Hierarchical Relative Lempel-Ziv Compression

Philip Bille
Department of Computer Science and Applied Mathematics,
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Simon R. Tarnow
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Abstract

Relative Lempel-Ziv (RLZ) parsing is a dictionary compression method in which a string S is compressed relative to a second string R (called the reference) by parsing S into a sequence of substrings that occur in R. RLZ is particularly effective at compressing sets of strings that have a high degree of similarity to the reference string, such as a set of genomes of individuals from the same species. With the now cheap cost of DNA sequencing, such datasets have become extremely abundant and are rapidly growing. In this paper, instead of using a single reference string for the entire collection, we investigate the use of different reference strings for subsets of the collection, with the aim of improving compression. In particular, we propose a new compression scheme hierarchical relative Lempel-Ziv (HRLZ) which form a rooted tree (or hierarchy) on the strings and then compress each string using RLZ with parent as reference, storing only the root of the tree in plain text. To decompress, we traverse the tree in BFS order starting at the root, decompressing children with respect to their parent. We show that this approach leads to a twofold improvement in compression on bacterial genome datasets, with negligible effect on decompression time compared to the standard single reference approach. We show that an effective hierarchy for a given set of strings can be constructed by computing the optimal arborescence of a completed weighted digraph of the strings, with weights as the number of phrases in the RLZ parsing of the source and destination vertices. We further show that instead of computing the complete graph, a sparse graph derived using locality-sensitive hashing can significantly reduce the cost of computing a good hierarchy, without adversely effecting compression performance.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Relative compression, Lempel-Ziv compression, RLZ, LZ77, string collections, compressed representation, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.18

Related Version *Previous Version*: `https://arxiv.org/abs/2208.11371`

Funding This work was supported in part by the Academy of Finland via grants 339070 and 351150 and the Danish Research Council via grant DFF-8021-002498.

1 Introduction

Given a collection of m strings $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of total length n, the *relative Lempel-Ziv (RLZ) compression scheme* parses each string S_i, $i > 1$, into a sequence of substrings (called *phrases*) of the string S_1 (we give a precise definition below). If the strings in S are highly similar, the number of phrases in the parsing is small relative to the total length of the collection. In order to achieve compression, the string S_1 is stored explicitly and the phrases of the other strings are encoded by their starting and ending positions in S_1. To decompress, we simply replace the encoding of each phrase by the corresponding substring in S_1. Note that we can even efficiently support *sequence retrieval*, that is, decompressing a single specified string S_i from $\mathcal{S}$, by simply decompressing the phrases of the S_i independently of the rest of the collection.

RLZ is ideal for compressing and storing collections of highly similar strings while supporting efficient sequence retrieval. In particular, RLZ is a natural choice for databases of full genome sequences of individuals of the same species [18, 6, 5, 36, 24]. Since these sequences are highly similar, RLZ is able to compress them well, while still supporting efficient sequence retrieval needed by applications. Enormous reductions over the past two decades in the cost of DNA sequencing has led to large and growing data bases containing hundreds of thousands of full genome sequences of strains of many known bacteria and viruses. These databases are key, for example, to the field of genomic epidemiology, to screen patient samples (which are sequenced and compared against the genome database) for known pathogenic strains to arrive at diagnosis and suitable treatments (see, e.g., [21]).

For a given bacterial species, a genome database may contain genome sequences of thousands of different strains. While all these strains are relatively similar to each other, some share a higher degree of similarity with genomes in a cluster of related strains than they do with other sequences in the database. With this is mind, while RLZ may result in good compression when a single arbitrary sequence is selected as the reference, intuitively it would seem that even more effective compression of the database could be achieved by selecting a different reference for each cluster of strains.

Our Contributions. In this paper we explore the use of more than one reference sequence in the context of RLZ compression. We present a new compression scheme, called *hierarchical relative Lempel-Ziv (HRLZ) compression*, that arranges the sequences in $\mathcal{S}$ into a rooted tree H, with root r, such that each node v corresponds to a unique string $S(v)$ from $\mathcal{S}$. To compress the collection we greedily parse $S(v)$ wrt. $S(\text{parent}(v))$ using RLZ for each non-root node v. The compressed representation then consists of $S(r)$, the edges of H, and the encoding of the $m-1$ parsings of the non-root strings. Note that RLZ may be viewed as the special case of HRLZ on a tree consisting of a root with $m-1$ children.

A key challenge in HRLZ compression is finding a hierarchy for the collection that achieves strong compression. We show how to adapt an approach from delta compression of string collections [25] to our relative compression scenario. This leads to a number of interesting algorithmic challenges:

1. To derive an effective hierarchical arrangement for a given set of strings, we compute the optimal arborescence of a complete weighted digraph of the strings, with edge weights assigned as the number of phrases in the RLZ parsing of the source and destination vertices. We show that this scheme leads to a factor of 2 improvement to compression on bacterial genomes, and up to a factor 10 on viral genomes, without adversely affecting the speed of decompression.
2. While the optimal arborescence leads to pleasing compression improvements, it adds significantly to compression time. We show that by sparsifying the graph via locality-sensitive hashing, compression time can be kept reasonable, while not sacrificing compression gains.
3. Along the way, we describe an efficient implementation of the optimal arborescence algorithm of Tarjan [32] that uses a two-level heap engineered for efficient meld operations, which may be of independent interest.

Our resulting HRLZ compression scheme achieves improved tradeoffs for genome databases. While the time to compress the database with HRLZ is slower than RLZ, HRLZ always improves the compression ratio (measured by the number of phrases) in some cases enormously – by a factor of up to 19 times in our experiments. HRLZ also matches or even slightly improves whole database decompression time and achieves very similar single sequence retrieval times (the time taken to extract a single requested genome from the database) as RLZ does. Thus, in practical scenarios where space and sequence retrieval time are the main bottlenecks (such as a genomic database) HRLZ provides an attractive alternative to RLZ.

We also compared HRLZ to the classic Lempel-Ziv 77 (LZ77) compression scheme. LZ77 provides a natural lower on the number of phrases we can hope to achieve with RLZ and HRLZ and thus provides a baseline for the compression ratio achieved by those schemes. Our experiments show that while HRLZ is larger than LZ77 (by a factor of 2 to 16) on small collections, HRLZ is able to scale to collections with sizes well beyond those LZ77 is able to process, and is also orders of magnitude faster for sequence retrieval compared to LZ77 on the all datasets we tested.

Related Work. The idea of constructing a hierarchy of compressed sequences from collections was proposed in the context of *delta-compression* [25]. To the best of our knowledge, this idea has not been explored for RLZ compression. A related notion is mentioned cryptically in Storer and Szymanski [31], but appears never to have been implemented. The closest work we could find in the literature is due to Deorowicz and Grabowski [6], who describe an RLZ-based scheme for genome compression in which a sequence is compressed relative to multiple reference sequences, with each phrase storing which reference sequence it is from (see also [5]). Another more recent hierarchical compression scenario is the persistent strings model [3]. Both of these are quite different to the hierarchical arrangement of sequences we describe here.

Beyond genomics applications, RLZ has also found wider use as a compressor for large text corpora in contexts where random-access support for individual documents is needed [14, 34, 35, 26, 20, 2] and as a general data compressor [17, 16]. In those contexts, S_1 is usually first constructed using substrings sampled from other strings in the collection in a preprocessing phase (Hoobin et al. [14] show that random sampling of substrings works well). The structure of the RLZ parsing reveals a great deal about the repetitive structure of the string collection and several authors have shown that this can be exploited to design efficient compressed indexes for pattern matching [13, 8, 23]. More recently, the practical utility of RLZ as a more general tool for compressed data structuring has also been demonstrated, compressing suffix arrays [27, 29], document arrays [28] and various components of suffix trees [9].

Outline. In Section 2 we set down notation and basic concepts used throughout. In Section 3 we formally define hierarchical relative Lempel-Ziv compression, and then go onto describe efficient methods for computing it in Section 4 and Section 5. Section 6 describes our engineering of the arborescence algorithm of Tarjan [32]. Our experimental results on three genomic datasets are presented in Section 7, before conclusions and reflections are offered.

2 Basics

Throughout we will consider a *string* $S = S[1..n] = S[1]S[2]\dots S[n]$ on an integer alphabet Σ of σ symbols. The *substring* of S that starts at position i and ends at position j, $j \geq i$, denoted $S[i..j]$, is the string $S[i]S[i+1]\dots S[j]$. If $i > j$, then $S[i..j]$ is the empty string ε. A suffix of S is a substring with ending position $j = n$, and a prefix is a substring with starting position $i = 1$.

Parsings. A *parsing* of a string S wrt. a reference string R is a sequence of substrings of R – called phrases – $R[i_1, i_1 + l_1 - 1], R[i_2, i_2 + l_2 - 1], \ldots, R[i_z, i_z + l_z - 1]$ such that $S = R[i_1, i_1 + l_1 - 1] \cdot R[i_2, i_2 + l_2 - 1] \cdots R[i_z, i_z + l_z - 1]$. The *encoding* of a parsing consists of the sequence of starting indices and lengths of the phrases $(i_1, l_1), (i_2, l_2), \ldots, (i_z, l_z)$.

The *greedy parsing* of S wrt. R is the parsing obtained by processing S from left to right and choosing the longest possible phrase at each step. For example, let $R = actccta$ and $S = ctctcc$. The greedy parsing of S wrt. R gives the phrases $R[2,4] = ctc$, $R[3,5] = tcc$ and the encoding $(2,3),(3,3)$. We can construct the parsing in $O(|R| + |S|)$ time using a suffix tree.

Relative Lempel-Ziv Compression. Throughout the rest of the paper let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings of total length $n = \sum_{i=1}^{m} |S_i|$. The *relative Lempel-Ziv* (RLZ) compression of $\mathcal{S}$ greedily parses each string S_i, $i > 1$, wrt. S_1. The *RLZ compressed representation* of $\mathcal{S}$ then consists of S_1 and the encoding of the parsings of each of the strings $S_2, \ldots, S_m$. For each string, we also save the number of phrases. In total, compression takes $O(n)$ time. Let z_i be the number of phrases in the parsing of S_i and let $z_R = \sum_{i=2}^{m} z_i$ denote the total number of phrases. The size of the RLZ compression is thus $O(|S_1| + z_R)$. Note that the size depends on the choice of the reference string (i.e. S_1) among the strings in $\mathcal{S}$. To decompress, we decode the phrases of each string using the explicitly stored reference string. This uses $O(\sum_{i=1}^{m} |S_i|) = O(n)$ time.

Throughout this paper, we use the number of phrases as the measure of compression. In a real compressor, the phrase positions and lengths undergo further processing in order to reduce the total number of bits used by the encoding (see, e.g., [11]). We remark that our hierarchical RLZ methods can be trivially adapted to use different encoding costs.

Graphs. Let G be a weighted directed strongly connected graph G. A *spanning arborescence* A of G with root r is a subgraph of G that is a directed rooted tree where all nodes are reachable from r. The weight of an arborescence S is the sum of the weight of the edges in A. A *minimum weight spanning arborescence* (MWSA) A is a spanning arborescence of minimum weight. Note that the root is not fixed in our definition and can thus be any node. For simplicity, we have assumed that G is strongly connected in our definition of MWSA since this is always the case in our scenario. Finally, for a node v in a tree, the parent of a node is denoted parent(v).

3 Hierarchical Relative Lempel-Ziv Compression

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of strings of total size n as above. We construct a rooted tree H, with root r, such that each node v represents a unique string $S(v)$ from $\mathcal{S}$. The *hierarchical relative Lempel-Ziv* (HRLZ) compression of $\mathcal{S}$ wrt. H greedily parses $S(v)$ wrt. $S(\text{parent}(v))$ for each non-root node v. In total, compression takes $O(n)$ time. The *HRLZ compressed representation* consists of $S(r)$, the edges of H, and the encoding of the $m-1$ parsings of the non-root strings. Let $z_H = \sum_{v \in H \setminus \{r\}} z_v$, where z_v is the number of phrases in the parsing of $S(v)$. Thus the size of the HRLZ compression is $O(|H| + |S(r)| + z_H) = O(|S(r)| + z_H)$. Note that the size depends on the choice of tree and assignment of strings from $\mathcal{S}$ to the nodes.

To decompress, we traverse the tree in breadth first search (BFS) order from the root. We decode the string at each node using the string of the parent node by decoding each phrase. As the string of the parent node is always decoded before or explicitly stored as the root node, this uses $O(\sum_{i=1}^{m} |S_i|) = O(n)$ time. Note that the output order of the sequences can differ from their input order since they are recovered in BFS order of the hierarchy.

4 Constructing an Optimal Tree

We first give a simple and inefficient algorithm to construct an optimal tree for the HRLZ compression. The algorithm forms the basis of our efficient algorithm in the following section. Recall that the collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ consists of m strings of total size n. The algorithm proceeds as follows:

Step 1: Construct Cost Graph. We first construct a complete weighted directed graph G with m nodes numbered $\{1, \ldots, m\}$ called the *cost graph* of $\mathcal{S}$. Node i corresponds to the string S_i in $\mathcal{S}$ and the weight of edge (i, j) is the number of phrases in the greedy parsing of S_j wrt. S_i.

We have that G contains m nodes and m^2 edges. Computing the weight of edge (i, j) takes $O(|S_j|)$ time. Thus in total we use $O((m-1)\sum_j |S_j|) = O(nm)$ time. The space is $O(m^2)$.

Step 2: Construct Minimum Weight Spanning Arborescence. We then construct a MWSA A of the cost graph G using the algorithm by Tarjan [32, 4]. This uses $O(e \log m) = O(m^2 \log m)$ time. Here e denotes the number of edges in the graph.

Step 3: Construct Compressed Representation. Finally, we construct the HRLZ compression from the MWSA A. This uses $O(n)$ time.

In total the algorithm uses $O(mn)$ time and $O(m^2)$ space. Note that the algorithm constructs an optimal tree but not necessarily the optimal HRLZ compression since the HRLZ compression also needs to explicitly encode the string of the root of the tree. It is straightforward to include the cost of encoding the root string in the algorithm, by adding an additional virtual root s and adding edges (s, i) to every other node i, $1 \leq i \leq m$, with weight $|S_i|$. The MWSA of the new graph G' will be rooted in s and the unique edge out of s determines the optimal root string for HRLZ compression. While G' is not strongly connected, the MWSA is still well-defined and the MWSA algorithm produces the correct result in the same complexity. In practice, our datasets consist of very similar length strings and hence we have chosen not to implement this extension.

5 Sparsifying the Cost Graph via Locality-Sensitive Hashing

The main bottleneck in the simple algorithm from Section 4 is the construction of the complete cost graph in Step 1. In this section, we show how to efficiently sparsify the graph using locality-sensitive hashing.

We first construct a sparse subgraph $\overline{G}$ of the complete cost graph. We do this in rounds keeping an auxiliary set of strings R as follows. Initially, we set $\overline{G}$ to be the graph with m nodes and no edges, and $R = \mathcal{S}$. We repeat the following steps until $\overline{G}$ is strongly connected.

Step 1: Generate fingerprints. We first generate fingerprints for each string in R using locality-sensitive hashing. Our locality-sensitive hashing scheme is based on k-mers (a substring of length k) combined with min-hashing. More precisely, given parameters k and q we pick q hash functions $h_1, \ldots, h_q$ and hash each k-mer of each string S in R. The fingerprint of S is the sequence $\min_1, \ldots, \min_q$ where $\min_i$ is a minimum value hash obtained with h_i. For fast hashing we use the simple *multiply-shift* hashing scheme [7].

Step 2: Generate edges. Let C be a group of strings in R with the same fingerprint. If $|C| \leq T$ for a threshold $T \geq 2$ then for each ordered pair (i, j) of strings in C we add (i, j) to $\overline{G}$.

Step 3: Pruning R. After every c-th round for some tuneable parameter c we prune the set R as follows.

For every connected component in $\overline{G}$ pick the string s that has had the most collisions until now (the total number of collisions of a string s is equal to the sum of the size of the buckets it has been in). We then continue with R being the set of representatives. If $|R| \leq T$ then for each ordered pair (i, j) of strings in R we add (i, j) to $\overline{G}$.

Finally, we compute the weight of the edges of the strongly connected graph $\overline{G}$, i.e., for each edge (i, j) we compute the number of phrases in the greedy parsing of S_j wrt. S_i.

The computed cost graph is likely to be sparse and thus step 1 and step 2 of the algorithm from Section 4 will be much faster, leading to a much faster solution. Note that the constructed tree is no longer guaranteed to be optimal. We show experimentally in Section 7 that the size of the compression in nearly all cases is within 5% of optimal.

6 Speeding Up the Minimum Weight Spanning Arborescence Algorithm

We now show how to efficiently implement Step 2 of the algorithm from Section 4 on the sparse cost graph computed in Section 5.

Let $\overline{G}$ be the sparse cost graph with m nodes and e edges computed in Section 5. The MWSA algorithm by Tarjan [32, 4] uses m priority queues $Q_1, Q_2, \ldots, Q_m$, one for each node, where Q_i consists of all edges going into node v_i. The queues support the following operations:

- init(L): Constructs a queue Q containing all the elements in the list L.
- extract-min(Q_i): Returns and removes the minimum element in the queue Q_i.
- add(Q_i, c): Adds a constant c to the value of all elements in the queue Q_i.
- meld(Q_i, Q_j): Adds the elements from queue Q_j to the queue Q_i.

The MWSA algorithm uses a *pairing heap* [12] to support init in $O(|L|)$ time and the other operations in $O(\log m)$ time. The algorithm uses $O(m)$ meld and init operations, $O(e)$ add and extract-min operations, and the total length of the lists for the init operations is $O(e)$. Thus, the total run time of the queue operations in the MWSA algorithm is $O(e \log m)$, and this is also the total runtime of the MWSA algorithm.

We present a simple and practical alternative to the pairing heap that we call a *two-level heap*. Our two-level heap leads to a slightly worse theoretical bound of $O(e \log m + m \log^2 m)$ time for the MWSA algorithm. However, we have found that our implementation significantly outperforms the pairing heap in practice. We note that Larkin, Sen, and Tarjan include a similarly modified pairing heap as one of the variants in their empirical study of priority queues [19]. That study, however, neglects the meld operation, which is essential to our implementation of the minimum weight spanning aborescense algorithm described above. We therefore now describe our two-level heap implementation in full.

The two-level heap consists of a *top heap* t and a list of q *bottom heaps* $B = \{b_1, b_2, \ldots b_q\}$. All heaps are implemented using standard binary heaps [37]. Each heap h has an associated *offset* o_h, such that any stored element x in h represents that actual value $x + o_h$. The top heap t consists of the minimum element in each bottom heap $b \in B$. For each element in the top heap we also store which botton heap it is from. We implement each of the operations as follows.

init(L). We construct a two-level heap consisting of a single bottom heap $B = \{b\}$ containing the elements of L and a top heap t containing the minimum element of b. We set the offsets o_b and o_t of b and t, respectively, to be 0. This uses $O(|L|)$ time and hence the total time for init in the MWSA algorithm is $O(e)$.

extract-min(Q_i). We extract the minimum element x from the top heap t and return $x + o_t$. Let b be the bottom heap that stored x. We extract x from b, find the new minimum element y in b, and copy y into the top heap with offset o_b. This uses $O(\log m)$ time and hence the total time for extract-min in the MWSA algorithm is $O(e \log m)$.

add(Q_i, c). We add c to the offset of the top heap t, i.e., we set $o_t = o_t + c$. This takes constant time and hence the total time for add in the MWSA algorithm is $O(e)$.

meld(Q_i, Q_j). Let $Q_i = (t_i, B_i)$ and $Q_j = (t_j, B_j)$ be the two-level heaps that we want to meld. Let $|B_i|$ and $|B_j|$ be the number of bottom heaps associated with two-level heap Q_i and Q_j, respectively, and assume wlog. that $|B_i| \geq |B_j|$. We move each bottom heap $b \in B_j$ into B_i, insert the minimum element of b into t_i with offset $o_b + o_{t_j} - o_{t_i}$, and update the offset associated with b to $o_b = o_b + o_{t_j} - o_{t_i}$.

Each time an element in a bottom heap b is moved, we must insert the minimum element of b into a top heap using $O(\log m)$ time. We only move the bottom heaps of the two-level heap with the fewest bottom heaps and hence the number of times a bottom heap can be moved is $O(\log m)$. It follows that total time for meld in the MWSA algorithm is $O(m \log^2 m)$.

In total the MSWA algorithm implemented with the two-level heap uses $O(e \log m + m \log^2 m)$ time.

7 Experimental Results

We implemented the methods for building hierarchical references described in the previous sections and measured their performance on real biological data.

7.1 Setup

Experiments were run on Nixos 21.11 kernel version 5.10.115. The compiler was `g++` version 11.3.0 with `-Wall -Wextra -pedantic -O3 -funroll-loops -DNDEBUG -fopenmp -std=gnu++20` options. OpenMP version 4.5 was used to compute the string fingerprints in parallel and compute the edge weights on the cost graph. The CPU was an AMD Ryzen 3900X 12 Core CPU clocked at 4.1 GHz with L1, L2 caches of size 64KiB, 512KiB, per core respectively and a shared L3 cache of size 64MiB. The system had 32GiB of DDR4 3600 MHz memory. We recorded the CPU wall time using GNU time and `C++` chrono library. Source code is available on request.

7.2 Datasets

We evaluated our method using 1,000 copies of human chromosome 19 from the 1000 Genomes Project [33]; 219 *E. coli* genomes taken from the GenomeTrakr project [30], and 400,000 *SARS-CoV2* genomes from EBI's COVID-19 data portal [1]. See Table 1 for a brief summary of the datasets.

We also ran our tests on prefixes of various sizes of these datasets.

Table 1 Datasets used in experimentation. Columns labelled σ, n, and m, give the alphabet size, total collection size, and number of sequences, respectively. The final column shows the average sequence length, for convenience.

Name	Description	σ	n	m	n/m
E.coli	*E.coli* genomes	4	$1,130,374,882$	219	$5,161,529$
SARS-CoV2	Covid-19 genomes	5	$11,949,531,820$	$400,000$	$29,873$
chr19	Human chromosome 19 assemblies	5	$59,125,151,874$	$1,000$	$59,125,151$

7.3 Methods Tested

We included the following methods in our experimental evaluation.

RLZ. This corresponds to standard, single reference RLZ. Because the choice of reference can affect overall compression, we report results across a number of reference selections. Specifically, we randomly sampled roughly 0.5% of the sequences of each dataset, corresponding to 2, 2000, and 5 different reference sequences from *E. coli*, *SARS-CoV2*, and the *chr19* dataset respectively as our reference in RLZ.

Optimal HRLZ. This is the method described in Section 4, i.e., optimal hierarchical RLZ making use of full weight information.

Approximate HRLZ. The LSH variant of hierarchical RLZ as described in Section 5. Specifically, we used k-mers of 256 characters in size and choose the number of hash functions to $q = 4$. We pruned the set R every $c = 10$ rounds. We used $T = 2 \cdot \sqrt{m}$ as our threshold.

LZ. As a compression baseline, we also compute the full LZ77 parsing of our datasets using the KKP-SE external memory algorithm and software of [15][1]. Because it allows phrases to have their source at any previous position in the collection, computing the LZ77 parsing is more computationally demanding than RLZ parsing, and so we compute it only for some prefixes of the collections. For similar reasons, although in principle the above RLZ-based methods could attain parsings as small as the LZ77 parsing, we expect them not to.

7.4 Compression Performance

In this section, we compare the compression size as measured by the number of phrases generated by single reference RLZ to Optimal HRLZ and Approximate HRLZ on the datasets with LZ as a baseline. We also compare the compression time of the algorithms.

The results of our compression experiments are shown in Figure 1. Some of the results for the compression size of RLZ on the *SARS-CoV2* dataset have been left out of the figure because they were orders of magnitude larger than the other algorithms. Furthermore, we were unable to get data points for LZ on the 1000 sequences of the *chr19* dataset and the Optimal HRLZ on the *SARS-CoV2* for anything more than 125000 sequences on our test machine. The specific values behind Figure 1 (including the leftout results for RLZ) can be read in Tables 2–7 in the appendix.

[1] Code available at `https://www.cs.helsinki.fi/group/pads/em_lz77.html`.

We observe that, as expected LZ consistently produces the best compression size, but was infeasible to run on the full *chr19* dataset – which is the largest dataset measured in the number of symbols – on our test machine due to space consumption.

While RLZ outperforms all other algorithms in regard to compression speed it also consistently produces the worst compression size. Results on the *SARS-CoV2* dataset show just how bad the compression size can deteriorate if an ill-fitting reference is chosen as the reference used by RLZ. We observe that both versions of HRLZ obtain a better compression that RLZ in all cases. Measured in the number of phrases approximate HRLZ improves the compression ratio by a factor 1.8 on the *E. coli* dataset and 19.3 on the *SARS-CoV2* dataset.

For the Optimal HRLZ we see that the compression time is the worst of all the algorithms and grows quadratically with the number of sequences and quickly becomes infeasible on the *SARS-CoV2* dataset. It does however consistently produce a smaller compression size than RLZ, as we would expect given that HRLZ could produce a star graph with a single sequence as the reference for all other sequences and thus yielding the same result as RLZ. The compression time for Approximate HRLZ is significantly less than Optimal HRLZ and is consistently within a factor 2 of LZ – even outperforming LZ on some of the larger experiments of the *chr19* dataset. Furthermore, we note that the compression size of the Approximate HRLZ is no more than 15% greater than the compression size of Optimal HRLZ and always better than that of RLZ.

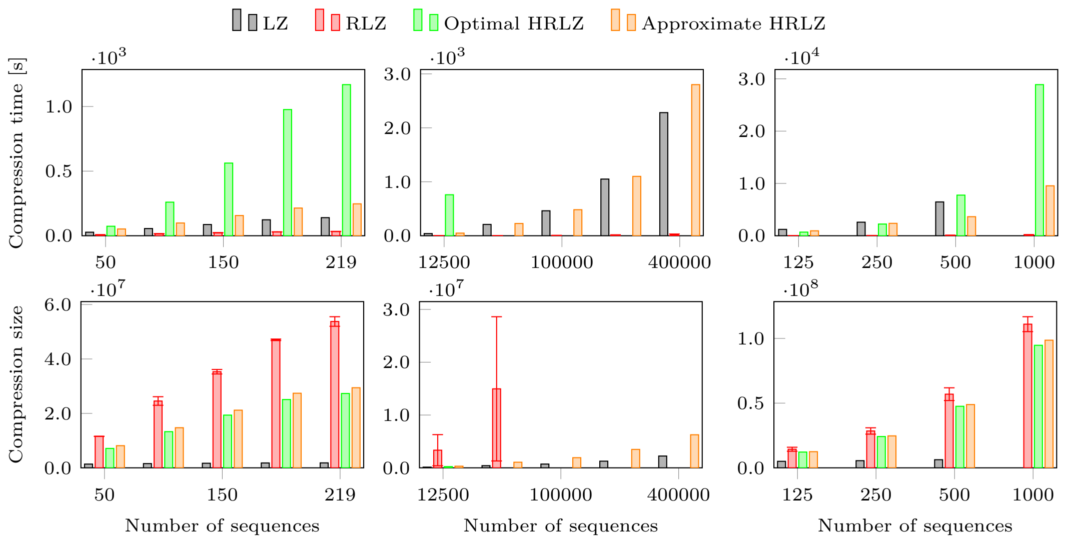

Figure 1 Compression time (top) and compression size (bottom) measured in the number of phrases as a function of the number of sequences in the in the *E. coli* (left), *SARS-CoV2* (center) and *chr19* (right) dataset. RLZ compression sizes were left out from the *SARS-CoV2* (center) plot because it is orders of magnitude larger than the rest of the compression sizes. We were unable to get data points for LZ on the 1000 sequences of the *chr19* dataset and Optimal HRLZ on the *SARS-CoV2* for anything more than 12500 sequences on our test machine due to memory consumption.

7.5 Decompression Performance

In this section, we compare the decompression time of the algorithms. The experiment measured the time to decompress the generated compressed dataset from our compression experiments and write the result to disk. This kind of streaming decompression use case is typical for, e.g., multi-pass index construction, machine learning, and data mining processes [10]. All methods have to write the same amount of data to storage when decompressing. Our HRLZ variants decompress sequences in BFS order according to hierarchy imposed on the sequences. This may require sequences that have previously written to disk being read back into memory (at most once) when they are needed as a reference in the decompression of other sequences.

We also compared the time it takes to decompress a single sequence for both variants of HRLZ and RLZ. We call this *sequence retrieval time.* For RLZ we took the best compressed sample for each experiment and used this for the test. We did not implement an equivalent solution for LZ since LZ is not built to easily decompress a single sequence. For each experiment we found the average and standard deviation of the sequence retrieval time over all sequences for that specific experiment.

The results of our decompression and sequence retrieval experiments are shown in Figure 2. We do not have data points for the experiments that we where unable to perform compression on. The specific values behind Figure 2 can be read in Tables 8–13 in the appendix. For Optimal HRLZ and Approximate HRLZ we also recorded the average and maximum node depth of the nodes in the arborescence. This can be viewed in table 3 in the appendix.

We observe that RLZ and the HRLZ variants have similar decompression performance characteristics, while LZ performs significantly worse. Interestingly, both versions of HRLZ outperformed RLZ in decompression time on most of the experiments. We believe this is because the longer phrases produced by the HRLZ variants result in fewer cache misses; that is, for every access to the reference made by HRLZ, more symbols are sequentially copied, improving cache performance and overall runtime. We believe that this also explains why both variants of HRLZ perform within a factor of 5 with respect to RLZ in sequence retrieval time, even though the average node depth in the minimum spanning arborescence on the largest *SARS-CoV2* dataset experiment is 100.

On larger datasets, which approach the size of RAM on the test machine, RLZ and HRLZ have similar decompression times, with RLZ being occasionally faster (on the largest of the *SARS-CoV2* datasets, for example). This can be explained by the above mentioned need for the HRLZ variants to read previously written sequences back into memory (when they are needed as reference sequences).

8 Concluding Remarks

We have shown that, from a space point of view, traditional single-reference RLZ compression can be significantly outperformed by imposing a hierarchy on the sequences to be compressed using a sequence's parent in the hierarchy as its reference sequence. Moreover, we have described efficient methods by which hierarchies can be efficiently obtained. Our experiments show that the time to subsequently decompress the set of sequences are at worst negligibly slower, and many times even faster than the single reference baseline.

There are many directions future work could take. Apart from compression, another feature of RLZ that makes it attractive in a genomic context is its ready support for efficient random access to individual sequences (and indeed substrings): having a compact, easily accessible representation of the genome sequences compliments popular indexing methods that do not readily support random access themselves (e.g., [22]). Supporting random access at a substring level for HRLZ compressed data (as opposed to sequence-level access we currently support) is an interesting avenue for future work.

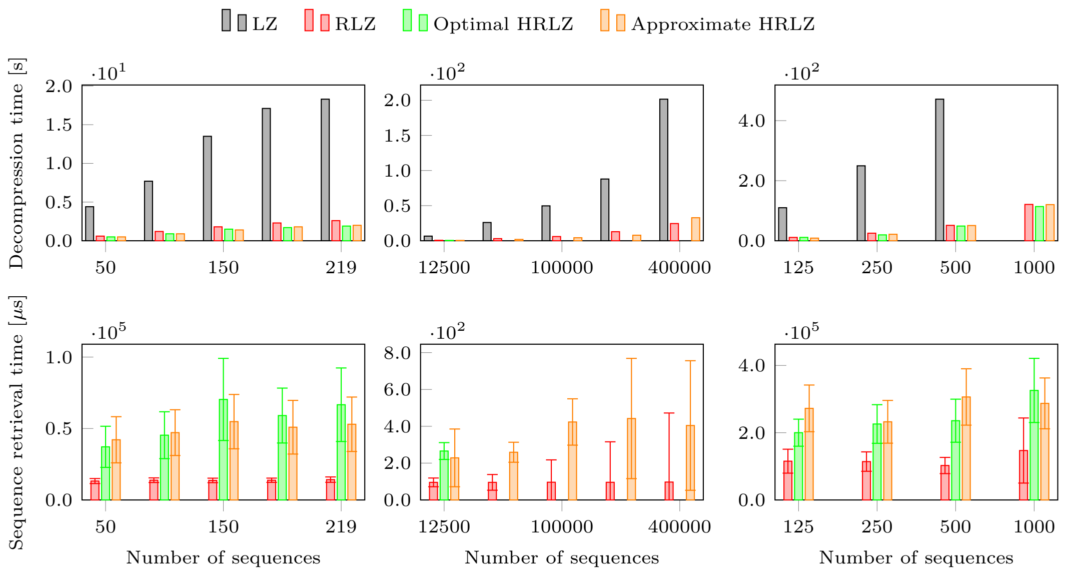

Figure 2 Decompression time (top) and sequence retrieval time (bottom) as a function of the number of sequences in the *E. coli* (left), *SARS-CoV2* (center) and *chr19* (right) dataset. Note that the missing samples are due to no compressed result being available.

As noted in the introduction, several recent works have demonstrated the practical utility of using RLZ as a tool for compressed data structuring [9, 27, 29, 28]. In that context, an *artificial* reference sequence is constructed from repeated pieces (e.g., subarrays, or subtrees) of the data structure to be compressed. It would be interesting to see if our methods could be adapted to construct better artificial reference sequences for use in those scenarios.

Finally, in this work we have used purely algorithmic methods to derive hierarchies for datasets: no biological characteristics of the sequences have been used. However, the field of phylogenetics has developed many techniques for imposing a hierarchy on a set of individuals based on biologically meaningful features in their genomic content. It may be interesting to examine any similarities between phylogenetic trees and our RLZ-based hierarchies, and whether phylogenetic trees may serve as good hierarchies in the context of compression.

References

1 Coronavirus genomes – NCBI datasets. Accessed 18/05/2022, `https://www.ncbi.nlm.nih.gov/datasets/coronavirus/genomes/`.

2 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

3 Philip Bille and Inge Li Gørtz. Random access in persistent strings. In *Proc. 31st ISAAC*, 2020.

4 P. M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9(4):309–312, 1979. `doi:10.1002/net.3230090403`.

5 Sebastian Deorowicz, Agnieszka Danek, and Szymon Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

6 Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011.

7 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

8 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theor. Comput. Sci.*, 532:14–30, 2014.

9 Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Relative suffix trees. *Comput. J.*, 61(5):773–788, 2018.

10 Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proc. 3rd WSDM*, pages 391–400, 2010.

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

12 Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

13 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.

14 Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endowment*, 5(3):265–273, 2011.

15 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th DCC*, pages 153–162, 2014.

16 Dominik Kempa and Ben Langmead. Fast and space-efficient construction of AVL grammars from the LZ77 parsing. In *Proc. 29th ESA*, pages 56:1–56:14, 2021.

17 Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. Lempel-ziv-like parsing in small space. *Algorithmica*, 82(11):3195–3215, 2020.

18 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

19 Daniel H. Larkin, Siddhartha Sen, and Robert Endre Tarjan. A back-to-basics empirical study of priority queues. In *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. SIAM, 2014.

20 Kewen Liao, Matthias Petri, Alistair Moffat, and Anthony Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proc. 25th WWW*, pages 807–816, 2016.

21 Tommi Mäklin, Teemu Kallonen, Jarno Alanko, Ørjan Samuelsen, Kristin Hegstad, Veli Mäkinen, Jukka Corander, Eva Heinz, and Antti Honkela. Bacterial genomic epidemiology with mixed samples. *Microbial Genomics*, 7(11), 2021.

22 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *J. Comput. Biol.*, 27(4):514–518, 2020.

23 Gonzalo Navarro and Victor Sepulveda. Practical indexing of repetitive collections using relative Lempel-Ziv. In *Proc. 29th DCC*, pages 201–210, 2019.

24 Gonzalo Navarro, Victor Sepulveda, Mauricio Marín, and Senén González. Compressed filesystem for managing large genome collections. *Bioinformatics*, 35(20):4120–4128, 2019.

25 Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Proc. 3rd WISE*, pages 257–266, 2002.

26 Matthias Petri, Alistair Moffat, P. C. Nagesh, and Anthony Wirth. Access time tradeoffs in archive compression. In *Proc. 11th AIRS*, pages 15–28, 2015.

27 Simon J. Puglisi and Bella Zhukova. Relative Lempel-Ziv compression of suffix arrays. In *Proc. SPIRE*, LNCS 12303, pages 89–96. Springer, 2020.

28 Simon J. Puglisi and Bella Zhukova. Document retrieval hacks. In *Proc. 19th SEA*, pages 12:1–12:12, 2021.

29 Simon J. Puglisi and Bella Zhukova. Smaller RLZ-compressed suffix arrays. In *Proc. 31st DCC*, 2021.

30 E.L. Stevens et al. The public health impact of a publically available, environmental database of microbial genomes. *Front. Microbiol.*, 8(808), 2017.

31 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.
32 R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977. doi:10.1002/net.3230070103.
33 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.
34 Jiancong Tong, Anthony Wirth, and Justin Zobel. Compact auxiliary dictionaries for incremental compression of large repositories. In *Proc. 23rd CIKM*, pages 1629–1638, 2014.
35 Jiancong Tong, Anthony Wirth, and Justin Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. 37th SIGIR*, pages 283–292, 2014.
36 Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC Genom.*, 19(S2), 2018.
37 John William Joseph Williams. Algorithm 232: heapsort. *Commun. ACM*, 7:347–348, 1964.

A Additional Figures

This appendix contains additional plots as well as data used to generate plots in the main document.

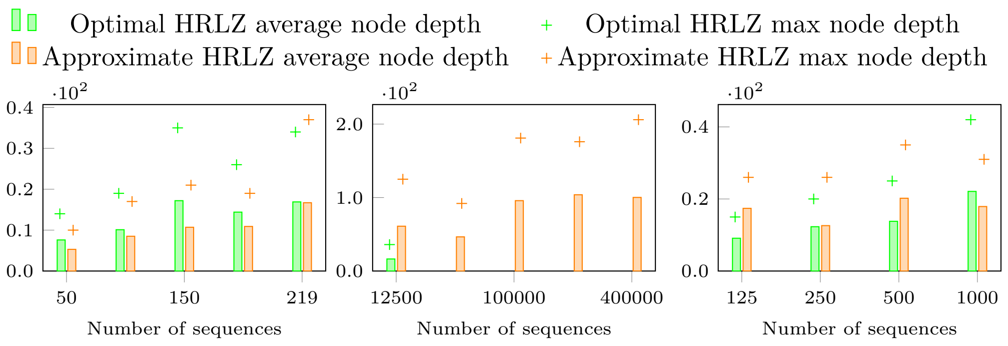

Figure 3 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *E. coli* (left), *SARS-CoV2* (center) and *human chromosome 19* (right) dataset.

Table 2 Number of phrases generated for each algorithm as a function of the number of sequences on the *E. coli* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
50	1379061	11562810.5	19485.7	7119429	8136762
100	1556541	24551113.5	1575921.1	13284440	14736830
150	1681574	35311948.5	820511.9	19369695	21180548
200	1787317	47030288.0	281705.7	25087136	27401597
219	1815168	53757548.5	1759746.2	27307037	29425274

Table 3 Compression time in seconds for each algorithm as a function of the number of sequences on the *E. coli* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
50	27.4	7.2	0.1	73.4	52.5
100	55.7	15.1	0.6	259.9	98.7
150	86.8	23.0	0.6	562.3	156.3
200	123.4	29.5	0.0	976.3	214.4
219	140.0	32.9	0.8	1169.2	246.9

Table 4 Number of phrases generated for each algorithm as a function of the number of sequences on the *SARS-CoV2* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
12500	133149	3339721.0	2948529.2	227037	308499
50000	408189	14958546.9	13657427.6	nan	1061000
100000	714174	29867903.0	27657360.9	nan	1919686
200000	1262731	62437789.1	63333546.4	nan	3483747
400000	2235801	120997669.9	112955869.2	nan	6260161

Table 5 Compression time in seconds for each algorithm as a function of the number of sequences on the *SARS-CoV2* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
12500	40.1	0.5	0.3	759.6	49.0
50000	209.5	2.3	1.6	nan	226.8
100000	462.5	4.5	3.3	nan	482.5
200000	1050.5	9.5	8.0	nan	1099.8
400000	2281.7	18.5	13.9	nan	2800.4

Table 6 Number of phrases generated for each algorithm as a function of the number of sequences on the human chromosome 19 dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
125	5024871	14364972.4	1550636.6	12196621	12470163
250	5526491	28489855.6	2453053.5	24174861	24688193
500	6263992	56926699.6	4927584.0	47547689	48888371
1000	nan	111021292.4	5760037.6	94684015	98637301

Table 7 Compression time in seconds for each algorithm as a function of the number of sequences on the human chromosome 19 dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
125	1215.0	24.8	1.8	694.7	941.9
250	2591.2	46.3	3.3	2238.0	2366.1
500	6463.0	89.1	5.9	7765.9	3641.0
1000	nan	187.4	6.8	28886.1	9546.1

Table 8 Decompression time in seconds as a function of the number of sequences in the *E. coli* dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
50	4.4	0.576	0.466	0.497
100	7.7	1.193	0.902	0.942
150	13.5	1.771	1.479	1.407
200	17.1	2.257	1.719	1.782
219	18.3	2.642	1.913	1.988

Table 9 Decompression time in seconds as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
12500	6.6	0.680	0.522	0.483
50000	26.0	2.360	nan	1.900
100000	49.7	4.492	nan	4.366
200000	87.9	13.252	nan	7.852
400000	201.6	26.827	nan	32.835

Table 10 Decompression time in seconds as a function of the number of sequences in the human chromosome 19 dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
125	110.0	11.071	11.186	8.759
250	249.6	25.200	19.592	21.600
500	471.5	51.297	49.011	50.697
1000	nan	120.984	113.942	120.307

Table 11 Sequence retrieval time in microseconds as a function of the number of sequences in the *E. coli* dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
50	13236.5	1755.6	37178.5	14394.6	42121.3	16156.5
100	13821.5	1620.0	45319.2	16403.6	47117.6	16014.0
150	13676.9	1570.5	70350.0	28741.0	54822.1	19014.8
200	13793.4	1494.6	59070.6	19208.5	50908.6	18783.8
219	14224.6	1889.1	66639.9	25731.6	52970.7	19047.7

Table 12 Sequence retrieval time in microseconds as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
12500	95.5	23.7	265.7	45.7	228.5	156.8
50000	95.5	42.7	nan	nan	259.0	54.3
100000	96.2	121.4	nan	nan	423.4	125.8
200000	95.8	220.0	nan	nan	442.3	326.5
400000	97.2	375.0	nan	nan	404.3	351.8

Table 13 Sequence retrieval time in microseconds as a function of the number of sequences in the human chromosome 19 dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
125	115321.9	35591.1	200032.1	40195.9	272391.0	69385.1
250	114071.5	28891.4	225877.9	57460.7	232411.1	63591.6
500	102449.2	24080.0	235679.4	64040.3	306264.9	83814.0
1000	146999.9	96793.4	325701.5	95408.6	287200.1	75598.3

Table 14 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *E. coli* dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
50	7.6	14	5.3	10
100	10.1	19	8.5	17
150	17.2	35	10.7	21
200	14.4	26	10.9	19
219	16.9	34	16.7	37

Table 15 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
12500	16.4	36	61.0	125
50000	nan	nan	46.5	92
100000	nan	nan	95.8	181
200000	nan	nan	103.8	176
400000	nan	nan	100.2	206

Table 16 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the human chromosome 19 dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
125	9.1	15	17.4	26
250	12.3	20	12.6	26
500	13.8	25	20.2	35
1000	22.1	42	17.9	31