the basic version of Re-Pair.

All implementations are in C++ and are largely sequential, allowing for a constant number of additional threads used for asynchronous I/O.

We also considered Online-RLBWT, an algorithm proposed by Ohno at al. [28], that given a text in a right-to-left streaming fashion, construct its run-length compressed BWT (RLBWT) in $\mathcal{O}(n\log r)$ time and using only $\mathcal{O}(r\log n)$ bits of working space (the implementation is available from: https://github.com/itomomoti/OnlineRlbwt). In the preliminary experiment we determined that while using only about a third of the memory of Big-BWT (on the 16GiB prefix of the kernel testfile), the algorithm was about 10x slower than Big-BWT. We also did not include [14], since in the preliminary experiments we found it to be slower (usually by about 5–10%) than Big-BWT, while using about 1.8x more RAM.

Experimental Platform and Datasets.

We performed experiments on a machine equipped with two twelve-core 2.2GHz Intel Xeon E5–2650v4 CPUs with 30MiB L3 cache and 512GiB of RAM. The machine used distributed storage achieving an I/O rate >220MiB/s (read/write).

The OS was Linux (CentOS 7.7, 64bit) running kernel 3.10.0. All programs were compiled using g++ version 4.8.5 with -O3-DNDEBUG-march=native options. All reported runtimes are wallclock (real) times. The machine had no other significant CPU tasks running. To measure the peak RAM usage of the programs we used the /usr/bin/time -v command.

The statistics of testfiles used in our experiments are shown in Table 1. Shorter version of files used in the scalability experiments are prefixes of full files. We used the files from the Pizza & Chili repetitive corpus available at http://pizzachili.dcc.uchile.cl/repcorpus.html. We

chose a sample of 8 real and pseudo-real files. Since all files are relatively small (less than 512MiB), we additionally include 2 large repetitive files:

- chr19.1000, a concatenation of 1000 versions of Human chromosome 19. The sequences were obtained from the 1000 Genomes Project [8]. One copy consists of ~58×10^6 symbols.
- kernel, a concatenation of ~10.7 million source files from over 300 versions of the Linux kernel (see http://www.kernel.org/).

Karp–Rabin Sampling Rate.

The key parameter that controls the runtime, peak RAM usage, and the size of the grammar in our algorithm is the probability p of including the Karp–Rabin fingerprint of a nonterminal in the hash table. In our first experiment, we study the performance of the Lazy-AVLG algorithm for different values of the parameter p. We tested all values $p \in \{0, 0.1, 0.2, ..., 1\}$ and for each we measured the algorithm's time and memory usage, and the size of the final grammar. The results are given in Figure 1.

While utilizing the Karp–Rabin fingerprints (i.e., setting $p > 0$) can notably reduce the final grammar size (up to 40% for the cere file), it is not worth using values p much larger than 0.1, as it quickly increases the peak RAM usage (e.g., by about 2.3x for the sources.001.2 testfile) and this increase is not repaid significantly in the further grammar reduction. The bottom right panel in Figure 1 provides some insight into the reason for this. It shows the percentage of cases, where during the greedy merging of nonterminals enclosed by the source of the phrase, the algorithm is able to avoid merging two nonterminals, and instead use the existing nonterminal. Having *some* fingerprints in the hash table turns out to be enough to avoid creating between 4–14% of the new nonterminals, but having more does not lead to a significant difference. We also observe that if a larger grammar is acceptable, disabling the use of Karp–Rabin fingerprints entirely (i.e., setting $p = 0$) can lead to a significant speed-up (the top right panel in Figure 1) and a small saving in the RAM usage (note, that it also makes the algorithm deterministic). We choose to use $p > 0$, however, since in our main application (BWT construction), the final grammar is subject to further processing, and since this processing may dominate the RAM usage, we prefer to keep the grammar as small as possible. Since peak RAM usage is the likely limiting factor for this algorithm – a slower algorithm is still usable, but exhaustion of RAM can prevent it running entirely – we chose $p = 0.125$ as the default value in our implementation (and use in the next two experiments).

Grammar Size.

In our second experiment, we compare the size of the grammar produced by Lazy-AVLG to Basic-AVLG and Re-Pair. In the comparison we also include the size of the grammar obtained by running Basic-AVLG and removing all nonterminals not reachable from the root. We have run the experiments on 8/10 testfiles, as running Re-Pair on the large files is prohibitively time consuming. The results are given in Figure 2.

Lazy-AVLG produces grammars that are between 1.59x and 2.64x larger than Re-Pair (1.95x on average). The resulting grammar is always at least 5x smaller than produced by Basic-AVLG, and also always smaller than Basic-AVLG (pruned). Importantly, the RAM usage of our conversion is proportional the size of the final grammar, whereas the algorithm to compute the pruned version of Basic-AVLG must first obtain the initial large grammar, increasing peak RAM usage. This is a major practical concern, as described in the next experiment. In conclusion, Lazy-AVLG compresses only slightly worse than Re-Pair, but its construction requires much less working space.

Application in the Construction of BWT.

In our third experiment, we evaluate the potential of the method to construct the RLBWT presented in [20], which works by first computing/approximating the LZ77 parsing of the text in $\Omega(n)$ time, and then converting the resulting compressed representation of T into the RLBWT in $\mathcal{O}(f \text{ polylog } n)$ time (where f denotes the number of factors). We use Re-LZ to implement the first step. As for the second step, we note that the conversion from the (approximate) LZ77 to RLBWT internally consists of two steps: (2a) (approximate) LZ77 → grammar, and (2b) grammar → RLBWT. In this experiment, we use Lazy-AVLG to implement step (2a). We have not implemented the step (2b), and leave it as a future work. The results reported here are therefore only a preliminary indication of what is achievable with the approach of [20]. Our baseline for the construction of RLBWT is the Big-BWT algorithm [4].

We evaluated the runtime and peak RAM usage of Big-BWT, Re-LZ, and Lazy-AVLG with $p = 0.125$ (the default value) on successively longer prefixes of the large testfiles (chr19.1000 and kernel). The RAM use of Re-LZ was set to match the RAM use of Big-BWT on the shortest prefix we tried. To allow a comparison with different methods in the future, we evaluated Lazy-AVLG on the LZ77 parsing rather than on the output of Re-LZ. Thus, to obtain the performance of the pipeline Re-LZ + Lazy-AVLG, one should multiply the runtime and RAM usage of Lazy-AVLG by the approximation ratio f/z of Re-LZ. The value f/z did not exceed 1.05 on any of the kernel prefixes, and 1.27 on any of the chr19.1000 prefixes (with the peak reached on the largest prefixes). This puts the RAM use of Lazy-AVLG on the output of Re-LZ still below that of Re-LZ. The results are given in Figure 3.

The runtime of Re-LZ is always below that of Big-BWT. The reduction is by a factor of at least three for all prefixes of chr19.1000, and by at least 25% for all prefixes of kernel. The runtime of Lazy-AVLG stays significantly below that of both other methods. Importantly, this also holds for Lazy-AVLG's peak RAM usage. Given these results, we conclude that the construction of the RLBWT via LZ parsing has the potential to achieve at least a three-fold speedup and reduction in the RAM usage. We also point out that the construction of RLBWT has received much attention in recent years, whereas the practical approximation of LZ77 is a relatively unexplored topic, and hence significant speedup may be possible, e.g., via parallelization. The intuition for this is that, unlike in the case of BWT construction, LZ approximation algorithms need not be exact.

Acknowledgements

We thank Simon J. Puglisi for providing us with the kernel data set at short notice.

Funding

DK and BL were supported by NIH HG011392 and NSF DBI-2029552 grants.

References

1. Belazzougui Djamal, Gagie Travis, Gawrychowski Paweł, Juha Kärkkäinen Alberto Ordóñez Pereira, Puglisi Simon J., and Tabei Yasuo. Queries on LZ-bounded encodings. In DCC, pages 83–92. IEEE, 2015. doi:10.1109/DCC.2015.69.
2. Bingmann Timo, Fischer Johannes, and Osipov Vitaly. Inducing suffix and LCP arrays in external memory. In ALENEX, pages 88–102. SIAM, 2013. doi:10.1137/1.9781611972931.8.
3. Boucher Christina, Gagie Travis, Tomohiro I, Dominik Köppl, Langmead Ben, Manzini Giovanni, Navarro Gonzalo, Pacheco Alejandro, and Rossi Massimiliano. PHONI: Streamed matching statistics with multi-genome references. In DCC, pages 193–202. IEEE, 2021. doi:10.1109/DCC50243.2021.00027.
4. Boucher Christina, Gagie Travis, Kuhnle Alan, Langmead Ben, Manzini Giovanni, and Mun Taher. Prefix-free parsing for building big BWTs. Algorithms Mol. Biol, 14(1):13:1–13:15, 2019. doi:10.1186/s13015-019-0148-5. [PubMed: 31149025]
5. Burrows Michael and Wheeler David J. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
6. Charikar Moses, Lehman Eric, Liu Ding, Panigrahy Rina, Prabhakaran Manoj, Sahai Amit, and Shelat Abhi. The smallest grammar problem. Trans. Inf. Theory, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.
7. Claude Francisco and Navarro Gonzalo. Self-indexed grammar-based compression. Fundam. Informaticae, 111(3):313–337, 2011. doi:10.3233/FI-2011-565.
8. The 1000 Genomes Project Consortium. A global reference for human genetic variation. Nature, 526:68–74, 2015. doi:10.1038/nature15393. [PubMed: 26432245]
9. Dietzfelbinger Martin, Gil Joseph, Matias Yossi, and Pippenger Nicholas. Polynomial hash functions are reliable. In ICALP, pages 235–246, 1992. doi:10.1007/3-540-55719-9_77.
10. Ferragina Paolo, Gagie Travis, and Manzini Giovanni. Lightweight data indexing and compression in external memory. Algorithmica, 63(3):707–730, 2012.
11. Furuya Isamu, Takagi Takuya, Nakashima Yuto, Inenaga Shunsuke, Bannai Hideo, and Kida Takuya. MR-RePair: Grammar compression based on maximal repeats. In DCC, pages 508–517. IEEE, 2019. doi:10.1109/DCC.2019.00059.
12. Gagie Travis, Gawrychowski Pawel, Juha Kärkkäinen Yakov Nekrich, and Puglisi Simon J. LZ77-based self-indexing with faster pattern matching. In LATIN, pages 731–742. Springer, 2014. doi:10.1007/978-3-642-54423-1_63.
13. Gagie Travis, Gawrychowski Paweł, Juha Kärkkäinen Yakov Nekrich, and Puglisi Simon J. A faster grammar-based self-index. In LATA, pages 240–251. Springer, 2012. doi:10.1007/978-3-642-28332-1_21.
14. Gagie Travis, Tomohiro I, Giovanni Manzini, Navarro Gonzalo, Sakamoto Hiroshi, and Takabatake Yoshimasa. Rpair: Rescaling RePair with Rsync. In SPIRE, pages 35–44, 2019. doi:10.1007/978-3-030-32686-9_3.
15. Gagie Travis, Navarro Gonzalo, and Prezza Nicola. On the approximation ratio of Lempel-Ziv parsing. In LATIN, pages 490–503. Springer, 2018. doi:10.1007/978-3-319-77404-6_36.
16. Gagie Travis, Navarro Gonzalo, and Prezza Nicola. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM, 67(1):1–54, April 2020. doi:10.1145/3375890.
17. Juha Kärkkäinen Dominik Kempa, and Puglisi Simon J. Lempel-Ziv parsing in external memory. In DCC, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

18. Juha Kärkkäinen Dominik Kempa, and Puglisi Simon J. Parallel external memory suffix sorting. In CPM, pages 329–342. Springer, 2015. doi:10.1007/978-3-319-19929-0_28.
19. Karp Richard M. and Rabin Michael O. Efficient randomized pattern-matching algorithms. IBM J. Res. Dev, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.
20. Kempa Dominik and Kociumaka Tomasz. Resolution of the Burrows-Wheeler Transform conjecture. In FOCS, pages 1002–1013. IEEE, 2020. Full version: https://arxiv.org/abs/1910.10631.doi:10.1109/FOCS46700.2020.00097.
21. Kempa Dominik and Prezza Nicola. At the roots of dictionary compression: String attractors. In STOC, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.
22. Knuth Donald E. The Art of Computing, Vol. III, 2nd Ed. Addison-Wesley, 1998.
23. Dominik Köppl, Tomohiro I, Isamu Furuya, Yoshimasa Takabatake, Kensuke Sakai, and Keisuke Goto. Re-Pair in small space. Algorithms, 14(1):5, 2021. doi:10.3390/a14010005.
24. Kosolobov Dmitry, Valenzuela Daniel, Navarro Gonzalo, and Puglisi Simon J. LempelZiv-like parsing in small space. Algorithmica, 82(11):3195–3215, 2020. doi:10.1007/s00453-020-00722-6.
25. Jesper Larsson N. and Alistair Moffat. Off-line dictionary-based compression. Proceedings of the IEEE, 88(11):1722–1732, 2000.
26. Navarro Gonzalo. Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Comput. Surv, 54(2):29:1–29:31, 2021. doi:10.1145/3434399.
27. Navarro Gonzalo. Indexing highly repetitive string collections, part II: Compressed indexes. ACM Comput. Surv, 54(2):26:1–26:32, 2021. doi:10.1145/3432999.
28. Ohno Tatsuya, Sakai Kensuke, Takabatake Yoshimasa, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online RLBWT and its application to LZ77 parsing. J. Discrete Alg, 52–53:18–28, 2018. doi:10.1016/j.jda.2018.11.002.
29. Policriti Alberto and Prezza Nicola. LZ77 computation based on the run-length encoded BWT. Algorithmica, 80(7):1986–2011, 2018. doi:10.1007/s00453-017-0327-z.
30. Rytter Wojciech. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci, 302(1–3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.
31. Sakai Kensuke, Ohno Tatsuya, Goto Keisuke, Takabatake Yoshimasa, Tomohiro I, and Hiroshi Sakamoto. RePair in compressed space and time. In DCC, pages 518–527. IEEE, 2019. doi:10.1109/DCC.2019.00060.
32. Ziv Jacob and Lempel Abraham. A universal algorithm for sequential data compression. Trans. Inf. Theory, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

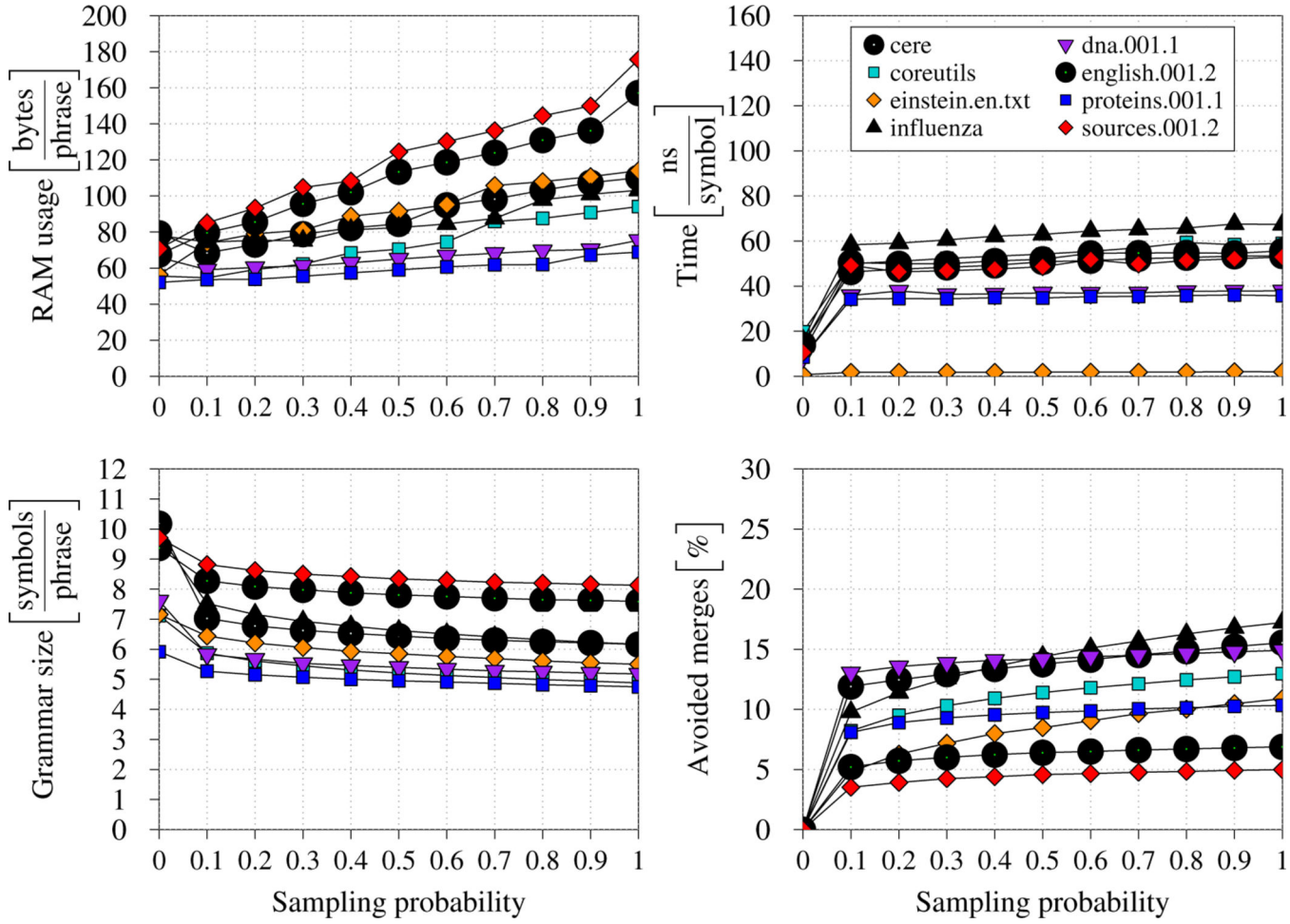

Figure 1.

Performance of the Lazy-AVLG algorithm for different values of the parameter p (probability of storing Karp–Rabin fingerprint in the hash table) on the files from Pizza & Chili corpus. The graphs in the top row show the normalized RAM usage (in bytes per phrase of the LZ77 parsing) and the runtime (in ns per symbols of the input text). The bottom row shows the resulting grammar size (the total length of right-hand sides of all productions) divided by z, and the percentage of merges avoided during the greedy merge procedure (in %).

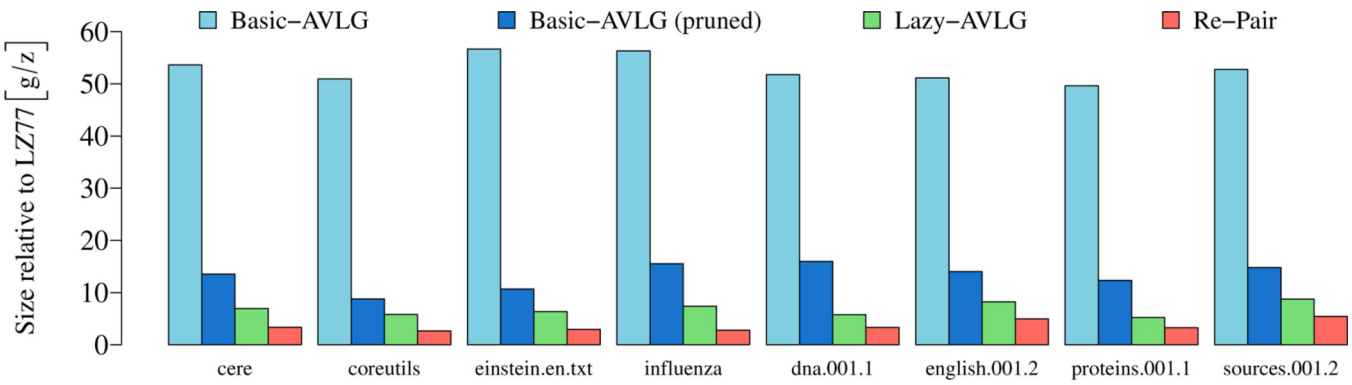

Figure 2.

Comparison of the size (measured as the total length of the right hand sides of all nonterminals) of grammars produced by the Basic-AVLG, Lazy-AVLG ($p = 0.125$), and Re-Pair algorithms on the files from the Pizza & Chili corpus. Basic-AVLG (pruned) denotes the size of grammar produced by Basic-AVLG with all nonterminals not reachable from the root removed. All sizes are normalized with respect to the size of the LZ77 parsing (z).

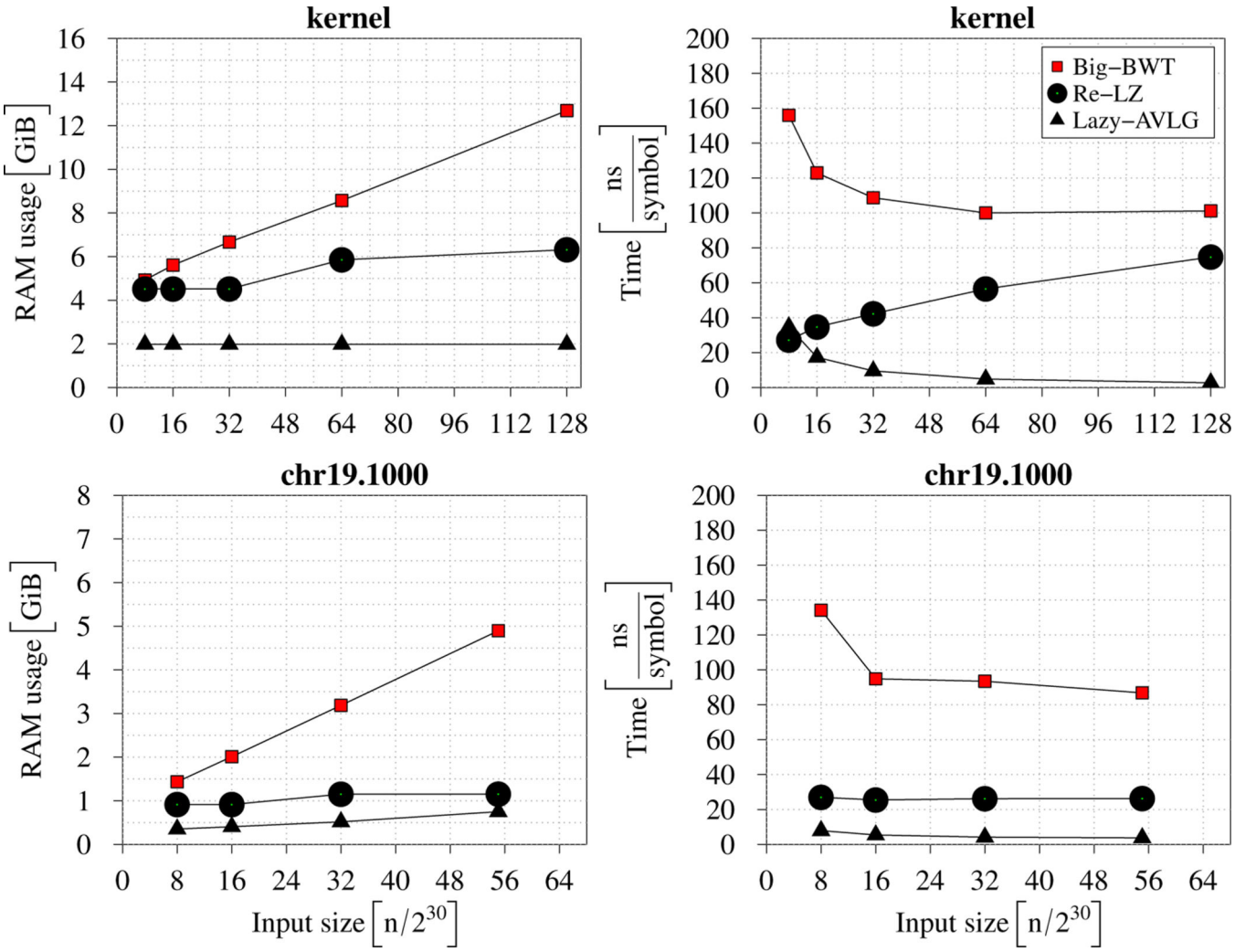

Figure 3.
Scalability of Big-BWT compared to Re-LZ and Lazy-AVLG. The graphs on the right show the normalized runtime in ns/char. The graphs on the left show the RAM usage in GiB.

Table 1

Statistics of files used in the experiments, with n denoting text length, σ denoting alphabet size, r denoting the number of runs in the BWT, and z denoting the number of phrases in the LZ77 parsing. For convenience, we also show the average BWT run length n/r and the average LZ77 phrase length n/z. Each of the symbols in the input texts is encoded using a single byte.

File name	n	σ	r	n/r	z	n/z
cere	461 286 644	5	11 574 640	39.85	1 700 630	271.24
coreutils	205 281 778	236	4 684 459	43.82	1 446 468	141.91
einstein.en.txt	467 626 544	139	290 238	1611.18	89 467	5 226.80
influenza	154 808 555	15	3 022 821	51.21	769 286	201.23
dna.001.1	104 857 600	5	1 716 807	61.07	308 355	340.05
english.001.2	104 857 600	106	1 449 518	72.33	335 815	312.24
proteins.001.1	104 857 600	21	1 278 200	82.03	355 268	295.15
sources.001.2	104 857 600	98	1 213 427	86.41	294 994	355.45
chr19.1000	59 125 116 167	5	45 927 063	1287.37	7 423 960	7964.09
kernel	137 438 953 472	229	129 506 377	1061.25	30 222 602	4547.55