From Relative Compression to Hierarchical Compression

Philip Bille
Department of Applied Mathematics and Computer Science (DTU Compute), Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
Department of Applied Mathematics and Computer Science (DTU Compute), Technical University of Denmark, Lyngby, Denmark

Máximo Pérez-López
Department of Applied Mathematics and Computer Science (DTU Compute), Technical University of Denmark, Lyngby, Denmark

Abstract

We introduce a framework to use any relative compression algorithm as a subroutine for hierarchical relative compression. In a dataset consisting of n sequences, it consists of constructing a rooted tree on the sequences, using hashing and similarity techniques, and compressing the children of a node relative to their parent. We build up on previous techniques [4], and optimize them further for computational efficiency. We test our framework with three existing relative compression algorithms on six genomic datasets, and we show that in datasets that contain heterogeneous data, hierarchical relative compression improves the compression ratio by a factor 2 or more, when compared to relative compression to a single sequence. Apart from compression ratio, we also explore the trade-offs with respect to compression speed, dataset decompression speed, and average sequence decompression speed. With two of the surveyed algorithms, dataset decompression becomes faster and sequence decompression remains practical, at the cost of compression time, which remains competitive for the datasets with highest variability.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Relative compression, RLZ, string collections, compressed representation, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.7

Supplementary Material *Software (Source code)*: `https://gitlab.gbar.dtu.dk/hierarchical-relative-compression/hrcimplementation` [2]

Funding *Philip Bille*: Supported by Danish Research Council grants 10.46540/3105-00302B and 10.46540/4283-00129B.
Inge Li Gørtz: Supported by Danish Research Council grants 10.46540/3105-00302B and 10.46540/4283-00129B.
Máximo Pérez-López: Supported by Danish Research Council grant 10.46540/3105-00302B.

Acknowledgements We thank Simon R. Tarnow for all the help with the existing codebase, and him and Simon J. Puglisi for providing useful datasets. We also thank the anonymous reviewers for their comments and suggestions.

1 Introduction

Given a collection of sequences $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, the relative compression method consists of choosing a sequence $S_R \in \mathcal{S}$ as the *reference*, and parsing every other *target* sequence $S_T \in \mathcal{S} \setminus S_R$ into a sequence of substrings of S_R, called *phrases*. Each phrase can be represented with a tuple (pos, len), that specifies the index in S_R where the substring starts,

and its length. For example, if $S_R = GGAACCGTT$ and $S_T = AACCTTGG$, a relative compression of S_T with respect to S_R can be $(3,4)(8,2)(1,2)$. Decompression is performed by replacing the phrases by the corresponding substrings of S_R. If the strings S_R and S_T are highly similar, we will be able to represent S_T with a small number of phrases, and thus a small compressed size.

The creation of many large genomic databases over the last two decades, made possible by great reductions in the cost of DNA sequencing, has motivated the research and development of better compression methods for this domain of data, where relative compression has found its place. A central relative compression algorithm is relative Lempel-Ziv (RLZ) [23], that greedily selects the longest phrase possible at each time. This and other relative compression algorithms [16, 33, 25, 40, 39] have been used to compress whole collections of highly similar sequences, most notably datasets of full genome sequences of individuals from the same species [23].

When applying reference compression to a set of sequences, the choice of reference is a sensitive variable that can have a big impact on the compression performance [4]. If the sequences can be grouped into distinctive clusters where the sequences are most similar to each other, then the differences between the clusters may pose a challenge for relative compression with respect to a single sequence. This phenomenon appears, for example, in datasets of common bacteria, where the bacteria is known to mutate into distinct strains. To address these issues, the authors of [4] devised a hierarchical modification of Relative Lempel-Ziv that builds a tree hierarchy on the sequences, where the children of a node are compressed with respect to their parent, and the parents are compressed recursively. This method of compression achieved improvements in compression ratio with respect to standard RLZ, most notably in an E-Coli dataset. Given this success, it is natural to propose if this hierarchical structure can be applied to other relative compression algorithms, more tailored to the specific data that we use.

Our contributions. The contributions of this paper are manyfold:

- We present a general framework to transform any relative compression algorithm of choice into a hierarchical one. We build on an existing method [4], and we apply optimizations based on *bottom-k hashing* and *Jaccard similarity* to improve compression runtimes.
- We apply our framework to develop two new hierarchical relative compression algorithms based on the existing relative genomic data compressors HiRGC [25] and iDoComp [33]. We use those, as well as hierarchical RLZ [4], to test the compression of genomic datasets of different kinds. We have three groups of datasets, depending on whether the genomes come from several species, a single species with high variability between the individual genomes, or from a single species with low genome variability.
- We compare the performance of the relative version versus the hierarchical one in terms of compression size, compression time, dataset decompression time and average sequence decompression time. We obtain that in the datasets with many different species, the hierarchical compression achieves around 50% smaller compression. On datasets with single species of high variability, the improvement is of 38%-50%, at the cost of compression time. A dataset of low variability is our worst case, where the hierarchical framework cannot provide improvements. With two of the three surveyed algorithms the decompression speed improves, and the sequence decompression speed remains fast.
- We offer the full, public implementation of all the algorithms used in the paper at `https://gitlab.gbar.dtu.dk/hierarchical-relative-compression/hrcimplementation`.

Techniques. The existing method of hierarchical compression uses locality-sensitive hashing to form clusters of similar sequences. Then, it obtains the relative compression size of each intra-cluster compression, and uses Tarjan's arborescence algorithm [44] to find the best tree hierarchy among the clusters. We further improve the computational efficiency of this method by using bottom-q hashing to compute the string fingerprints, and Jaccard similarity [9, 8] to compute the edge weights instead of true compression size. To compute the relative compression we try 0.5% of the possible references from the dataset and keep the one that provides better compression ratio. Note that previous works had only compared the compression sizes in terms of the *average* of a series of relative compressions, which can be many times larger than the best one found [4].

Related work. The idea of constructing a hierarchy of compressed sequences was proposed in the context of *delta-compression* [36], and was also mentioned in Storer and Szymanski [42]. Deorowicz et al. proposed an RLZ-based method of relative compression that allows a target to be compressed with respect to several references, in a series of papers [17, 16]. Another recent hierarchical compression scenario is the persistent strings model [3]. Both of these methods are quite different to the hierarchical arrangement of relative compressions that was proposed in [4] and that we use here. Tang et al. [43] made a clustering of a dataset for compression, also using several different relative compressors in a similar spirit as this paper, but only with 2 levels of depth. Recently, Břinda et al. [10] also made a hierarchical clustering of genomic datasets based on phylogenetic data, but approached the problem with self-referential compressors like GZip and XZ instead of relative compression.

Other researchers have compressed genomic datasets using precise information about the variants in the sequences [37, 15], for example with available VCF files [14]. This opens the door to substantial compression ratio improvements, at the cost of computing the variants between the sequences of a collection, which can be computationally very expensive [16]. Instead, in this paper we work with the original, plain text FASTA files.

Apart from its compression potential, relative compression has also been a favoured method of compression for its good random-access and pattern-matching properties [19, 20, 30], that have been exploited in other domains such as large text corpora [1, 22, 24, 38, 45, 46].

Outline. In Section 2 we lay down the notation and basic concepts used throughout the paper. In Section 3 we formally define relative compression, and the framework for hierarchical relative compression that we use. In Section 4 we give an overview of the relative compression algorithms that we used for our experiments. In Section 5 we present our experiments and results on six different datasets. Finally, in Section 6 we offer conclusions and closing reflections.

2 Preliminaries

A *string* $S = S[1..n] = S[1]S[2]\dots S[n]$ is a sequence of n symbols from an alphabet Σ of size σ. The genomes we use in this paper are represented in FASTA format [31], which contains a first line starting with a caret ">", followed by a string from the alphabet $\Sigma = \{A, C, G, T\}$, sometimes augmented with extra characters like "N". A *k-mer* of the string is a substring of length k.

The *fingerprint* of a string is a short approximation of the string that we use for efficient comparisons between strings, sometimes also called *sketch*. Let $h : \Sigma^k \to [m]$ be a random hash function that takes as input a k-mer, and produces a number in the range $[m] =$

$\{0, 1, \ldots, m-1\}$, where $m \ll \Sigma^k$. The *min-hash* scheme computes a fingerprint for a string S by hashing every k-mer of S with q different hash functions, and taking the array of the minimum hash values obtained with each hash function as the fingerprint [8, 9].

Let $G = (V, E)$ be a weighted, directed, strongly connected graph. A *spanning arborescence* of G with root r is a subgraph of G that is a directed rooted tree with root vertex r, where all the vertices are reachable from r. The weight of an arborescence is the sum of the weights of its edges. A *minimum weight arborescence* of G is a spanning arborescence of G of minimum weight. A minimum weight arborescence can be computed efficiently with an algorithm of Tarjan [44].

3 Relative and Hierarchical Relative Compression

In this section we describe in detail the concepts of relative compression between two sequences, and for a dataset $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, consisting of n sequences [23]. Then, we describe the hierarchical relative compression of a dataset $\mathcal{S}$ [4]. At the end, we discuss the decompression of the dataset, and of individual sequences.

Relative Compression

In a relative compression between two sequences we choose one as the reference sequence, named R, and the other as the target sequence, named S. The *relative compression of S with respect to R* consists of a compressed representation of S, that refers to data from R. Usually, this means that the compressed representation of S consists of references to substrings of R. An example of such a relative compression is the relative Lempel-Ziv parsing of S with respect to R [23]. The relative compression can be extended to a collection of sequences $\mathcal{S}$ by choosing a reference $S_R \in \mathcal{S}$, and compressing every $S \in \mathcal{S} \setminus \{S_R\}$ with respect to S_R. Thus, the compressed representation of $\mathcal{S}$ is the uncompressed reference S_R, together with the relative-compressed versions of all other $S \in \mathcal{S} \setminus \{S_R\}$.

To choose the reference, in this paper we consider the strategy of choosing a small, constant percentage of the sequences at random, compressing the dataset with respect to each of them, and keeping only the compressed dataset that gives the best compression ratio.

Hierarchical Relative Compression

Let $\mathcal{S}$ be a collection of n sequences and consider a rooted tree T of n vertices, where each vertex represents a different sequence $S \in \mathcal{S}$. The *hierarchical relative compression* of $\mathcal{S}$ according to T consists of the uncompressed sequence S_R at the root of the tree, together with the relative compression of every other $S \in \mathcal{S} \setminus \{S_R\}$ with respect to its parent sequence in T [4]. Note that the relative compression of a collection of sequences corresponds to the hierarchical compression where the tree is a star.

The size of the hierarchical relative compression of a collection of sequences of $\mathcal{S}$ heavily depends on the structure of the tree. To apply hierarchical relative compression to $\mathcal{S}$, we need to find a good tree T. To this end, we follow the method of the *cost graph*, as in [4].

3.1 Cost Graph

We can summarize the cost graph method in the following three points:

1. Build a complete, directed graph G of n vertices, where the vertices are the sequences $S_1, \ldots, S_n$.

2. For each edge (S_i, S_j), compute the size of the relative compression of S_j with respect to S_i, and store it as a weight on the edge.[1]
3. Compute the minimum weight arborescence on the cost graph, where the weight is taken as the sum of the edges plus the size of the uncompressed root sequence. The resulting arborescence is our tree T.

It is easy to see that the algorithm leads to the best possible hierarchical relative compression of $\mathcal{S}$. However, computing the entire cost graph is computationally expensive, because of the $n(n-1)$ required relative compressions. To reduce the computational cost, instead of computing the complete cost graph, we compute a sparse version of it, that contains only the edges with weights that we expect to be part of a good arborescence. This does not provide theoretical guarantees, but previous results show that there exist sparse approximations that only have small compression losses in comparison to the complete graph approach [4]. Additionally, we use the efficient arborescence computation described in [4] that uses Tarjan's algorithm [44], though it does not include the size of the root as part of the computation of the arborescence.

3.2 Sparsification of the Cost Graph

To compute the sparse cost graph, we use the algorithm from [4] with one extra addition. The main idea is to use locality sensitive hashing to cluster together sequences that are similar, and in the cost graph add edges between the sequences of a cluster. When clusters are found, we choose a representative from each cluster, and cluster the representatives again, recursively. It is only when we have found the sparse graph, that we proceed to compute the relative compressions corresponding to the edges. We give now a more detailed overview of the algorithm for completeness.

Let $R \subseteq \mathcal{S}$ be a subset of "representatives" of $\mathcal{S}$. Initially, R is equal to $\mathcal{S}$. We iterate the following steps:

1. Generate fingerprints of each $S \in R$ using the classic min-hashing scheme [8], with q different random hash functions. The parameter q is the size of the fingerprint.
2. Let $C \subseteq R$ be a group of sequences with the same fingerprint. If $|C| \leq \tau$ for some threshold τ, we add the edges (i, j) and (j, i) to G for any pair of sequences $S_i, S_j \in C$.
3. After a round, if the number of connected components did not change, we increase the threshold by a certain factor. Precisely, the ith time that we increase the threshold, we set it to $i^2 \cdot \tau$.
4. Every c-th round, we prune R: we select, from each connected component in G, a single representative. It will be the string S that has had the most collisions with other sequences, where the collisions are the number of strings that had the same fingerprint as S.
5. When $|R| \leq \tau$, we add all edges between members of R, and we stop.

The only change from [4] is point 3. This step helps the algorithm adapt to large clusters of sequences. It also allows it to stop in step 5 with a larger number of representatives, if there were many distinct clusters that could not be joined together. Note that, in the worst case, this last step can potentially add a quadratic number of edges to G, if τ becomes large, and make the following computation of the real distances very heavy.

[1] Note that, in general, the weight of (S_i, S_j) is not the same as the weight of (S_j, S_i).

3.3 Computational Optimizations

We have made further optimizations to the algorithm above. Firstly, we used *bottom-q hashing* to compute the fingerprints in the sparsification algorithm, instead of computing the min-q hash. Secondly, to compute the weights of the edges of the cost graph, we computed a sketch of each sequence and used the Jaccard similarity between the sketches as the weight. We now describe these two approaches.

Bottom-q fingerprinting. Instead of computing q minimum hashes for each sequence, we compute the hash values of all the k-mers with a single hash function, and take the q smallest ones. This technique is known as *bottom-q hashing*, and has been used extensively to improve the running time of min-hashing schemes, by removing a factor q from the complexity of computing the fingerprints [35, 34, 9]. The set of the smallest q hash values is called the *bottom-q sketch*.

Jaccard similarity as weights. For each edge (S_i, S_j) of the complete graph, instead of computing the relative compression of S_j w.r.t. S_i, we compute a bottom-q sketch of both S_i and S_j, named $sk(S_i)$ and $sk(S_j)$, and we let the weight be $1-|sk(S_i)\cap sk(S_j)|/|sk(S_i)\cup sk(S_j)|$. The Jaccard similarity of the sketches is intuitively lighter to compute than the full relative compression between two sequences, and even though the resulting weights will not be as accurate as the actual compression size, Jaccard similarity can provide a reasonable approximation at a lower computational cost [35].

3.4 Decompression

The decompression of a relative-compressed dataset is straightforward: apply the proper decompression algorithm to all sequences S_i with respect to the reference string, which we keep uncompressed. When we apply hierarchical compression, we decompress in levels, following a BFS traversal of the tree. First we decompress the sequences that are compressed with respect to the root, then the ones that are compressed with respect to the children of the root, etc. To decompress a single sequence, first we find the path from the sequence to the root, and then we decompress each sequence on the path, starting from the child of the root until arriving to the desired sequence.

4 Relative Compression Algorithms

We use three existing relative compression algorithms to test our framework. The first two are HiRGC [25] and iDoComp [33], which are referential genome compressors that were designed specifically to target full DNA sequences. The other one is the classic relative Lempel-Ziv compression [23], a general purpose referential compressor. We now give an overview of how these algorithms work.

4.1 HiRGC

HiRGC [25] preprocesses the reference at runtime to extract only the sequence of *A, C, T, G* characters of the FASTA file. Then, it builds a global hash table that maps every k-mer of the sequence to its positions on the reference. The target is parsed left-to-right. At every position, it either finds a k-mer matching the reference, or considers the next characters of the sequence as a mismatch. If it finds a matching k-mer, it exhaustively searches in all its reference occurrences for the longest match it can find. It always considers the next character after a match as a mismatch.

The algorithm stores the matches and mismatches in the order they appear on the target, in plain text. The matches are encoded as pairs (pos, len), where only the differences between the positions of consecutive matches are stored. It has a special encoding for extra characters (outside of *A, C, T, G*) in the target. Finally, we encode everything with an ANS compressor [12]. The original algorithm used PPM [11, 26], but we observed experimentally that ANS provided the same or better compression ratio with faster encoding and decoding speed.

4.2 iDoComp

The iDoComp algorithm [33] first applies a greedy RLZ parsing to all the target, using a precomputed suffix array of the reference. It adds the next mismatched character to every phrase, and represents the ith phrase as a triple $m_i = (pos_i, len_i, ch_i)$. A postprocessing step finds all the cases where two consecutive matches m_i, m_{i+1} can be merged into one by recording only one substitution or insertion. Precisely, in the case where $pos_{i+1} = pos_i + len_i + 1$, it merges the two matches into one and record a substitution $(pos_i + len_i, ch_i)$. Similarly, if $pos_{i+1} = pos_i + len_i$, the character ch_i can be recorded as an insertion at position $pos_i + len_i$. There's an extra case for short matches whose source in the reference is distant from the source of the previous match, and that can be encoded using a number of substitutions in a way that benefits the encoder.

As in HiRGC, the algorithm only stores the differences between consecutive positions, but it does so in absolute value, and records the corresponding signs. All the resulting integers are encoded with 4 bytes, and 4 different byte streams are created for each of them. These streams are encoded independently with ANS. Finally, the mismatched characters and the signs are also encoded with ANS. The only changes to the original algorithm are a simplification of the encoding of the characters and the use of ANS instead of arithmetic encoding. We were unable to reuse the original implementation of the arithmetic encoder.

4.3 Relative Lempel-Ziv

The relative Lempel-Ziv algorithm [23] first builds the suffix array and LCP array of the reference sequence. To parse the target, it greedily chooses the longest possible match at each point in the target sequence, using the suffix and LCP arrays, where a match is a tuple (pos, len), or a single character, if no match was found in the reference. The original algorithm stored the positions as binary integers, and used bitvectors for the lengths to allow for random access queries. Since in this paper we only analyse the raw compression size, to be consistent with the other two methods, we also apply encoding to the output. Given that in the case of RLZ it consists mainly of integers and not characters, we use a Huffman encoder [27, 47, 48] that is tailored to sequences of binary integers.[2]

5 Experiments

In this section we compare hierarchical relative compression against relative compression, in terms of compression size, compression time, decompression time and average sequence decompression time. We first describe the datasets we use, then the precise methods that we applied, the hardware setup, and finally we give an exposition and an analysis of the results.

[2] Note that the ANS encoder by Collet [12] is character-based, not integer-based.

5.1 Data

We evaluate our tools on a series of bacterial and human genome datasets. We group them into three categories, depending on the species diversity and similarity between the sequences:

1. Many species. Datasets with genomes from several species, with high variability between the individual sequences. We choose collections of bacteria:
- NCTC3k: 1065 draft genomes from 259 different bacterial species [7].
- 661k: The original collection contains 661405 draft assemblies of all kinds of bacteria. We took a prefix of 30000 samples, of 98GB of size [5, 6].

2. Single species, high variability. Datasets with genomes from a single species, that have a considerable variability between the individuals. We choose three bacterial datasets:
- E-Coli: 219 assembled bacterial genomes taken from the GenomeTrackr project [41, 28].
- GISP: 1102 draft genomes of N. Gonorrhoeae [21].
- Staph: 62 assembled genomes of Staphylococcus [29].

3. Single species, low variability. Datasets with genomes from a single species, with low variability between the individuals. The classic example is human genomes:
- Chr-19: 1000 assembled, human chromosome 19s from the 1000 Genomes Project [13].

The GISP and NCTC3k datasets were put together by the authors of [10]. The 661k dataset was described in [5]. The precise Chr-19 dataset is available on request. In table 1 we see the different parameters of the datasets. We preprocess the datasets to fit the different levels of assembly of the data (full chromosome assembly vs. draft (contigs) assembly) and to eliminate newline characters; we leave the details in appendix B.

5.2 Methods

We run hierarchical relative compression with custom implementations of HiRGC and iDoComp, together with a well-known implementation of RLZ [23]. Then, we compare it to relative compression, which we executed with the three relative compression algorithms too. With the compressed datasets we decompress $0.5\%n$ random sequences (with a minimum of 5), and record the average time it takes. Finally, we record how long it takes to decompress the entire dataset. In the following, we use the abbreviation *HRC* to mean *hierarchical relative compression*, and *RC* to mean *relative compression*. We now give details about each of the methods.

Hierarchical Relative Compression (HRC). We execute the HRC method of Section 3, using graph sparsification. We use the pair multiply-shift family [18] to draw our random hash functions for fingerprinting, with a k-mer size of 256. We set the fingerprint size between 1 and 4 depending on the dataset (see Table 1). We use bottom-q hashing with a standard red-black tree to keep only the smallest q hash values when computing the fingerprint of a string. The Jaccard similarity is computed with an additional round of bottom-q hashing, where we use a different value of q of 1024 as the size of the sketch for all datasets. To speed up the slowest experiments (with the 661k and Chr-19 datasets), we use parallelism to compute the fingerprints and the weights of each edge, and we record and add up the time spent in each thread to obtain the sequential time. The initial threshold T for the clustering was set at $2 \log n$, where n is the number of sequences in the dataset.

Relative Compression (RC). We test 0.5% of the possible relative compressions with a maximum of 10, and we keep the one with the best compression ratio. We execute them in parallel, and add up the time spent in each thread to compute the sequential time.

HiRGC. We re-implement HiRGC to use a hash table size that is proportional to the size of the reference string, instead of the fixed 2^{28} size of the original implementation. This is important in terms of compression time (and memory usage) for the smaller sequences. We use a 2-independent random hash function with the hash table, and linear probing to solve the collisions.

iDoComp. We re-implement iDoComp to solve the crashes and data corruption that the original implementation suffered from (this was already mentioned in [25, Supplementary material]). The parsing and post-processing are identical, but we simplify the character stream to contain all mismatched characters as they appear on the target. We keep an independent encoding for each of the byte streams, as described in the original paper, only with ANS instead of arithmetic encoding.

To run relative compression with iDoComp, we pre-compute the suffix array of all sequences in the dataset, and we store them alongside the sequences. We do not consider the initial FASTA definition line of the sequence when computing the suffix array.

RLZ. The implementation of RLZ follows the $O(m + \log n)$ standard algorithm to find a match of a pattern of length m with the suffix array and LCP array of the reference string. Everything else is as in the standard algorithm. We use the same suffix arrays that are needed for iDoComp, and compute the LCP array in memory after reading the reference and the suffix array. Note that this is only needed for compression, not decompression.

5.3 Hardware setup

We run all the experiments on a desktop PC running Kubuntu version 24.04, kernel version 6.8.0-90-generic, 64 bits. The code was built with CMake version 3.28.3, with the Release profile. The compiler was `g++` version 13.3.0, targeting C++20, with optimization flags `-O3 -funroll-loops -DNDEBUG -fopenmp`. The version of OpenMP is 4.5. The CPU is a 4.3GHz AMD Ryzen 9 9950X, with 64GB of DDR5 main memory. The CPU has 16 cores with 2 threads per core, for a total of 32 possible parallel threads. It has separate instruction and data L1 caches, of 768 KiB and 512KiB respectively, for each of the 16 cores. It has L2 caches of 16 MiB for each core, and two shared L3 caches of 64 MiB each.

5.4 Experimental results

In this section we compare hierarchical relative compression with relative compression in terms of compression size, compression time, and decompression time, across the three different relative compression methods and the three dataset groups.

Many species. The results of the experiments are shown in Figure 1. We observe that hierarchical relative compression produces a compressed version of NCTC3k that is around 50% smaller than the best of the attempted relative compressions, with any of the three algorithms; and around 56% smaller for 661k. The time it takes to compress with HRC is slightly longer than the equivalent RC when using HiRGC in both datasets, but faster when using iDoComp and RLZ. We remark that the times of relative compression are an aggregate

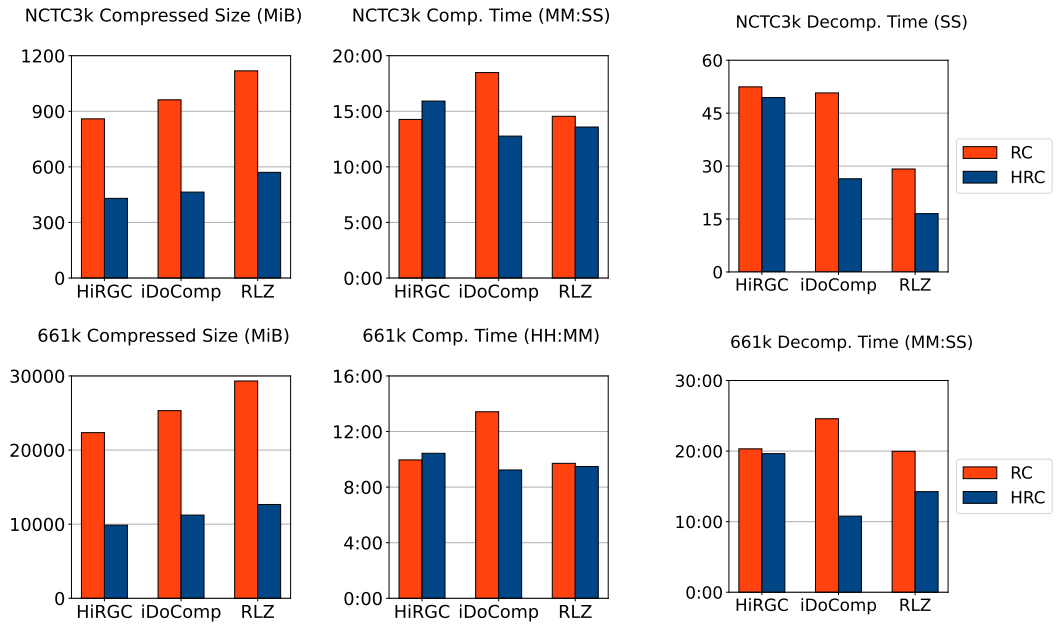

Figure 1 Compression sizes, and compression and decompression times for the datasets with many species.

of $0.5\%n$ attempts with different random references, that have been capped to 10 for the 661k dataset (in this case, 0.5% of the sequences would amount to the excessive number of 150 compressions).

The comparison of the decompression times is favourable for HRC, specially with iDoComp and RLZ, that are more than 40% faster on NCTC3k. On the 661k dataset, iDoComp runs more than 50% faster, and RLZ runs 29% faster.

In terms of sequence decompression, the comparison to relative compression is poor with all the different algorithms. Table 5 (appendix A) shows the exact runtimes. However, this is to be expected, since in HRC we may have to decompress many intermediate sequences (up to $n-2$) before decompressing the one we aim for, while in relative compression we can directly decompress the chosen sequence. Nonetheless, we wish to highlight that the absolute time required on average is fast: for NCTC3k it takes 128ms with iDoComp and 77ms with RLZ, while in 661k we need 1.56 seconds with iDoComp and 822ms with RLZ.

Single species, high variability. Figure 2 contains the results of the experiments. In these datasets, HRC achieves a 38% improvement in compression ratio for E-Coli and GISP, and around a 50% for Staph. This comes at a cost in compression time, that is around 3-5 times slower in E-Coli, 1.5-2.3 times slower in GISP, and around 10 times slower in Staph. We note that the Staph times are still fast, under 40 seconds, for all compressors.

In terms of the decompression time, we see an asymmetry between the underlying algorithms, with HiRGC slightly slower with HRC than with RC, but iDoComp and RLZ being faster with HRC, between 25% and 40% depending on the dataset. With regards to average sequence decompression, the picture is the same as in group 1, with HRC being several times slower than RC, but still achieving times that are fast. This time, the average with RLZ is less than 80ms in any of the three datasets, while iDoComp ranges from 28ms to 129ms, and HiRGC ranges from 121ms to 377s.

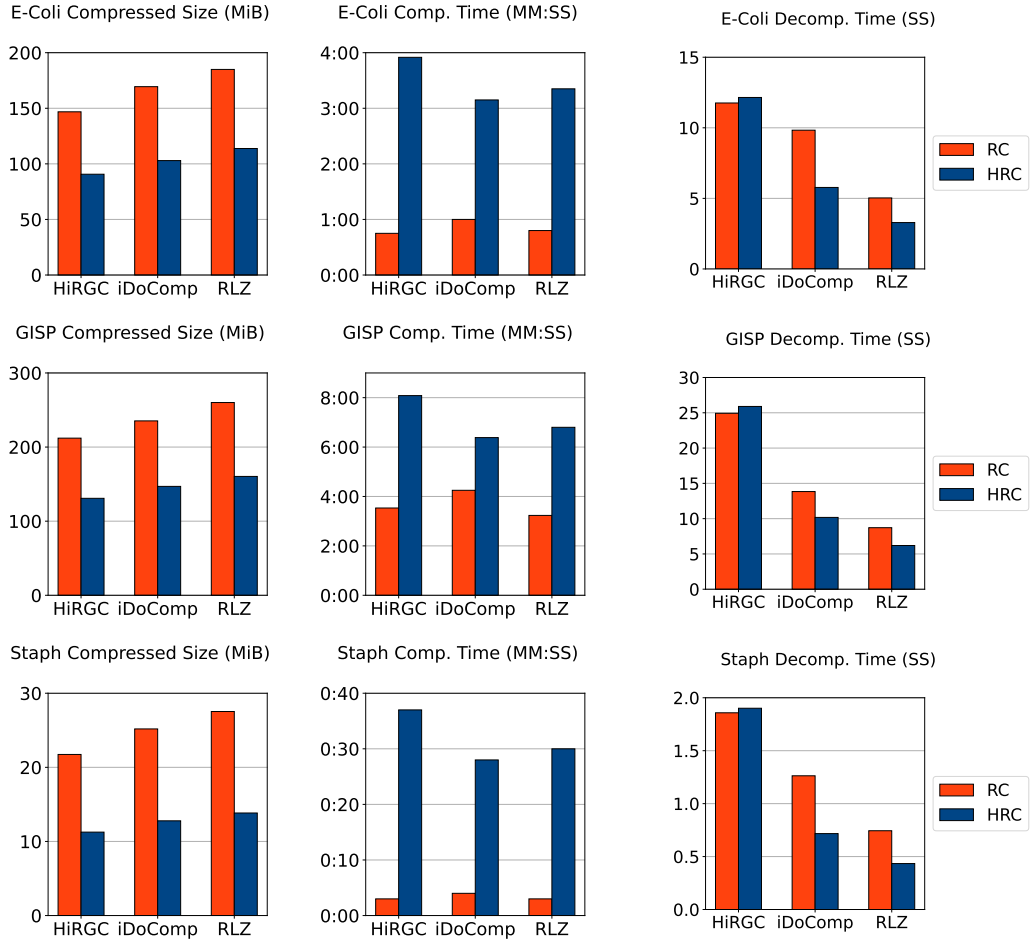

Figure 2 Compression sizes, and compression and decompression times for the datasets with a single species and high variability between the sequences.

Single species, low variability. This group is the worst case for HRC. We still achieve better compression rates than RC with all three relative compression algorithms, but the improvements are small this time, of around 3%. The compression time is an order of magnitude slower with all three algorithms. The decompression times are also slower with all algorithms. The average sequence decompression time, as before, compares poorly to RC, but is still fast with iDoComp (949ms) and RLZ (563ms).

5.4.1 Analysis

Compression Size. The resulting compression sizes can be explained by the dissimilarity between sequences in the datasets: the 661k and NCTC3k datasets contain DNA from several different species, and this can pose a challenge for relative compression, that can only choose a single sequence from one species as the reference. Thus, the relative compression performance may suffer when compressing sequences from a different species than the reference, with a possibly highly dissimilar DNA. Hierarchical relative compression addresses this issue, by clustering together sequences that have a similar DNA, achieving a good compression ratio on them, and then having to compress only representatives from each cluster further up in the

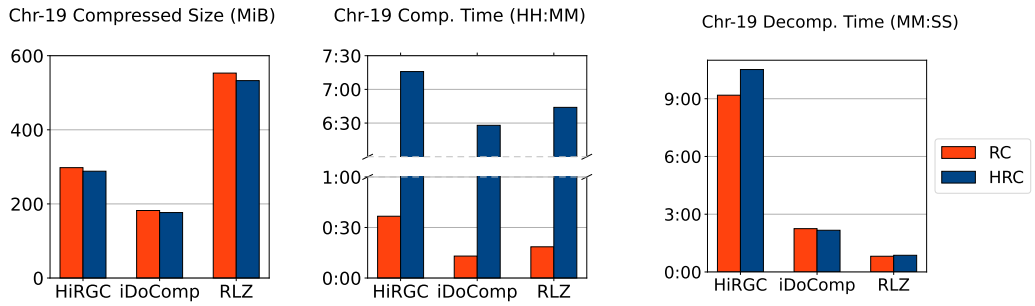

Figure 3 Compression size, and compression and decompression time for the dataset with a single species, and low variability between the sequences (Chr-19).

compression tree. In addition, thanks to the depth of the compression tree, we can recursively exploit the similarities between the representatives, that further improves the compression performance. This takes advantage of the similarities between subsets of the sequences in the dataset, while relative compression can only take advantage of similarities between the individual targets and a single reference. By the same argument, human DNA, which is highly similar between individuals, can be effectively compressed with relative compression; hence the difference between the two methods shows up to be minimal.

Compression Time. There is an overarching pattern that relates the results over the three kinds of datasets, namely that improvements in the compression ratio of HRC with respect to RC are correlated with improvements in compression time with respect to RC. Indeed, the datasets with many species, which present the best improvements in compression ratio, also show the best comparison in terms of compression time, specially with iDoComp and RLZ. The datasets of single species with high dissimilarity, which benefit from improved compression ratios, albeit not as substantial as the many-species datasets, also present compression time comparisons that are in between the other two categories.

To explain this phenomenon, we focus on how the parsing is done. Most relative compression algorithms, including the three that we survey in this paper, rely on greedy scans of the target and the reference to find the longest matches possible. These scans are fast and cache-efficient on modern hardware. The longer the matches are (leading to smaller compression size), the fewer scans we need to cover the target, and the faster the compression algorithm runs. Furthermore, it creates a smaller output that will be processed faster by the subsequent encoding algorithm.

Another factor is at play in the compression time exhibited by hierarchical relative compression, and that is the length of the sequences. We see that in the Chr-19 dataset, where the sequences are the longest, the bottleneck for the compression time is the computation of the fingerprints for the clustering algorithm. The other datasets, which consist of bacteria, that have relatively smaller genomes, do not experience such a heavy slowdown.

Decompression Time. The same explanation as above can be used for the decompression algorithms, that usually do not do much more than copying sections of the reference to the output, which is also fast and cache-efficient. The longer the matches are, the fewer jumps in the reference we need to do, which results in less cache-misses and faster runtime. In the case of HiRGC though, we do not observe substantial speed-ups. We believe that this is due to the algorithm having to process the reference when decompressing to remove extra

Table 1 Parameters of the datasets.

	Many species		Single species, high variability			Single species, low variability
Parameters	NCTC3k	661k	E-Coli	GISP	Staph	Chr-19
Size	4.1 GB	98 GB	1.1 GiB	2.3 GB	166 MB	56 GiB
Nº of sequences	1067	30000	219	1104	62	1000
Avg sequence length	3.82 MB	3.27 MB	5 MB	2.08 MB	2.72 MB	59MB
Assembly	Draft	Draft	1 Chr.	Draft	1 Chr.	1 Chr.
Fingerprint size	1	1	4	4	4	4

characters, whereas iDoComp and RLZ require no such processing, just the raw reference. This extra processing needs to be applied to each sequence that is an internal node of the hierarchy, and it slows down the decompression in HRC, while RC only needs to process one reference (the root).

Sequence Decompression Time. In terms of sequence decompression, we have already mentioned that it performs many times slower with HRC than with RC; however this is to be expected, given the many decompressions that are needed to extract an arbitrary sequence from the compressed dataset. We remain satisfied with the fact that, when using iDoComp and RLZ, the average sequence decompression times are fast and practical. Still, we can observe that the computation time grows as the number of sequences in the dataset, and potentially the hierarchy depth, grows. It will require a new algorithmic insight to design an algorithm that can decompress arbitrary sequences in HRC-compressed datasets with a reduced dependency on the path length to the root.

6 Conclusion

We have presented a framework to turn any relative compression algorithm into a hierarchical one. We have shown that in two datasets with several species, hierarchical compression improves the compression size over relative compression around 50%, and the compression time is competitive against the strategy of testing 0.5% of the possible relative compressions. The method also improves the compression on datasets of single species with high variability from 38% to 50%, at the cost of compression time. In both settings, when the underlying relative decompression algorithm requires no computations on the reference, the decompression time becomes faster, and the average sequence decompression time remains practical.

References

1 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. doi:10.1007/s00453-017-0380-7.

2 Philip Bille, Inge Li Gørtz, and Máximo Pérez-López. Hierarchical Relative Compression reference implementation. Software, version 1.0., Supported by Danish Research Council grant 10.46540/3105-00302B (visited on 2026-05-29). URL: https://gitlab.gbar.dtu.dk/hierarchical-relative-compression/hrcimplementation, doi:10.4230/artifacts.26213.

3 Philip Bille and Inge Li Gørtz. Random Access in Persistent Strings. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2020.48.

4 Philip Bille, Inge Li Gørtz, Simon J. Puglisi, and Simon R. Tarnow. Hierarchical Relative Lempel-Ziv Compression. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms (SEA 2023)*, volume 265 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2023.18.

5 Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):e3001421, 2021. doi:10.1371/journal.pbio.3001421.

6 Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Index of /pub/databases/ENA2018-bacteria-661k/Assemblies, November 2021. URL: https://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k/Assemblies/.

7 Karel Brinda. NCTC 3000 complete assemblies, 2021. doi:10.5281/zenodo.4838517.

8 Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-Wise Independent Permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000. doi:10.1006/jcss.1999.1690.

9 A.Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997. doi:10.1109/SEQUEN.1997.666900.

10 Karel Břinda, Leandro Lima, Simone Pignotti, Natalia Quinones-Olvera, Kamil Salikhov, Rayan Chikhi, Gregory Kucherov, Zamin Iqbal, and Michael Baym. Efficient and robust search of microbial genomes via phylogenetic compression. *Nature Methods*, 22(4):692–697, 2025. doi:10.1038/s41592-025-02625-2.

11 J. Cleary and I. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984. doi:10.1109/TCOM.1984.1096090.

12 Yann Collet. Cyan4973/FiniteStateEntropy, 2026. original-date: 2013-12-15T14:05:21Z. URL: https://github.com/Cyan4973/FiniteStateEntropy.

13 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015. doi:10.1038/nature15393.

14 Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, and Richard Durbin. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011. doi:10.1093/bioinformatics/btr330.

15 Sebastian Deorowicz, Agnieszka Danek, and Szymon Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013. doi:10.1093/bioinformatics/btt460.

16 Sebastian Deorowicz, Agnieszka Danek, and Marcin Niemiec. GDC 2: Compression of large collections of genomes. *Scientific Reports*, 5(1):11565, 2015. doi:10.1038/srep11565.

17 Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011. doi:10.1093/bioinformatics/btr505.

18 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

19 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel–Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014. doi:10.1016/j.tcs.2013.07.024.

20 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A Faster Grammar-Based Self-index. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 240–251, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-28332-1_21.

21 Yonatan Grad. Data for "Genomic Epidemiology of Gonococcal Resistance to Extended-Spectrum Cephalosporins, Macrolides, and Fluoroquinolones in the United States, 2000-2013", 2019. doi:10.5281/zenodo.2618836.

22 Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endow.*, 5(3):265–273, 2011. doi:10.14778/2078331.2078341.

23 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval. *String Processing and Information Retrieval*, 6393:201–206, 2010. doi:10.1007/978-3-642-16321-0_20.

24 Kewen Liao, Matthias Petri, Alistair Moffat, and Anthony Wirth. Effective Construction of Relative Lempel-Ziv Dictionaries. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 807–816, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee. doi:10.1145/2872427.2883042.

25 Yuansheng Liu, Hui Peng, Limsoon Wong, and Jinyan Li. High-speed and high-ratio referential genome compression. *Bioinformatics*, 33(21):3364–3372, 2017. doi:10.1093/bioinformatics/btx412.

26 A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990. doi:10.1109/26.61469.

27 A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997. doi:10.1109/26.634683.

28 Tommi Mäklin, Teemu Kallonen, Jarno Alanko, Veli Mäklinen, Jukka Corander, and Antti Honkela. mGEMS Escherichia coli reference dataset, 2020. doi:10.5281/zenodo.3724112.

29 Tommi Mäklin, Teemu Kallonen, Jarno Alanko, Veli Mäklinen, Jukka Corander, and Antti Honkela. mGEMS Staphylococcus aureus reference dataset, 2020. doi:10.5281/zenodo.3724135.

30 Gonzalo Navarro and Víctor Sepúlveda. Practical Indexing of Repetitive Collections Using Relative Lempel-Ziv. In *2019 Data Compression Conference (DCC)*, pages 201–210, 2019. ISSN: 2375-0359. doi:10.1109/DCC.2019.00028.

31 NCBI. FASTA Format for Nucleotide Sequences. URL: https://www.ncbi.nlm.nih.gov/genbank/fastaformat/.

32 NCBI. The NCBI Genome assembly data model. Section: data-processing. URL: https://www.ncbi.nlm.nih.gov/datasets/docs/v2/data-processing/policies-annotation/data-model/.

33 Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. iDoComp: a compression scheme for assembled genomes. *Bioinformatics*, 31(5):626–633, 2015. doi:10.1093/bioinformatics/btu698.

34 Brian D. Ondov, Gabriel J. Starrett, Anna Sappington, Aleksandra Kostic, Sergey Koren, Christopher B. Buck, and Adam M. Phillippy. Mash Screen: high-throughput sequence containment estimation for genome discovery. *Genome Biology*, 20(1):232, 2019. doi:10.1186/s13059-019-1841-x.

35 Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, 2016. doi:10.1186/s13059-016-0997-x.

36 Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *Proceedings of the Third International Conference on Web Information Systems Engineering, 2002. WISE 2002.*, pages 257–266, 2002. doi:10.1109/WISE.2002.1181662.

37 Dmitri S. Pavlichin, Tsachy Weissman, and Golan Yona. The human genome contracts again. *Bioinformatics*, 29(17):2199–2202, 2013. doi:10.1093/bioinformatics/btt362.

38 Matthias Petri, Alistair Moffat, P. C. Nagesh, and Anthony Wirth. Access Time Tradeoffs in Archive Compression. In Guido Zuccon, Shlomo Geva, Hideo Joho, Falk Scholer, Aixin Sun, and Peng Zhang, editors, *Information Retrieval Technology*, pages 15–28, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-28940-3_2.

39 Subrata Saha and Sanguthevar Rajasekaran. ERGC: an efficient referential genome compression algorithm. *Bioinformatics*, 31(21):3468–3475, 2015. doi:10.1093/bioinformatics/btv399.

40 Wei Shi, Jianhua Chen, Mao Luo, and Min Chen. High efficiency referential genome compression algorithm. *Bioinformatics*, 35(12):2058–2065, 2019. doi:10.1093/bioinformatics/bty934.

41 Eric L. Stevens, Ruth Timme, Eric W. Brown, Marc W. Allard, Errol Strain, Kelly Bunning, and Steven Musser. The Public Health Impact of a Publically Available, Environmental Database of Microbial Genomes. *Frontiers in Microbiology*, 8, 2017. doi:10.3389/fmicb.2017.00808.

42 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982. doi:10.1145/322344.322346.

43 Tao Tang, Yuansheng Liu, Buzhong Zhang, Benyue Su, and Jinyan Li. Sketch distance-based clustering of chromosomes for large genome database compression. *BMC Genomics*, 20(10):978, 2019. doi:10.1186/s12864-019-6310-0.

44 R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977. doi:10.1002/net.3230070103.

45 Jiancong Tong, Anthony Wirth, and Justin Zobel. Compact Auxiliary Dictionaries for Incremental Compression of Large Repositories. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1629–1638, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2661829.2661961.

46 Jiancong Tong, Anthony Wirth, and Justin Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, SIGIR '14, pages 283–292, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2600428.2609576.

47 A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622–628, 2000. doi:10.1109/26.843129.

48 Andrew Turpin. turpinandrew/shuff, 2024. original-date: 2017-04-03T01:08:56Z. URL: https://github.com/turpinandrew/shuff.

A Additional tables

Tables 2, 3 and 4 contain the exact values of the compression sizes, compression times and decompression times of each of the compression methods tested with all the datasets.

Table 2 Compression sizes in MiB.

	Many species		Single species, dissimilar			Single species, similar
	NCTC3k	661k	E-Coli	GISP	Staph.	Chr-19
Uncompressed	4149.609	97214.224	1078.035	2257.09	165.291	56386.246
HiRGC RC	859.144	22,350.982	146.819	212.050	21.739	297.850
HiRGC HRC	429.983	9,870.341	90.675	130.848	11.259	288.228
iDoComp RC	169.445	25,320.466	304.300	235.321	25.183	182.213
iDoComp HRC	463.786	11,225.914	102.936	146.949	12.783	176.690
RLZ RC	1,118.198	29,320.629	184.946	260.097	27.534	553.057
RLZ HRC	570.361	12,657.886	113.815	160.390	13.840	532.746

Table 3 Compression times (HH:MM:SS).

	Many species		Single species, dissimilar			Single species, similar
	NCTC3k	661k	E-Coli	GISP	Staph.	Chr-19
HiRGC RC	00:14:16	09:57:27	00:00:45	00:03:32	00:00:03	00:36:41
HiRGC HRC	00:15:55	10:26:04	00:03:55	00:08:05	00:00:37	07:15:52
iDoComp RC	00:18:29	13:25:18	00:01:00	00:04:15	00:00:04	00:13:02
iDoComp HRC	00:12:46	09:14:04	00:03:09	00:06:23	00:00:28	06:28:00
RLZ RC	00:14:33	09:42:41	00:00:48	00:03:14	00:00:03	00:18:32
RLZ HRC	00:13:35	09:29:12	00:03:21	00:06:48	00:00:30	06:43:59

Table 4 Decompression times (MM:SS).

	Many species		Single species, dissimilar			Single species, similar
	NCTC3k	661k	E-Coli	GISP	Staph.	Chr-19
HiRGC RC	00:52.454	20:18.605	00:11.758	00:24.928	00:01.858	09:10.538
HiRGC HRC	00:49.413	19:38.480	00:12.149	00:25.898	00:01.901	10:31.625
iDoComp RC	00:50.729	24:34.847	00:09.831	00:13.840	00:01.263	02:14.716
iDoComp HRC	00:26.409	10:47.079	00:05.775	00:10.177	00:00.717	02:10.452
RLZ RC	00:29.178	19:57.554	00:05.032	00:08.719	00:00.744	00:49.031
RLZ HRC	00:16.525	14:15.167	00:03.287	00:06.196	00:00.434	00:52.668

B Preprocessing of the data

In terms of assembly level, it is important to remark that many DNA datasets do not have their samples assembled to one contiguous sequence. This means that the actual input can be files containing many pieces of DNA data, that could be put together, or *assembled*, to produce the full original sequence. Depending on which level of assembly has been performed, from smallest to largest length we can classify the data as *reads*, *contigs, scaffolds, chromosomes* or *whole genomes*. Sometimes, the assemblies at the level of contigs or scaffolds are called *draft assemblies*. Many species of bacteria and viruses only have one chromosome in their genome, and thus their one assembled chromosome is the whole genome. For humans, the whole genome normally consists of 46 chromosomes, plus other genetic data (see the NCBI genome assembly data model for more details [32]).

The assembly level affects our setup, insofar as the FASTA file of one individual may contain a definition line per each contig or scaffold, and these breaks can affect compression performance. Furthermore, some compression methods are only computationally efficient up to a certain level of assembly or sequence length. For example, HiRGC was designed to process sequences no longer than the length of a single human chromosome. To get around these issues, we preprocess our contig datasets to extract all the FASTA definition lines except for the first one, and concatenate all the DNA data of all the contigs together into one line. The length of this line does not exceed the expected length for a human chromosome in any of the contig datasets. We store the original definition lines in an auxiliary file, together with the position in the preprocessed file where they occurred originally. We do not count the size of this auxiliary data in our experiments, to focus on the compressibility of the DNA sequences. Finally, we use the character N to signify that a base pair is missing. In those

Table 5 Average sequence decompression time in milliseconds (ms), between a minimum of 5 and $n \cdot 0.5\%$ for all the datasets.

	Many species		Single species, dissimilar			Single species, similar
Avg. seq. dec. (ms)	NCTC3k	661k	E-Coli	GISP	Staph.	Chr-19
HiRGC RC	41.970	68.827	38.766	18.342	22.403	407.160
HiRGC HRC	472.244	3,730.250	376.630	121.476	128.463	5,883.910
iDoComp RC	25.579	58.221	18.148	5.697	9.683	186.187
iDoComp HRC	128.095	1,565.440	128.095	28.243	36.491	948.977
RLZ RC	17.222	30.690	11.525	4.193	6.076	124.084
RLZ HRC	76.945	822.050	76.945	17.438	23.410	562.515

datasets where the character "-" is used instead, we replace them by Ns. Any other details can be found at the GitLab repository: `https://gitlab.gbar.dtu.dk/hierarchical-relative-compression/hrcimplementation`.