RENANO: a REference-based compressor for NANOpore FASTQ files

Guillermo Dufort y Álvarez[1], Gadiel Seroussi[1], Pablo Smircich[2,3], José Sotelo[2,3], Idoia Ochoa[4,5], and Álvaro Martín[1]

[1] *Facultad de Ingeniería, Universidad de la República, Uruguay*
[2] *Facultad de Ciencias, Universidad de la República, Uruguay*
[3] *Departamento de Genómica, Instituto de Investigaciones Biológicas Clemente Estable, Uruguay*
[4] *TECNUN School of Engineering, University of Navarra, Spain*
[5] *Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA*

Abstract

Nanopore sequencing technologies are rapidly gaining popularity, in part, due to the massive amounts of genomic data they produce in short periods of time (up to 8.5 TB of data in less than 72 hs). In order to reduce the costs of transmission and storage, efficient compression methods for this type of data are needed. Unlike short-read technologies, nanopore sequencing generates long noisy reads of variable length. In this note we introduce RENANO, a reference-based lossless FASTQ data compressor, specifically tailored to compress FASTQ files generated with nanopore sequencing technologies. RENANO builds on the recent compressor ENANO, which is currently state of the art. It focuses on improving the compression of the base call sequence portion of the FASTQ file, leaving the other parts of ENANO intact. Two novel reference-based compression algorithms are introduced, contemplating different scenarios: in the first scenario, a reference genome is available without cost to both the compressor and the decompressor; in the second, the reference genome is available *only* on the compressor side, and a compacted version of the reference is transmitted to the decompressor as part of the compressed file. To evaluate the proposed algorithms, we compare RENANO against ENANO on several publicly available nanopore datasets. In the first scenario considered, RENANO improves the base call sequences compression of ENANO by 40.8%, on average, over all the datasets. As for total compression (including the other parts of the FASTQ

file), the average improvement is 13.1%. In the second scenario considered, the base call compression improvements of RENANO over ENANO range from 15.2% to 49.0%, depending on the coverage of the compressed dataset, while in terms of total size, the improvements range from 5.1% to 16.5%.

Contents

1 Background **2**

2 Methods **5**
- 2.1 Notations and definitions 6
- 2.2 RENANO: A reference-dependent compression and decompression scheme 9
  - 2.2.1 The encoding algorithm 9
  - 2.2.2 The decoding algorithm 16
- 2.3 $\text{RENANO}_{\text{IND}}$: a reference-dependent compression scheme, with a reference-independent decompression scheme 19
- 2.4 Alignment Information 22

3 Experimental results **27**
- 3.1 Datasets 28
- 3.2 PAF files generation 29
- 3.3 Algorithm parameters 29
- 3.4 Comparative experimental results 30
  - 3.4.1 Comparison against ENANO 30
  - 3.4.2 Comparison against Genozip 33

1 Background

We investigate compression of DNA data generated by *Nanopore sequencing* (NPS) technologies [25] and stored in the FASTQ format [3]. A FASTQ file stores the result of a sequencing process, which consists of a set of readings of genome fragments, called *reads*. Each read contains an *identifier string*, a *base call sequence* (also referred to as read), and a *quality score sequence*. The identifier string is generally a short free text segment, which identifies the read. The base call sequence is a string of letters from the set $\{A, C, G, T, N\}$, where letters A, C, G, and T, represent the nitrogenous bases (base-pairs) of a DNA sequence, and letter N is a special character that represents a nitrogenous base that could not be determined. For convenience, we refer to the base call sequence as a *base call string*, as it will often be used as a string throughout the rest of this document. Finally, the quality score sequence is a string of symbols, of

the same length as the base call string, where the i-th symbol encodes an estimated probability of the i-th base call being correct.

The characteristics of the base call string and the quality score sequence that compose each read vary depending on the technology used to perform the DNA sequencing. For example, Next Generation Sequencing (NGS) technologies generate high quality short reads (a few hundred base-pair long), generally of fixed length, with a quality score alphabet that ranges from 4 values to about 40, depending on the specific technology. On the other hand, NPS technologies generate very long reads (up to hundreds of thousands base-pair long), of variable length, which generally have lower quality than those generated by NGS, and a larger quality scores alphabet size (94 for NPS).

In previous work we developed ENANO [5], a lossless compression algorithm especially designed for NPS FASTQ files. ENANO achieves state of the art compression performance by using a novel quality score sequence compression algorithm that exploits the specific characteristics of NPS data. In this work we focus on further improving the compression of NPS FASTQ files by addressing, specifically, the compression of the base call strings.

Many algorithms have been proposed in the literature to compress the base call strings of a FASTQ file [7]. These compression algorithms can be roughly divided into two categories: *reference-free* methods, in which no external information is used to aid compression, and *reference-based* methods, in which an external reference genome is used to aid compression. A reference genome file stores the base call strings that compose the chromosomes of a genome, usually in FASTA format [19], which is a variation of the FASTQ format that does not store quality scores.

Reference-free compression methods have the advantage of being self-contained and not needing any external resource to work properly, but usually lack in compression performance compared to reference-based methods. On the other hand, reference-based methods require access to a proper reference genome file, that is, a genome of the same species as the one sequenced and stored in the FASTQ file. In turn, these methods achieve better compression performance (see, e.g., [12], [8], [2]) by exploiting the similarities across genomes of the same species, which, for example, in the case of human genomes exceeds 99% of the base-pairs [16].

To exploit the information provided by an external reference genome, reference-based compression algorithms start by *aligning* the base call strings of the FASTQ file against the strings in the reference genome file, producing a series of *alignments*. Loosely speaking, an alignment is a description of a string q in terms of a reference string r, which describes the editing operations needed on the reference string r to produce the string q (these notions, as applied in this work, will be defined more precisely in Section 2.1). If the aligned string q is similar to the reference string r, then the alignment serves as a compact representation of q. Consequently, reference-based

compression algorithms efficiently compress q by encoding the alignment instead. The decoder can later reconstruct the string q by decoding the alignment and applying the editing operations described in it to the reference string r, which is assumed available on the decoder side.

Although the availability of a proper reference genome may seem a demanding requirement at first sight, most bioinformatic analysis tasks performed on FASTQ data, such as sequence analysis or gene expression analysis, already require a step in their pipeline where the reads of the FASTQ file are aligned against a reference genome [17], using specialized alignment tools. In the case of the long noisy reads produced by NPS technologies, the state of the art alignment tool is Minimap2 [18], which outputs the result of aligning each read of a FASTQ file against a reference genome, in PAF format [18]. In this sense, having a reference genome file and corresponding alignment information of the FASTQ file of interest at our disposal is not an uncommon scenario in bioinformatic tasks. In this work we focus on exploiting this scenario by taking a reference-based approach to compressing the read base call strings of the FASTQ file.

As mentioned, the decoding process requires access to the same reference genome file used during compression, which may not always be practical. To address this problem, reference-based compression methods can encode the relevant parts of the reference genome file into the compressed file, thus sacrificing some compression performance in exchange for making the decompression algorithm independent of the original reference genome. We will use this approach in one variant of the algorithms proposed in this work.

Most of the reference-based compression algorithms for base call strings available in the literature [14, 8, 2, 12, 13, 1, 6, 9] are specifically tailored to compress high quality short reads generated by NGS technologies, and therefore do not perform well (or at all) on the noisy long reads generated by NPS technologies. The more recent work [15] presents a compression tool capable of compressing NPS FASTQ files, which offers both a reference-free mode and a reference-based mode. However, our experiments have shown that, when running both methods on NPS FASTQ data, the reference-free mode consistently outperforms the reference-based mode, and the reference-based mode fails to compress some of the tested datasets (results of these experiments are presented in Section 3.4.2).

In this note we introduce RENANO, a lossless NPS FASTQ data compressor that builds on its predecessor ENANO, introducing two novel reference-based compression algorithms for base call strings that significantly improve the state of the art compression performance.

The ENANO compressor [5] parses the input FASTQ file into blocks, whose boundaries are dynamically determined so that every read is entirely contained in a single block. Each block is divided into three streams: read identifiers, base call

strings, and quality score sequences. For each stream ENANO uses a specific *context model* [23] to determine a probability distribution for each symbol, which, in turn, is passed to an arithmetic encoder [24] to generate the output bit stream. These probability distributions are dynamically updated as the file is sequentially compressed. The decompressor works in lockstep, symmetrically. The input file is encoded while the statistics are adaptively updated for each context model, for a certain number of blocks of the file. Afterwards, the statistics are fixed, allowing for fast parallel compression and decompression. Full details are given in [5]. We build RENANO by replacing the base call strings compression scheme of ENANO with one of two novel reference-based compression algorithms introduced in Sections 2, while keeping the processing of the other streams intact. The two variants of RENANO thus constructed are complete FASTQ file compression schemes. However, since the rest of the paper deals only with the compression of base call strings, we will slightly abuse terminology and use the same name RENANO to refer to the compression algorithm for base call strings described in Section 2.2, and to the variant $\text{RENANO}_{\text{IND}}$ described in Section 2.3, which does not require a reference genome on the decompression side.

The rest of the note is organized as follows. Section 2 introduces basic notation, terminology, and tools and, as mentioned, describes the two proposed reference-based base call string compression algorithms. Section 3 presents experimental results, and comparisons of RENANO to ENANO and to the scheme Genozip of [15]. The experiments reveal that both variants of RENANO significantly outperform the schemes they are compared against.

2 Methods

In this section we present RENANO and $\text{RENANO}_{\text{IND}}$, two novel long-read base call string compression algorithms that use the alignment information (against a reference sequence) to improve compression. The general idea behind the proposed compression schemes is to encode a large portion of each base call string as a series of alignments to a reference genome, such that the alignments can be described more compactly than the raw base call string, thus yielding better compression. For both schemes, we assume that a reference sequence (e.g., a genome) in FASTA format is available on the compressor side. For RENANO, we consider the scenario in which the reference sequence is also available without cost to the decompressor, while for $\text{RENANO}_{\text{IND}}$ we consider the scenario in which the reference sequence is *not* available to the decompressor, but a compacted version of that sequence is stored as part of the compressed output, thus incurring a code length cost. In both cases, we assume that the alignment information is obtained from an available alignment tool, such as Minimap2 [18], and that is represented in the PAF format [18]. Finally,

both algorithms assume the base call strings are stored in the widely used FASTQ format [3]. We show in Section 3 that both compression schemes exhibit improvements in compression ratio as compared to ENANO [5], which does not use a reference sequence or alignment information, and to Genozip [15], which offers two compression modes: a reference-based mode, and a reference-free mode.

Both of the presented compression algorithms take the alignment information generated by the alignment tool and transform it to an internal representation, which, together with the reference sequence, allows for perfect reconstruction of the original base call strings. The ability to encode the internal representation of alignments compactly is at the core of the schemes. In Section 2.1 we present the general notations and definitions needed to formalize the proposed internal alignment representation. Sections 2.2 and 2.3 describe RENANO and $\text{RENANO}_{\text{IND}}$, respectively, in detail. Finally, Section 2.4 provides specific details on how the output of a popular alignment tool, as Minimap2 [18], can easily be transformed into our internal representation.

Implementations of full FASTQ file compressors/decompressors based on RENANO and $\text{RENANO}_{\text{IND}}$ are freely available for download at `https://github.com/guilledufort/RENANO`.

2.1 Notations and definitions

Let $\Sigma = \{A, C, G, T, N\}$ be the alphabet of base calls, where $\{A, C, G, T\}$ represent the DNA nucleotides, and N represents an undetermined base call. For a symbol $b \in \Sigma$, $\bar{b}$ is its complementary nucleotide, that is, $\bar{A} = T, \bar{C} = G, \bar{G} = C$, and $\bar{T} = A$, and for symbol N we define $\bar{N} = N$. We say that a string $s = s_1 s_2 ... s_n$, $s_i \in \Sigma$, is a *base call string* of length $|s| = n$, and define its *reverse complement* as $\bar{s} = \bar{s}_n \bar{s}_{n-1} ... \bar{s}_1$. We also use the notation $s[i : j] = s_i ... s_j$, to denote a substring of length $j - i + 1$, starting at position i and ending at position j; if $j < i$, we let $s[i : j]$ be the empty string, denoted by λ. For a base call string s and $d \in \{0, 1\}$, we define the *strand function* $\pi(s, d)$ as $\pi(s, d) = s$ if $d = 0$, and $\pi(s, d) = \bar{s}$ if $d = 1$. We refer to d as the *strand direction indicator* of $\pi(s, d)$.

We define an *encoding transformation*, ϕ, as a sequence of $K + 1$ *transformation steps* that converts a base call string, s, into another base call string, $\phi(s)$, where K is a positive integer. The first K transformation steps construct a string s', starting from $s' = \lambda$, while scanning two input strings: s, and an additional (given) string $\mathcal{I}$. Each such transformation step is represented by a triplet of non-negative integers, (I, S, M), that can be interpreted as driving three elementary string operations on s', in order:

1. append a copy of the next I symbols from string $\mathcal{I}$ to s';
2. skip the following S symbols of s;

3. append a copy of the next M symbols from s to s'.

The $(K+1)$-th, and final, step of the encoding transformation consists of applying the strand function to the constructed string s', obtaining the result $\phi(s) = \pi(s', d)$, given a strand direction indicator value d. More specifically, we let $\phi = \Big(\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}, \mathcal{I}, d\Big)$, where each triplet (I_k, S_k, M_k) is comprised of an *insertion* length I_k, a *skip* length S_k, and a *match* length M_k; $\mathcal{I}$ is the *insertions base call string* of ϕ, with length $|\mathcal{I}| = \sum_{k=1}^{K} I_k$; and d is the strand direction indicator. We always apply $\phi(s)$ to strings s of length equal to $\sum_{k=1}^{K} \Big(S_k + M_k\Big)$. Algorithm 1 below summarizes the encoding transformation process. An example of the application of Algorithm 1 is shown in Figure 1.

Algorithm 1: Convert string s into string $\phi(s)$.

Input : String s, and $\phi = \Big(\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}, \mathcal{I}, d\Big)$
Output: String $\phi(s)$

```
Let s' = λ
Let i = j = 1
for k = 1 to k = K do
    Get (I_k, S_k, M_k) from φ
    Append I[i : i + I_k − 1] to s'
    i = i + I_k
    j = j + S_k
    Append s[j : j + M_k − 1] to s'
    j = j + M_k
end
return π(s', d)
```

A reference genome is usually composed of multiple base call strings; for example, in the case of the human genome there is at least one base call string for each chromosome. Therefore, we represent a reference genome as an ordered set, $\mathcal{R} = \{r_k\}_{1 \leq k \leq |\mathcal{R}|}$, of base call strings. Now, let q be a read base call string, and let $r \in \mathcal{R}$ be a reference base call string. We define an *atomic alignment* between q and r, denoted $\alpha(q, r)$, as an encoding transformation from a substring of r to a substring of q. More specifically, we let $\alpha(q, r) = (i, j, i', j', \phi)$, where ϕ is an encoding transformation such that $\phi(r[i' : j']) = q[i : j]$. Finally, we define a *full alignment* between q and $\mathcal{R}$, as a sequence of atomic alignments $A(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_A}$, where k_t is the index of

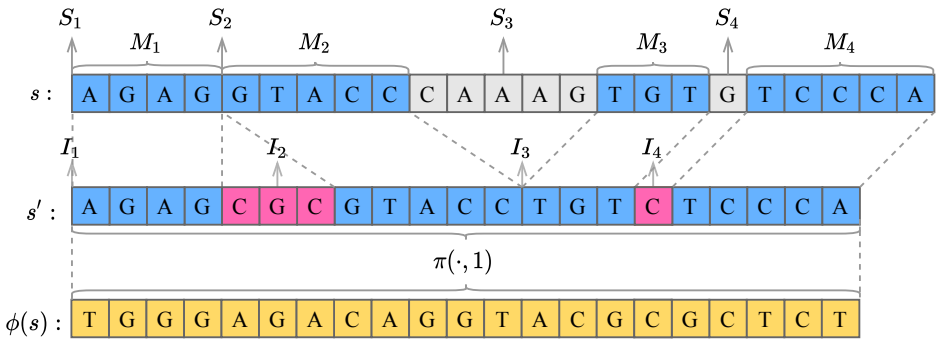

Figure 1: Constructing $\phi(s)$ by executing Algorithm 1 with inputs: s, and encoding transformation $\phi = \Big(\big((0,0,4),(3,0,5),(0,5,3),(1,1,5)\big), \mathcal{I} = \text{CGCC}, d = 1\Big)$. Notice that when an insertion length or a match length is 0, no symbols are appended to s' from $\mathcal{I}$ or s, respectively. Also, if the strand direction indicator is $d = 0$, then $\phi(s) = s'$.

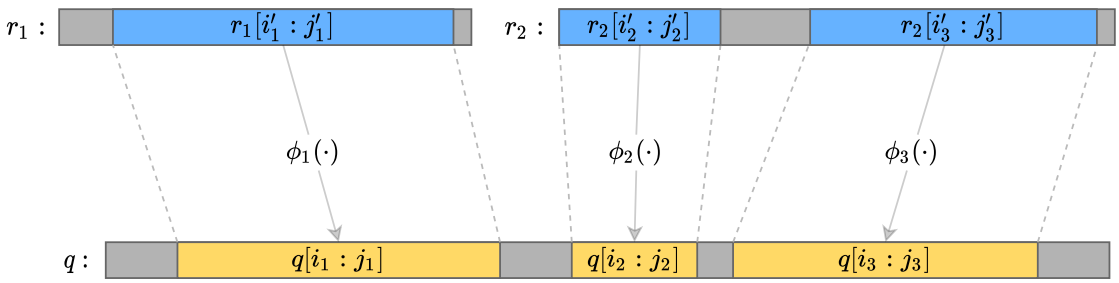

Figure 2: Example of a full alignment $A(q, \mathcal{R}) = \{\alpha_1(q, r_1), \alpha_2(q, r_2), \alpha_3(q, r_2)\}$, between a base call string q and a set of reference base call strings $\mathcal{R} = \{r_1, r_2\}$.

a reference string in $\mathcal{R}$, and L_A is the length of (i.e., number of atomic alignments in) $A(q, \mathcal{R})$. We consistently use the same subscript t of an atomic alignment $\alpha_t(q, r_{k_t})$ to denote all of its components, i.e., $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$. In Figure 2 we show an example of a full alignment. We refer to a full alignment $A(q, \mathcal{R})$ as *non-overlapping*, if none of the aligned substrings of q overlap (such as the one shown in Figure 2). Notice that the atomic alignments in the example do not cover the whole read q, a situation that may occur also in the general case. We refer to segments of q not covered by $A(q, \mathcal{R})$ as *unaligned*. Later on, we will discuss how these segments are dealt with in our algorithms.

2.2 RENANO: A reference-dependent compression and decompression scheme

In this section we present RENANO, an algorithm that compresses the base call strings of a FASTQ file assuming that the set of reference strings $\mathcal{R}$ is available to both the compressor and the decompressor. We further assume that the full alignment $A(q, \mathcal{R})$ for each base call string q against the set of reference strings $\mathcal{R}$ is available during the compression stage. In practice, this means that we assume an alignment tool has been run on the base call strings in the file against the set of reference strings $\mathcal{R}$, and that $A(q, \mathcal{R})$ has been derived from the alignment information obtained (details on how this is done are provided in Section 2.4). For convenience, we also assume the atomic alignments in $A(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_A}$ to be non-overlapping (as exemplified in Figure 2). Again in Section 2.4, we show that any full alignment can be transformed to satisfy this assumption.

As discussed in Section 1, RENANO builds over its predecessor ENANO [5], which parses and compresses the FASTQ file in blocks that always contain a whole number of reads. For consistency, our algorithm compresses the sequence of read base call strings contained in a block of the FASTQ file, denoted by $Q = q_1, \cdots, q_{|Q|}$, independently of the base call strings contained in other blocks. This makes the compression scheme compatible with the parallelization strategy implemented in ENANO.

Recall from Section 2.1 that, given an atomic alignment $\alpha_t(q, r) = (i, j, i', j', \phi)$, we can reconstruct the substring $q[i : j] = \phi(r[i', j'])$ by executing Algorithm 1 with inputs ϕ and $r[i', j']$. As we have access to the reference string r during decompression, encoding the atomic alignment $\alpha_t(q, r)$ suffices to describe $q[i : j]$. Extending this idea, RENANO compresses the aligned substrings of a read base call string q by encoding the full alignment $A(q, \mathcal{R})$, while the unaligned parts of q are encoded as raw base call strings, as in ENANO. We now proceed to describe the compression scheme in detail.

2.2.1 The encoding algorithm

To encode Q, we split the various elements of the representations of its read base call strings q_i, i.e, the aligned substrings represented in $A(q_i, \mathcal{R})$ and the unaligned substrings, into separate sequences, which we call *streams*. We refer to the stream values that comprise a representation of any of these elements as its *stream representation*. To help improve compression performance, the streams are designed so that each stream gathers parts of the representations that we expect to be correlated. The streams, in turn, are encoded independently. Different streams may contain different types of data. In particular, as described in more detail below, we will have streams comprised of raw base call symbols, streams comprised of binary symbols,

and streams comprised of non-negative integers. For each stream S of the latter, we define a parameter η_S that represents the bit size representation of the integers, where η_S is, for convenience, assumed to be a positive multiple of 8. The values of these parameters used in our experiments are discussed later in Section 3.3. Specifically, we define the following streams:

- $\mathcal{B}$: base call strings stream used to encode individual raw base call symbols (A, C, G, T, N), which include: the unaligned base call strings, and the insertion base call strings $\mathcal{I}$ that are part of the encoding transformations.
- $\mathcal{L}$: base call string lengths stream used to store the non-negative integer $|q|$, i.e., the length of each read basecall string q, using an $\eta_{\mathcal{L}}$-bit representation.
- $\mathcal{Q}$: full alignment sizes stream used to store the non-negative integers L_A, using an $\eta_{\mathcal{Q}}$-bit representation.
- $\mathcal{S}$: starting position increments stream for aligned substrings of q, i.e., the non-negative integers $i_t - i_{t-1}$, with the convention $i_0 = 0$, using an $\eta_{\mathcal{S}}$-bit representation.
- $\mathcal{E}$: lengths stream for aligned substrings of q, i.e., the non-negative integers $z_t = j_t - i_t + 1$, using an $\eta_{\mathcal{E}}$-bit representation.
- $\mathcal{U}$: aligned reference string identity indexes stream used to store the non-negative integers r_{k_t}, using an $\eta_{\mathcal{U}}$ -bit representation.
- $\mathcal{S}'$: aligned reference substring starting positions stream, which stores the non-negative integers i'_t, using an $\eta_{\mathcal{S}'}$-bit representation.
- $\mathcal{D}$: strand direction indicator stream, which stores the binary values of $d_t \in \{0, 1\}$.
- $\mathcal{N}$: insertion lengths stream that stores the non-negative integers I_k, using an $\eta_{\mathcal{N}}$-bit representation.
- $\mathcal{K}$: skip lengths stream, which stores the non-negative integers S_k, using an $\eta_{\mathcal{K}}$-bit representation.
- $\mathcal{M}$: match lengths stream used to store the non-negative integers M_k, using an $\eta_{\mathcal{M}}$-bit representation.

For convenience, we define $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ as the set of all the proposed streams.

Algorithm 2: Encode a sequence of read base call strings.

Input : Sequence of read base call strings $Q = q_1, \cdots, q_{|Q|}$, full alignment $A(q_i, \mathcal{R})$ of each q_i in Q

Output: An encoding of Q is output to a bit stream F

1 Let $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ be a set of empty streams
2 Output a η_F-bit representation of the non-negative integer $|Q|$ to F
3 **for** $i = 1$ *to* $i = |Q|$ **do**
4 | Incorporate a stream representation of q_i into $\mathbb{S}$ using Algorithm 3 (see Section 2.2.1.1) with q_i and $A(q_i, \mathcal{R})$ as inputs
5 **end**
6 **foreach** *stream* $S \in \mathbb{S}$ **do**
7 | Generate an encoding X of S as explained in Section 2.2.1.2
8 | Let n_X be the length of X, in bytes. Output a η_F-bit representation of n_X to F
9 | Output X to F
10 **end**

In Algorithm 2 we present the general compression scheme for a sequence of base call strings $Q = q_1, \cdots, q_{|Q|}$. The algorithm receives Q as input together with a full alignment $A(q_i, \mathcal{R})$ for each q_i in Q, and generates an encoding of Q that is output to a bit stream F. All the streams in $\mathbb{S}$ are initialized as empty sequences in Step 1. In Step 2, the algorithm outputs the number of read base call strings in Q, so that the decoder can retrieve how many read base call strings are encoded into the streams. In Step 3, the algorithm loops over each base call string q_i, splitting a representation of q_i into the various streams in $\mathbb{S}$ by running Algorithm 3, which we discuss later on. Then, the loop in Step 6 encodes the content of each stream S in $\mathbb{S}$. To this end, in Step 7, S is encoded using compression techniques that we present later in this section. Next, the encoding of the stream is output to F preceded by its length (in bytes). This ensures that the decoder can access each stream separately. Both numbers output in steps 2 and 8 are encoded as η_F-bit integers, where η_F is an implementation parameter discussed in Section 3.3. We now proceed to describe steps 4 and 7 of the algorithm in detail.

2.2.1.1 Splitting the representation of each base call string into separate streams (Step 4 of Algorithm 2).

To generate a stream representation of an individual read base call string q, we use Algorithm 3.

Algorithm 3: Describe a full alignment $A(q, \mathcal{R})$ and the needed parts of base call string q using the streams in $\mathbb{S}$.

Input : Read base call string q, full alignment $A(q, \mathcal{R})$
Output: Updated streams in $\mathbb{S}$

1 Append the string length $|q|$ to stream $\mathcal{L}$
2 Append the number of atomic alignments L_A to stream $\mathcal{Q}$
3 Sort the atomic alignments in $A(q, \mathcal{R})$ in increasing order of i_t
4 Let $i = 1$
5 **for** $t = 1$ *to* $t = L_A$ **do**
6 **if** $i < i_t$ **then**
7 Append raw base call string $q[i : i_t - 1]$ to stream $\mathcal{B}$
8 **end**
9 $i = i_t$
10 Append reference string index k_t to stream $\mathcal{U}$
11 Append a stream representation of $\alpha_t(q, r_{k_t})$ to the streams in $\mathbb{S}$ using Algorithm 4 (see below)
12 $i = j_t + 1$
13 **end**
14 **if** $i \leq |q|$ **then**
15 Append base call string $q[i : |q|]$ to stream $\mathcal{B}$
16 **end**

The algorithm starts by appending the length of q to stream $\mathcal{L}$ (Step 1), and the number of atomic alignments in $A(q, \mathcal{R})$ to stream $\mathcal{Q}$ (Step 2). A representation of q is incorporated into the streams progressively, scanning q from start to end as the algorithm iterates over the atomic alignments in $A(q, \mathcal{R})$, which are previously sorted in increasing order of i_t (Step 3). The variable i, initialized to $i = 1$ in Step 4, maintains the starting position of the portion of q that remains to be incorporated into the stream representation. In Step 5, the algorithm loops over each atomic alignment in $A(q, \mathcal{R})$. Each iteration starts by checking, in Step 6, if the current value of i is smaller than the starting position of the current atomic alignment, i_t. If so, then $q[i : i_t - 1]$ is an unaligned substring of q, which is directly appended to stream $\mathcal{B}$ in Step 7. The index k_t of the reference string of the current atomic alignment, $\alpha_t(q, r_{k_t})$, is appended to the stream $\mathcal{U}$ in Step 10, followed by an execution of Algorithm 4,

which appends a representation of $\alpha_t(q, r_{k_t})$ itself to the streams in $\mathbb{S}$. The iteration ends by updating the value of i to the next position yet to be represented, $i = j_t + 1$. By the end of the loop in Step 5, every symbol in q has been represented as part of either a raw base call string in Step 7 or an atomic alignment in Step 11, with the possible exception of the trailing symbols in an unaligned suffix of q, which are incorporated in Step 15.

Notice that the starting and ending positions of the unaligned substrings are not explicitly described, as they are fully determined by the starting and ending positions of the atomic alignments, together with the total length of q.

To generate a stream representation of each atomic alignment $\alpha_t(q, r_{k_t})$, we use Algorithms 4 and 5.

Algorithm 4: Generate a stream representation of an atomic alignment.

Input : Full alignment $A(q, \mathcal{R})$ and index $t \geq 1$ of an atomic alignment $\alpha_t(q, r_{k_t})$ in $A(q, \mathcal{R})$

Output: A representation of $\alpha_t(q, r_{k_t})$ is appended to the streams in $\mathbb{S}$

1 Let $\delta = i_t - i_{t-1}$ `/* recall that` $i_0 = 0$ `*/`
2 Append δ to stream $\mathcal{S}$
3 Append $z = j_t - i_t + 1$ to stream $\mathcal{E}$
4 Append the reference starting index i'_t to stream $\mathcal{S}'$
5 Append the insertions base call string $\mathcal{I}$ of ϕ_t to stream $\mathcal{B}$
6 Append the strand direction indicator d of ϕ_t to stream $\mathcal{D}$
7 **foreach** $(I_k, S_k, M_k) \in \phi_t$ **do**
8 | Run Algorithm 5 (see below) with triplet (I_k, S_k, M_k) as input
9 **end**

Algorithm 4 starts by appending a differential representation of the starting position of the aligned substring, i_t, to stream $\mathcal{S}$ (Step 2). With this representation, the positions i_t are implicitly determined by the successive differences, $i_t - i_{t-1}$, which are stored in $\mathcal{S}$ (recall that by convention, $i_0 = 0$, and hence the difference is well defined for all $t \geq 1$). Since $A(q, R)$ is sorted in increasing order of i_t, these differences are non-negative and, in general, with a high frequency of relatively low values, which we exploit to achieve efficient compression. In Step 3, the length of the aligned substring, $z = j_t - i_t + 1$, is appended to stream $\mathcal{E}$. We choose to store the length of the substring instead of its ending position, as this also generally results in a high frequency of relatively low values. Step 4 appends the starting position of the reference substring, i'_t, to stream $\mathcal{S}'$, and the following steps generate a stream

representation of the encoding transformation ϕ_t. In Steps 5 and 6, the insertions base call string $\mathcal{I}$ and the strand direction indicator d are appended to streams $\mathcal{B}$ and $\mathcal{D}$, respectively. We do not store the length of the insertions base call string $\mathcal{I}$, as it can be calculated as the sum of the insertion lengths, $|\mathcal{I}| = \sum_{k=1}^{K} I_k$. Finally, in Step 7, the algorithm loops through every triplet of the encoding transformation, and incorporates its stream representation into $\mathbb{S}$ using Algorithm 5.

Algorithm 5: Recursively generate a stream representation of a triplet (I, S, M), with constrained-length operations.

Input : Triplet (I, S, M)
Output: A representation of (I, S, M) is appended to the streams in $\mathbb{S}$

```
if I > T_N then
    Run Algorithm 5 with triplet (T_N, 0, 0) as input
    Run Algorithm 5 with triplet (I − T_N, S, M) as input
else if S > T_K then
    Run Algorithm 5 with triplet (0, T_K, 0) as input
    Run Algorithm 5 with triplet (I, S − T_K, M) as input
else if M > T_M then
    Run Algorithm 5 with triplet (I, S, T_M) as input
    Run Algorithm 5 with triplet (0, 0, M − T_M) as input
else
    Append I, S, and M to streams N, K, and M, respectively
end
```

We observed, empirically, that most of the values that typically compose triplets (I, S, M) lie within a narrow range of small non-negative integers, with, possibly, a few outliers. For this reason, it is convenient to choose small values for the bit size parameters $\eta_{\mathcal{N}}$, $\eta_{\mathcal{K}}$, and $\eta_{\mathcal{M}}$, used to represent I, S, and M in the streams $\mathcal{N}$, $\mathcal{K}$, and $\mathcal{M}$, respectively. These parameters, in turn, determine thresholds, $T_{\mathcal{N}} = 2^{\eta_{\mathcal{N}}} - 1$ for insertion lengths, $T_{\mathcal{K}} = 2^{\eta_{\mathcal{K}}} - 1$ for skip lengths, and $T_{\mathcal{M}} = 2^{\eta_{\mathcal{M}}} - 1$ for match lengths. Since a stream S, for $S \in \{\mathcal{N}, \mathcal{K}, \mathcal{M}\}$, does not admit a single-value representation of an operation length larger than the threshold T_S, every triplet (I, S, M) with such an operation length is divided by Algorithm 5 into a sequence of triplets, with trimmed operation lengths that do satisfy these constraints. This substitute sequence represents a combination of string operations that produce the same result as the original operations represented by (I, S, M).

Algorithm 5 proceeds recursively. If an operation length, say I, is larger than the

corresponding threshold, $T_{\mathcal{N}}$, then the algorithm is recursively invoked for two triplets. One of them represents a single insertion of maximal length, $T_{\mathcal{N}}$. The other represents the remaining operations, in this case an insertion of length $I - T_{\mathcal{N}}$ together with the original skip and match operations, of lengths S and M, respectively. Skip and match lengths exceeding the thresholds $T_{\mathcal{K}}$ and $T_{\mathcal{M}}$, respectively, are handled analogously. Since an invocation of the algorithm for triplets (I, S, M) having at least one value exceeding the corresonding theshold produces recursive calls for triplets (I', S', M'), where at least one of I', S', M' is strictly smaller than I, S, M, respectively (and none of them is larger), then eventually no operation length exceeds its threshold, and the algorithm generates a stream representation of (I, S, M) directly in Step 11, with no further recursive calls.

As a last comment regarding Algorithm 4, notice that j'_t is not explicitly represented in any stream. Nevertheless, the length, z', of the reference substring is implicitly determined by the skip and match lengths of the encoding transformation,

$$z' = \sum_{k=1}^{K} \Big(S_k + M_k \Big) ,$$

where K is the number of transformation steps. The value of j'_t, in turn, can be calculated as $j'_t = i'_t + z' - 1$. We also notice that the number of transformation steps, K, is not explicitly represented either; we show later in Section 2.2.2.1 that it can be reconstructed by the decoder from the streams in $\mathbb{S}$.

2.2.1.2 Encoding the streams (Step 7 of Algorithm 2).

The algorithms described so far have constructed a set $\mathbb{S}$ of streams of various data types, from which, given the reference strings $\mathcal{R}$, the base call strings in the original FASTQ file can be fully reconstructed. To complete a compression algorithm, we must encode these streams, as efficiently as possible, into bitstreams. Next we describe the encoding of RENANO.

To encode each stream we follow a method similar to ENANO [5], which combines *context modeling* [23] with *arithmetic encoding* [24]. In short, we encode the sequence of symbols of a stream by sequentially estimating, for each position of the sequence, a probability distribution over the set of possible values for the symbols in the stream (its *alphabet*). Each symbol position is assigned a specific *context*, which is an abstract class that groups symbols that we model as obeying the same probability distribution. The context assigned to each position is selected from a finite set of possible contexts, based on the values of previously encoded symbols, and the probability distributions determined by each context is sequentially and adaptively estimated from statistics collected from previous occurrences of the same context. To encode each symbol,

both the symbol and the estimated probability distribution for the position of the symbol are given as input to an arithmetic encoder. The arithmetic encoder, in turn, generates a sequence of bits that encodes the provided sequence of symbols. The overall code length, in bits, for an input sequence $s_1, s_2, \ldots, s_n$ of stream symbols is

$$\ell = -x \log_2 \prod_{i=1}^{n} P(s_i|C_i) + O(1),$$

where C_i denotes the context assigned to location i in the sequence, and $P(s_i|C_i)$ is the probability assigned to s_i in this context. This code length is, in a well-defined sense, optimal up to an additive constant [24]. On the decoder side, the symbol sequence can be perfectly decoded by calculating the same sequence of probability distributions. Notice that this is possible because the probability distributions are estimated only from occurrences of previously encoded/decoded symbols. We refer the reader to Section 2 of [5] for a more detailed explanation of the method. For the proposed algorithm, we define a specific context model for each stream, as discussed below.

To encode a base call symbol from the stream $\mathcal{B}$, we use the previous k base call symbols as the context. Thus, for any given k-tuple $s = s_1 s_2 \ldots s_k$, all symbols that follow s in $\mathcal{B}$ are collected into the same context (we use a default value of $k = 7$). For the strand direction indicator stream, $\mathcal{D}$, we directly encode each binary value $d_t \in \{0, 1\}$ as a single bit, as experiments show that the strand direction indicators are close to uniformly distributed. The remaining streams are comprised of sequences of non-negative integers of various representation sizes. We use the same method to encode each such stream. To encode an η-bit integer stream, where η is a multiple of 8, we split the bit representation of each integer of the stream into $\frac{\eta}{8}$ bytes. We encode each byte separately, starting from the least significant to the most significant byte, using a specific context for each of the $\frac{\eta}{8}$ byte positions. In other words, all the least significant bytes of the integers in the stream are encoded in the same context, all the second least significant bytes are encoded in the same separate context, and so forth. Since, as mentioned, these non-negative integer sequences tend to concentrate towards the lower end of their range, the higher order bytes end up being highly compressible, and there is little penalty in over-estimating the selected integer size η for each stream.

2.2.2 The decoding algorithm

In Algorithm 6 we present the general decompression scheme that decodes the sequence of base call strings $Q = q_1, \cdots, q_{|Q|}$ given the set of reference base call strings $\mathcal{R}$ and the compressed file, accessed through an input bitstream, F. Notice that,

as the encoding and decoding algorithms work in lockstep, at any stage of the decoding process, the decoder knows exactly what stream to read from and how much information it needs from it.

Algorithm 6: Decode the sequence of read base call strings in a block of the FASTQ file.

Input : Set of reference base call strings $\mathcal{R}$, input stream F
Output: Sequence of read base call strings $Q = q_1, \cdots, q_{|Q|}$

1 Let Q be an empty sequence
2 Let $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ be a set of empty streams.
3 Retrieve the number of encoded read base call strings from F into n_Q
4 **foreach** *stream* $S \in \mathbb{S}$ **do**
5 | Retrieve the length in bytes, n_X, of the encoding of S from F
6 | Retrieve n_X bytes from F into X
7 | Decode S from X
8 **end**
9 **for** $i = 1$ *to* $i = n_Q$ **do**
10 | Reconstruct q_i using Algorithm 7 (see Section 2.2.2.1) with $\mathbb{S}$ and $\mathcal{R}$ as inputs
11 | Append q_i to Q
12 **end**
13 **return** Q

Algorithm 6 reverses the steps of Algorithm 2. It starts by initializing Q as an empty sequence in Step 1 and all the streams in $\mathbb{S}$ as empty streams in Step 2. The contents of both Q and $\mathbb{S}$ are constructed in subsequent steps. First, the number of base call strings to be decoded and added to Q, n_Q, is obtained from F in Step 3. Next, each stream in $\mathbb{S}$ is decoded individually. This is done by retrieving the length, in bytes, of the encoding of the stream in Step 5, followed by the encoding itself in Step 6, which is decoded in Step 7. Once the streams in $\mathbb{S}$ are decoded in Step 10, the algorithm reconstructs each of the read base call strings of Q from its stream representation by running Algorithm 7 with $\mathbb{S}$ and $\mathcal{R}$ as inputs. Next, we describe Step 10 in more detail.

2.2.2.1 Reconstructing read base call strings (Step 10 of Algorithm 6).

Algorithm 7 reconstructs a base call string q from its stream representation in $\mathbb{S}$ and the set of reference strings $\mathcal{R}$.

Algorithm 7: Reconstruct base call string q.

Input : Reference strings $\mathcal{R}$, set of streams $\mathbb{S}$
Output: Read base call string q

1 Let $q = \lambda$ and $i = 1$
2 Retrieve the number of symbols to be added to q from $\mathcal{L}$ into n_q
3 Retrieve the number of atomic alignments L_A from $\mathcal{Q}$
4 **for** $t = 1$ *to* $t = L_A$ **do**
5 Retrieve reference string index k_t from $\mathcal{U}$
6 Retrieve i_t from $\mathcal{S}$
7 **if** $i < i_t$ **then**
8 Extend q by retrieving $q[i : i_t - 1]$ from $\mathcal{B}$
9 **end**
10 Reconstruct $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ by retrieving j_t, i'_t, j'_t, and ϕ_t using Algorithm 8 (see below) with i_t and $\mathbb{S}$ as inputs
11 Extend q by executing Algorithm 1 to obtain $q[i_t : j_t]$ from ϕ_t, and $r_{k_t}[i'_t : j'_t]$
12 Let $i = j_t + 1$
13 **end**
14 **if** $i \leq n_q$ **then**
15 Extend q by retrieving $q[i : n_q]$ from $\mathcal{B}$
16 **end**
17 **return** q

The Algorithm 7 reverses the steps of Algorithm 3. In the algorithm, q is initialized to an empty string in Step 1, and it is reconstructed progressively, appending symbols up to length n_q (obtained in Step 2). The variable i, initialized in Step 1, maintains the position of the next symbol to be added to q. In Step 3, the algorithm retrieves the number of atomic alignments L_A that encode substrings of q, and loops over each atomic alignment in Step 4. Each atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ is retrieved in a two state fashion. Firstly, k_t and i_t are obtained in Steps 5 and 6, respectively, of Algorithm 7. The index i_t is used to determine, in Steps 7–9, the length of an eventual unaligned string that may precede the atomic alignment. The remaining components of the atomic alignment, i.e., j_t, i'_t, j'_t, and ϕ_t, are retrieved in a second stage by calling Algorithm 8 in step 10 with i_t and $\mathbb{S}$ as inputs.

Algorithm 8 reverses the steps of Algorithm 4. Notice that, during the reconstruction process, we do not have direct access to the number of transformation steps, K,

Algorithm 8: Retrieve an atomic alignment from its stream representation in $\mathbb{S}$.

Input : Starting position i_t, and set of streams $\mathbb{S}$
Output: Atomic alignment $(i_t, j_t, i'_t, j'_t, \phi_t)$

1 Retrieve z from $\mathcal{E}$
2 Let $j_t = i_t + z - 1$
3 Retrieve the reference starting index i'_t from $\mathcal{S}'$
4 Let $z_1 = 0, j'_t = i'_t - 1, k = 1, \mathcal{I} = \lambda$
5 **repeat**
6 Retrieve I_k from stream $\mathcal{N}$
7 Retrieve S_k from stream $\mathcal{K}$
8 Retrieve M_k from stream $\mathcal{M}$
9 Add (I_k, S_k, M_k) to ϕ
10 Retrieve I_k base calls from $\mathcal{B}$ and append them to $\mathcal{I}$
11 Let $j'_t = j'_t + S_k + M_k$
12 Let $z_1 = z_1 + I_k + M_k$
13 Let $k = k + 1$
14 **until** $z_1 = z$
15 Retrieve strand direction indicator d from $\mathcal{D}$
16 **return** $(i_t, j_t, i'_t, j'_t, \phi_t)$

of the encoding transformation ϕ. However, the length of a string resulting from applying ϕ to a reference string equals the sum of the lengths of the insertion and match operations in ϕ. Therefore, for the length of the aligned substring, z, retrieved in Step 1, we have

$$z = \sum_{k=1}^{k=K} \Big(I_k + M_k \Big) .$$

Consequently, the loop in Step 5 retrieves triplet operations, accumulating the lengths of the insertion and match operations in z_1, until z_1 reaches z.

2.3 RENANO$_{\text{IND}}$: a reference-dependent compression scheme, with a reference-independent decompression scheme

In this section we present RENANO$_{\text{IND}}$, a variation of RENANO that makes the decompression process independent of the set of reference base call strings $\mathcal{R}$. The main idea is to create a new artificial reference string, r', composed of carefully

selected parts of the set of reference base call strings $\mathcal{R}$ and encode it in the compressed file. The atomic alignments associated to the read base call strings are then modified to align against r' instead of the original strings in $\mathcal{R}$. At that point, we can compress the read base call strings of the FASTQ file by applying the same encoding scheme presented in Section 2.2.1, with $\mathcal{R} = \{r'\}$. Subsequently, we can decode the read base call strings by first decoding r', followed by applying the decoding scheme of Section 2.2.2, again with $\mathcal{R} = \{r'\}$.

To create the new reference string r', we start by analyzing the full alignments $A(q, \mathcal{R})$ obtained against the full reference set $\mathcal{R}$. We are interested in keeping the parts of the reference base call strings in $\mathcal{R}$ that are deemed useful for the compression of the read base call strings q. More concretely, we first say that an atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi)$, *uses* a position h of r_{k_t} if $i'_t \leq h \leq j'_t$. Naturally, the positions of a reference string r_k that are useful for compression are those that are actually used by atomic alignments, and we would like to discard positions that are not used by any atomic alignment. Moreover, taking into account that describing r' incurs a coding cost, positions used by only one atomic alignment are not beneficial for compression either, as both the raw base call symbol for that position in the reference r' and the alignment information need to be encoded. We empirically observe that good compression results are obtained by keeping the positions of the reference base call strings that are used by at least two atomic alignments, with no significant improvement for larger thresholds. Consequently, we say that h is a *surviving position* of r_k if it is used by at least two atomic alignments. We define the new reference r' as the concatenation of the bases in surviving positions of the reference strings. Figure 3 shows an example of the proposed construction of r'.

Clearly, r' can be constructed in a single pass through all atomic alignments, by keeping count of the number uses of each position of r_j, $1 \leq j \leq |\mathcal{R}|$. Moreover, as a byproduct of this construction, we can obtain a mapping ψ that maps each surviving position h of r_j to its corresponding position, $\psi(j, h)$, in r'.

Once the new reference string r' is built, RENANO$_{\text{IND}}$ generates a binary encoding, denoted by $\text{Enc}(r')$, using the same technique proposed for encoding the stream of base call strings $\mathcal{B}$ in Section 2.2.1.2. The algorithm then stores $\text{Enc}(r')$ into the compressed file by outputting a fixed-size binary representation of its length $|\text{Enc}(r')|$, followed by $\text{Enc}(r')$ itself. Clearly, this process can be reversed on the decoder side, giving the decoder access to r'.

To generate the artificial reference r', we made use of the information in the atomic alignments, which were aligned against the original reference strings in $\mathcal{R}$. However, the decoder has access to r', but not to $\mathcal{R}$. Therefore, before being described to the decoder, each atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$, must be modified to align against r'. If the atomic alignment uses no surviving positions (see $r_1[i'_4 : j'_4]$ in Figure 3), it is fully discarded and removed from its respective full alignment. Otherwise,

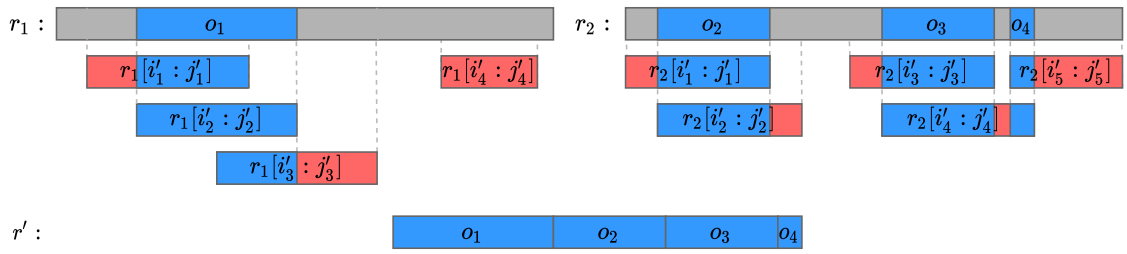

Figure 3: Example of the construction of an artificial reference string r'. The top row represents the original reference strings r_1 and r_2. The bottom row represents the constructed new reference r'. The three middle rows represent reference substrings, $r_{k_t}[i'_t : j'_t]$, used by the atomic alignments. The string r' is constructed by concatenating the bases in surviving positions (condensed in four segments, o_1, o_2, o_3, and o_4, marked in blue) of r_1 and r_2. Not-surviving positions (marked in gray) are discarded. In the reference substrings of the atomic alignments, the substrings that correspond to surviving positions are marked in blue, while substrings that correspond to not-surviving positions are marked in red.

the atomic alignment is assigned a new reference substring of r', $r'[i''_t : j''_t]$, with $i''_t = \psi(k_t, h_i)$ and $j''_t = \psi(k_t, h_j)$, where h_i and h_j are smallest and largest surviving positions used by $\alpha_t(q, r_{k_t})$, respectively. In addition, ϕ_t is modified so that $q[i_t : j_t] = \phi_t(r'[i''_t : j''_t])$, taking into account that some positions of the original reference substrings are no longer present in r'. Specifically, all triplet operations (I_k, S_k, M_k) where a portion of the match or skip operations refers to not-surviving positions, are transformed: match operations are transformed into insertion operations, and the portions of skip operations that lie on not-surviving positions are removed. If during this process a triplet operation (I_k, S_k, M_k) ends up with a value of $M_k = 0$, and it is not the last triplet of the encoding transformation, then the triplet is merged with the next triplet operation, $(I_{k+1}, S_{k+1}, M_{k+1})$, resulting in a single triplet operation $(I_k + I_{k+1}, S_k + S_{k+1}, M_{k+1})$.

At first sight, it may seem that the construction of r' requires loading all the atomic alignments into memory at once. In practice, however, we can build r' by loading, initially, only the information of the atomic alignments strictly required to determine the surviving positions, that is, the indexes of the aligned substrings k_t, i'_t and j'_t. The remaining information of each atomic alignment is loaded only at the time of its encoding, where also the transformation of ϕ_t is performed.

2.4 Alignment Information

Here we describe the format in which the alignment information is provided to RENANO and RENANO$_{\text{IND}}$, and how we construct, for each base call string q in the FASTQ file, a full alignment $A(q, \mathcal{R})$ (as defined in Section 1, in our internal representation) against the set of reference sequences $\mathcal{R}$.

To obtain the alignment information of a specific FASTQ file against a reference genome, our software implementation of RENANO receives an input text file in *Pairwise mApping Format* [18] (PAF), typically used for nanopore sequence alignment. Each line of a PAF file, which represents an alignment between a *query* base call string and a *target* base call string, consists of a list of TAB-delimited fields. For our application, each query base call string is a portion of a read base call string of a FASTQ file to be compressed, and each target base call string is a portion of a base call string of a reference genome, stored in a FASTA file. In Table 1 we describe the fields that are relevant to RENANO and RENANO$_{\text{IND}}$. Most fields are identified by position, with the only exception of the field labeled *cs*, which is identified through a tag at the beginning of the field.

Field	Type	Description
1	string	Query string name
3	int	Query start coordinate
4	int	Query end coordinate
5	char	'+' if query/target on the same strand; '-' if opposite
6	string	Target string name
8	int	Target start coordinate on the original strand
9	int	Target end coordinate on the original strand
cs	string	Base-alignment string in *cs* format

Table 1: Fields of the PAF alignment format relevant to RENANO and RENANO$_{\text{IND}}$.

Our implementation reads and parses the input PAF file, line by line, to generate full alignments for each base call string q of the FASTQ file. The first field in each line of a PAF file is the sequence identifier of the read base call string in the FASTQ file. This read base call string, together with the starting and ending positions in fields 3 and 4 of the PAF file line, respectively, determine the query base call string of the alignment represented by this line. Each read base call string of a FASTQ file, q, may participate in zero or more alignments; each one is represented by a separate line in a PAF file, all of which bear the same sequence identifier of q in the first field. If a read base call string q has no associated alignments, then $A(q, \mathcal{R})$ is set to an empty list, and $L_A = 0$. Otherwise, for each line in the PAF file where q is the query string,

we build an atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$. Recall from Section 2 that RENANO assumes that the atomic alignments of a full alignment $A(q, \mathcal{R})$ are non-overlapping, that is, that the aligned substrings, $q[i_t : j_t]$, of the atomic alignments in $A(q, \mathcal{R})$ do not overlap. However, this condition is not necessarily satisfied by the alignments extracted from the PAF file. Consequently, we first create an auxiliary full alignment $\bar{A}(q, \mathcal{R})$ by parsing every atomic alignment associated to q in the PAF file, and we later use $\bar{A}(q, \mathcal{R})$ to generate a non-overlapping full alignment $A(q, \mathcal{R})$.

To generate an atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ from a line of the PAF file, we use the values in fields 3 and 4 to determine the aligned q substring, $q[i_t : j_t]$, where i_t is the value in field 3 and j_t is the value in field 4. We also use the values in fields 6, 8, and 9, to determine the reference substring of the atomic alignment, $r_{k_t}[i'_t : j'_t]$, where the index of the reference substring k_t is determined by the name of the target string in field 6, and fields 8 and 9 determine i'_t and j'_t, respectively. Finally, to determine the encoding transformation ϕ_t, we use field 5 and the *cs* tag. Field 5 indicates if the alignment is between the original query string and the target string, in which case the value is '+', or if it is between the reverse complement of the query string and the target string, in which case the value is '-'. We use this field to determine the strand direction indicator d of ϕ_t, such that $d = 0$ if field 5 is '+', and $d = 1$ if field 5 is '-'. Finally, the *cs* field has a base-alignment string, called *cs*-string, which encodes the difference between the query string and the target string, as a sequence of string operations, which include: *matches*, *insertions*, *substitutions* (single nucleotide polymorphisms), and *deletions*. For more information on *cs*-strings we refer the reader to the Supplementary Data of [18]. We use Algorithm 9 (see below) to process the *cs*-string, and obtain the triplet sequence $\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}$ and the insertions base call string, $\mathcal{I}$, of an encoding transformation ϕ_t, such that $q[i_t : j_t] = \phi_t(r_{k_t}[i'_t : j'_t])$.

The value of input d determines if the *cs*-string alignment utilizes the original query string, or its reverse complement. Thus, in Step 1, the algorithm starts by defining the auxiliary string $q' = \pi(q[i_t : j_t], d)$, which is used to build the insertions base call string. In Step 2, the algorithm initializes the insertions base call string $\mathcal{I}$ as the empty string, and the triplet sequence T as an empty sequence. The triplet operations and the insertions base call string are progressively constructed by scanning the *cs*-string and q' from start to end as the algorithm iterates over the *cs*-string operations, appending triplets to T and base call strings to $\mathcal{I}$ as necessary. The variable i, initialized to $i = 1$ in Step 3, maintains the starting position of the portion of q' that remains to be processed. The auxiliary variable I represents the insertion length value of the next triplet to be appended, while S represents the skip length value of the next triplet to be appended. Both variables are initialized to 0 in Step 3.

In Step 4, the algorithm loops over each operation, X, in the *cs*-string, and takes a different action depending on the type of operation being processed. If the operation is

Algorithm 9: Obtain the sequence of triplet operations and the insertions base call string, from a *cs*-string, the aligned substring of q, and the strand direction indicator d.

Input : *cs*-string, base call substring $q[i_t : j_t]$, strand direction indicator d
Output: Sequence of triplets $\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}$, and insertions base call string $\mathcal{I}$

```
Let q' = π(q[i_t : j_t], d)
Let I = λ be an empty string, and T = {} an empty sequence
Let i = 1, I = 0, and S = 0
foreach operation X in the cs-string do
    Let L be the length of operation X
    if X is an insertion then
        I = I + L
        Append q'[i : i + L − 1] to I
        i = i + L
    else if X is a deletion then
        S = S + L
    else if X is a substitution then
        I = I + 1
        Append q'[i : i] to I
        i = i + 1, S = S + 1
    else if X is a match then
        if L > m_L then
            Append (I, S, L) to T
            I = 0, S = 0
        else
            I = I + L, S = S + L
            Append q'[i : i + L − 1] to I
        end
        i = i + L
    end
end
if I > 0 or S > 0 then
    Append (I, S, 0) to T
end
return T and I
```

an insertion, the algorithm adds the length of the operation, L, to the current insertion length, I (Step 7), and appends the corresponding inserted base calls, $q'[i : i + L - 1]$, to string $\mathcal{I}$ (Step 8). The algorithm proceeds to adjust the value of index i accordingly (Step 9). If the operation is a deletion, the algorithm adds the deletion length L to the current skip length S, in Step 11. We interpret a substitution operation as a combination of an insertion and a deletion both of length 1. Therefore, for this kind of operations, the algorithm adds 1 to the value of I (Step 13), appends the base call in position i of q, $q'[i : i]$, to $\mathcal{I}$ (Step 14), and adds 1 to the current skip value S (Step 15). The index i is adjusted accordingly in Step 15. Finally, if the operation is a match, the algorithm checks, in Step 17, if the match length L is greater than a *minimum match length* threshold parameter, m_L. If it is, the triplet (I, S, L) is appended to T, and the variables I and S are reset to 0, in steps 18 and 19, respectively. Otherwise, the match operation is interpreted as an insertion and a deletion, both of length L. Hence, the algorithm adds L to both I and S (Step 21), and appends string $q'[i : i + L - 1]$ to $\mathcal{I}$ (Step 22). Index i is adjusted accordingly in Step 24. As a last step, if there is a remaining triplet operation that has yet to be appended to T, that is, if I or S are greater than zero, the corresponding triplet is appended to T (Step 28).

The parameter m_L regulates the trade off between the cost of encoding a potentially long sequence of atomized small matches, against the cost of encoding fewer insertion operations, which incur an additional cost of encoding extra symbols in the insertions base call string. The effect of this parameter on compression performance is studied in Section 3.3, where we choose a default value for it.

Once the alignments in the PAF file associated to a base call string q are parsed and transformed into an auxiliary full alignment sequence $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_{\bar{A}}}$, we generate a non-overlapping full alignment, $A(q, \mathcal{R})$, by executing Algorithm 10 with $\bar{A}(q, \mathcal{R})$ as input.

The algorithm starts by sorting the sequence of atomic alignments, $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_{\bar{A}}}$, in decreasing order of j_t (Step 1). In Step 2, sequence $A(q, \mathcal{R})$ is initialized with the first atomic alignment of the sorted sequence, $A(q, \mathcal{R}) = \{\alpha_1(q, r_{k_1})\}$, and its size L_A is initialized to $L_A = 1$. The algorithm progressively appends atomic alignments to $A(q, \mathcal{R})$, such that $A(q, \mathcal{R})$ always holds a non-overlapping sequence of atomic alignments. Notice that $A(q, \mathcal{R})$ is initialized as a non-overlapping sequence, as it contains a single atomic alignment. An example of a sequence of overlapping atomic alignments, sorted in decreasing order of j_t, is presented in the top (pink) box in Figure 4.

In Step 3 of the algorithm, the variable $\hat{\imath}$, initialized to $\hat{\imath} = i_1$, maintains the starting position of the last atomic alignment appended to $A(q, \mathcal{R})$. In Step 4, the algorithm loops over the remaining atomic alignments in $\bar{A}(q, \mathcal{R})$, and checks, in Step 5, if the ending position of the current atomic alignment, j_t, is smaller than $\hat{\imath}$. If so,

Algorithm 10: Generate a non-overlapping full alignment $A(q, \mathcal{R})$.

Input : Auxiliary full alignment $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_{\bar{A}}}$ obtained from the PAF file.

Output: Non-overlapping full alignment $A(q, \mathcal{R})$

Sort sequence $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_{\bar{A}}}$ in decreasing order of j_t
Let $A(q, \mathcal{R}) = \{\alpha_1(q, r_{k_1})\}$, and let $L_A = 1$
Let $\hat{\imath} = i_1$
for $t = 2$ *to* $t = L_{\bar{A}}$ **do**
  if $j_t < \hat{\imath}$ **then**
    Append $\alpha_t(q, r_{k_t})$ to $A(q, \mathcal{R})$
    $L_A = L_A + 1$
    $\hat{\imath} = i_t$
  else if $i_t < \hat{\imath}$ **then**
    $\hat{\jmath}_t = \hat{\imath} - 1$
    Obtain ϕ'_t, such that $q[i_t : \hat{\jmath}_t] = \phi'_t(r_{k_t}[i'_t : j'_t])$, by modifying ϕ_t
    Append atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, \hat{\jmath}_t, i'_t, j'_t, \phi'_t)$ to $A(q, \mathcal{R})$
    $L_A = L_A + 1$
    $\hat{\imath} = i_t$
  end
end
return $A(q, \mathcal{R})$

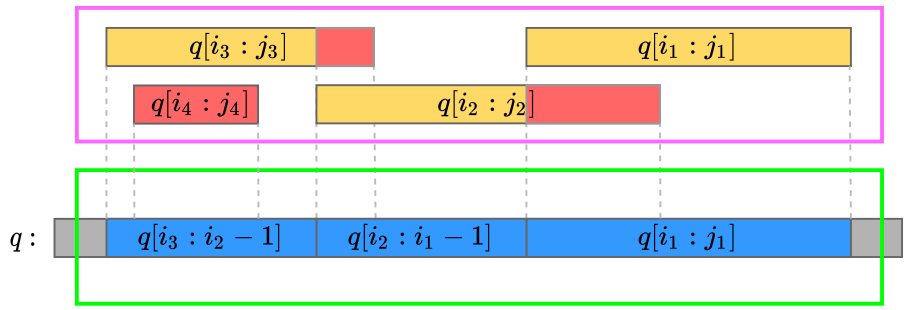

Figure 4: Example of a full alignment, $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_{\bar{A}}}$, obtained from a PAF file with overlapping between aligned q substrings, $q[i_t : j_t]$. The segments in the top box (pink) represent the original overlapping q substrings, while the segments in the bottom box (green) represent the result of running Algorithm 10 with $\bar{A}(q, \mathcal{R})$ as input.

the current alignment does not overlap with the alignments in $A(q, \mathcal{R})$ (as $i_t < j_t$), and it is directly appended to $A(q, \mathcal{R})$ in Step 7, while $\hat{\imath}$ is updated to $\hat{\imath} = i_t$, in Step 8. Otherwise, if j_t is greater or equal than $\hat{\imath}$, there is overlapping between $q[i_t : j_t]$ and $A(q, \mathcal{R})$. In this case, the algorithm checks, in Step 9, if i_t is smaller than $\hat{\imath}$. If so, the substring $q[\hat{\imath} : j_t]$, contained in $q[i_t : j_t]$, overlaps with $A(q, \mathcal{R})$ (see examples $q[i_2 : j_2]$ and $q[i_3 : j_3]$ in Figure 4). The algorithm follows by adding a modified version of the current atomic alignment to $A(q, \mathcal{R})$, discarding the overlapping portion of the alignment. First, the variable $\hat{\jmath}_t$ is initialized to $\hat{\jmath}_t = \hat{\imath} - 1$ in Step 10, which is the largest non-overlapping position of $q[i_t : j_t]$. The algorithm proceeds to construct ϕ'_t, such that $q[i_t : \hat{\jmath}_t] = \phi'_t(r_{k_t}[i'_t : j'_t])$, in Step 11, by modifying ϕ_t. Specifically, the parts of the triplet sequence and the insertions base call string of the encoding transformation ϕ_t that operate from position $\hat{\imath}$ to j_t are discarded. In Step 12, the modified version of the atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, \hat{\jmath}_t, i'_t, j'_t, \phi'_t)$, is appended to $A(q, \mathcal{R})$, and $\hat{\imath}$ is updated to $\hat{\imath} = i_t$ in Step 14. Notice that, if i_t is greater or equal than $\hat{\imath}$, then $q[i_t : j_t]$ fully overlaps with $A(q, \mathcal{R})$ in which case it is discarded in Step 9 (see example $q[i_4 : j_4]$ in Figure 4). Finally, $A(q, \mathcal{R})$ is returned in Step 17. In Figure 4, the bottom (green) box shows the result of executing Algorithm 10 with the auxiliary full alignment in the top (pink) box as input.

3 Experimental results

In this section we report on experiments performed on a collection of datasets of nanopore FASTQ files. We describe the datasets in Section 3.1. In Section 3.2 we explain how we generate the input PAF files that are necessary to run the proposed compression algorithms. In Section 3.3 we discuss the default values used for various algorithm parameters of RENANO and $\text{RENANO}_{\text{IND}}$. Finally, in Section 3.4 we evaluate the performance of RENANO and $\text{RENANO}_{\text{IND}}$ by comparing them against other compression tools.

To measure the performance of a compressor on a dataset, we compress each file of the dataset separately and calculate the *compression ratio*, CR, defined as $CR = C/T$, where T is the total size in bytes of the files (or file parts) under consideration, and C is the total size in bytes of the corresponding compressed files (or parts). Notice that smaller values of CR correspond to better compression performance. We will use the measure CR in two settings: in the first one, C and T refer to the base call strings part of the FASTQ file only, while in the second one C and T refer to the full FASTQ file. Thus, the first setting focuses only on the parts that are different in ENANO and RENANO, while the second setting reflects the impact on the overall compression of the FASTQ files. To compare compression ratios, we define the *percentage relative difference*, $\frac{CR_2 - CR_1}{CR_1} \times 100$, between the compression ratios CR_1 and CR_2, with respect

to CR_1. Clearly, negative values of this measure indicate that CR_2 is better than CR_1, while positive values indicate that CR_2 is worse. Finally, to compare overall performance between compressors we report *simple* and *weighted* averages of the results over the test datasets, the latter computed by weighting each result by the size of its corresponding dataset. The weighted average is highly influenced by larger datasets and, in particular, by the specific choice of datasets available for evaluation. This can be misleading when assessing the capability of the tools to process different types of datasets. Hence, in the rest of this section we report both averages, but we generally refer to the simple average when discussing and comparing the evaluated tools, unless otherwise specified.

All experiments were conducted on a server with 80 64 bit x86 Intel Xeon CPUs, 503.5GB of RAM memory, and CentOS Linux release 7.7.1908.

3.1 Datasets

We evaluate the proposed algorithms on a collection of publicly available datasets, described in Table 2.

Name	Num. Files	Total size (GB)	Description
hss	1	268	Human GM12878 Utah/Ceph cell line [11]
bra	18	46	Brassica napus L. [20]
sor	4	134	Sorghum bicolor Tx430 [4]
fly	1	17	Drosophila ananassae [21]
yst	6	8	Saccharomyces cerevisiae S288C [10]

Table 2: Nanopore sequencing datasets used for evaluation.

The selected datasets cover a variety of dissimilar organisms including human, plant, animal, and fungi. For each dataset we obtained a reference genome file from the NCBI database [22], from which we extract the reference base call strings used in our algorithms. In Table 3 we present the total size of the read base call strings in each dataset, the total size of the reference base call strings of the genome associated to the dataset, and the coverage of the dataset (total and average per file).

The datasets cover different possible compression scenarios, such as having 268 GB of human data in a single file (hss), with 41x coverage, or having 46 GB of plant data distributed in 18 files (bra), with 1x average coverage per file. Further instructions on how to download the datasets, and the reference genomes, are available at `https://github.com/guilledufort/RENANO`.

Name	Total reads sz. (Mbp)	Reference sz. (Mbp)	Total coverage	Avg. cov. per file
hss	132931	3272	41x	41x
bra	23038	976	24x	1x
sor	66488	709	94x	23x
fly	8363	217	39x	39x
yst	3177	12	265x	53x

Table 3: Read base call string information of the different datasets and their associated reference genomes. The total sizes of the read base call strings and the reference strings are presented in mega base-pairs (Mbp). The total coverage is calculated by dividing the total number of base call symbols in the dataset reads by the number of base call symbols in the strings of the reference genome. The average coverage per file was calculated by dividing the total coverage of the dataset by the corresponding number of files.

3.2 PAF files generation

For our experiments, we generate PAF files with Minimap2 [18], a state of the art sequence alignment tool for long reads. We run Minimap2 with the FASTQ file we want to compress and the proper reference genome file as inputs. In turn, the tool outputs the result of aligning each base call string of the FASTQ file against the reference genome, in PAF format. Specifically, we execute Minimap2 with the following configuration options:

- *-x map-ont*: an option that sets the configuration of the tool to be optimized for reads generated with nanopore technologies. This option is recommended.
- *--cs*: an option that makes the tool perform base-alignment between the aligned sections of the read base call strings and the reference strings. The base-alignment is expressed as a *cs-string*. This option is necessary, as the base-alignment information is needed for our algorithms.
- *--secondary=no*: an option that makes the tool output only primary alignments. This prevents the tool from outputting multiple alignments for the same sections of the read base call strings. This option is recommended.

3.3 Algorithm parameters

In this section we specify the values of the parameters used in our implementations. First, we address the sizes of integers in the streams defined in Section 2.2.1. The following values were used in our experiments, but can eventually be modified as they

are easily re-configurable: $\eta_{\mathcal{L}} = 24, \eta_{\mathcal{Q}} = 8, \eta_{\mathcal{S}} = 24, \eta_{\mathcal{E}} = 24, \eta_{\mathcal{U}} = 16, \eta_{\mathcal{S}'} = 32, \eta_{\mathcal{N}} = 16, \eta_{\mathcal{K}} = 16$, and $\eta_{\mathcal{M}} = 16$.

Furthermore, in Algorithm 9, we introduce the minimum match length threshold, m_L. This parameter determines which match operations in the base-alignment *cs*-strings are interpreted as match operations when transformed into the triplets of encoding transformations described in Section 2.1, and which ones are disregarded and interpreted as an insertion. To determine the default value of m_L, first we evaluate its impact on the compression performance of RENANO, by running the algorithm with different values of m_L, ranging from $m_L = 1$ to $m_L = 6$, on the datasets specified in Table 2. The compression results for the base call strings of each dataset are shown in Table 4.

Dataset	$m_z = 1$	$m_z = 2$	$m_z = 3$	$m_z = 4$	$m_z = 5$	$m_z = 6$
hss	0.118	**0.117**	0.118	0.119	0.120	0.122
bra	**0.141**	**0.141**	0.142	0.143	0.145	0.147
sor	0.161	**0.160**	0.161	0.162	0.164	0.167
fly	0.118	**0.117**	0.118	0.119	0.121	0.123
yst	0.174	**0.173**	**0.173**	0.174	0.175	0.177

Table 4: Reads base call strings compression ratio of RENANO, with different values of minimum match length m_L. Best results for each dataset are bold-faced.

The results show that the best compression performance for every tested dataset is obtained at $m_L = 2$, which we set as the default value in RENANO (the minimum is rather shallow, though, so the precise choice of value is not critical).

3.4 Comparative experimental results

In this section we evaluate RENANO and RENANO$_{\text{IND}}$ by comparing their performance against other compression tools. In Section 3.4.1, we compare against the state of the art nanopore sequencing FASTQ data compressor ENANO [5], on the datasets specified in Table 2. In Section 3.4.2, we also compare against the FASTQ compressor Genozip [15], a recently developed compression tool that can be used for nanopore data and offers both reference-based and reference-free compression modes.

3.4.1 Comparison against ENANO

To evaluate the performance of RENANO and RENANO$_{\text{IND}}$ against ENANO, we run each compressor on the datasets specified in Table 2. Each compressor is configured to run in its default configuration (see Section 3.3 for RENANO and [5] for ENANO).

The read base call strings and total compression ratios obtained on each dataset are shown in Table 5.

Dataset	Read base call strings			Total		
	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$
hss	0.236	**0.117 (-50.3)**	0.123 (-47.8)	0.452	**0.393 (-13.0)**	0.396 (-12.4)
bra	0.238	**0.141 (-40.7)**	0.202 (-15.2)	0.355	**0.307 (-13.6)**	0.337 (-5.1)
sor	0.245	**0.160 (-34.6)**	0.171 (-30.3)	0.374	**0.332 (-11.3)**	0.337 (-9.9)
fly	0.242	**0.117 (-51.5)**	0.123 (-49.0)	0.354	**0.292 (-17.3)**	0.295 (-16.5)
yst	0.236	**0.173 (-26.9)**	0.178 (-24.7)	0.297	**0.266 (-10.4)**	0.269 (-9.6)
S. average	0.239	**0.142 (-40.8)**	0.159 (-33.4)	0.366	**0.318 (-13.1)**	0.327 (-10.7)
W. average	0.239	**0.132 (-44.6)**	0.145 (-39.4)	0.415	**0.362 (-12.7)**	0.368 (-11.1)

Table 5: Read base call strings and total compression ratios (*CR*) for ENANO, RENANO, and $\text{RENANO}_{\text{IND}}$, on all the datasets. The percentage relative difference of RENANO, and $\text{RENANO}_{\text{IND}}$with respect to ENANO are shown in parenthesis. The table also shows the simple (S.) and weighted (W.) *CR* averages. Best results for each dataset are bold-faced.

The results show that both RENANO and $\text{RENANO}_{\text{IND}}$ outperform ENANO for all the datasets. In particular, the best result for each dataset is achieved by RENANO. For read base call strings compression, RENANO shows improvements relative to ENANO ranging from 26.9% (in *yst*) to 51.5% (in *fly*), with an average improvement of 40.8% over all the datasets. As for total compression, the improvements range from 10.4% (in *yst*) to 17.3% (in *fly*), with an average improvement of 13.1%. $\text{RENANO}_{\text{IND}}$ also consistently improves over ENANO. For read base call strings, the improvements range from 15.2% (in *bra*) to 49.0% (in *fly*), while in terms of total size, the improvements range from 5.1% (in *bra*) to 16.5% (in *fly*).

Compared to RENANO, $\text{RENANO}_{\text{IND}}$ achieves similar compression results in datasets with high coverage per file (*hss* 41x, *sor* 23x, *fly* 39x, and *yst* 53x) with a relative percentage deterioration with respect to RENANO ranging from 2.9% (in *yst*) to 6.9% (in *sor*). In the case of dataset *bra*, where the average coverage per file is 1x, the relative percentage deterioration between $\text{RENANO}_{\text{IND}}$ and RENANO reaches 43.3%. This is due to $\text{RENANO}_{\text{IND}}$ directly benefiting from having multiple atomic alignments that use the same sections of the reference strings, which is less likely to happen in files with low coverage. However, even for the dataset *bra*, which has an average coverage per file as low as 1x, $\text{RENANO}_{\text{IND}}$ improves the compression of the read base call strings by 15.6% relative to ENANO, which leads to a total compression improvement of 5.1%.

In Tables 6 and 7 we show the total compression and decompression times (in *h:mm:ss* format) and speeds (in MB/s), respectively, for each algorithm on each

dataset. Both algorithms were run in multi-threading environments, using eight threads in each run.

Dataset	Total compression time (h:mm:ss)			Total decompression time (h:mm:ss)		
	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$
hss	**0:37:09**	0:49:56	0:51:24	0:52:55	**0:49:40**	0:51:19
bra	**0:05:26**	0:11:32	0:08:30	**0:07:40**	0:07:52	0:09:34
sor	**0:13:09**	0:14:51	0:17:34	0:19:45	**0:19:15**	0:20:33
fly	**0:01:41**	0:03:23	0:02:08	0:02:30	**0:02:19**	0:02:26
yst	**0:00:49**	0:01:33	0:01:00	**0:01:05**	**0:01:05**	0:01:06

Table 6: Encoding and decoding times (in *h:mm:ss* format) for ENANO, RENANO, and $\text{RENANO}_{\text{IND}}$, on all the datasets. Best results, for each dataset, both for encoding and decoding, are bold-faced.

Dataset	Compression speed (MB/s)			Decompression speed (MB/s)		
	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$
hss	**120**	90	87	84	**90**	87
bra	**142**	67	91	**101**	98	81
sor	**169**	150	127	113	**116**	108
fly	**168**	83	132	113	**122**	116
yst	**131**	70	107	99	**100**	97
S. average	**146**	92	109	102	**105**	98
W. average	**138**	104	101	95	**99**	94

Table 7: Encoding and decoding speeds in MB/s for ENANO, RENANO, and $\text{RENANO}_{\text{IND}}$, on all the datasets. The table also shows the simple (S.) and weighted (W.) averages of the results. Best results, for each dataset, both for encoding and decoding, are bold-faced.

The results show that ENANO is the fastest compressor for all the datasets, while RENANO is on average the fastest decompressor. Specifically, during compression ENANO was 1.3x and 1.4x times faster on average than RENANO and $\text{RENANO}_{\text{IND}}$, respectively. During decompression, RENANO was 1.05x and 1.1x times faster on average than ENANO and $\text{RENANO}_{\text{IND}}$, respectively.

In table 8 we show the maximum memory required by each compressor during the encoding and decoding processes on all the files of each dataset.

The results show that ENANO is the most efficient in terms of memory for all considered datasets, both for compression and decompression. This is in part due to RENANO and $\text{RENANO}_{\text{IND}}$ having to load a reference genome file into memory

Dataset	Max. compression memory use (GB)			Max. decompression memory use (GB)		
	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$	ENANO	RENANO	$\text{RENANO}_{\text{IND}}$
hss	**0.220**	3.723	10.399	**0.231**	3.441	3.790
bra	**0.210**	1.187	1.539	**0.226**	1.182	0.868
sor	**0.214**	0.943	2.998	**0.227**	0.923	0.933
fly	**0.210**	0.441	1.134	**0.225**	0.439	0.410
yst	**0.205**	0.240	0.345	**0.223**	0.238	0.238
Maximum	**0.220**	3.723	10.399	**0.231**	3.441	3.790

Table 8: Maximum memory usage (in GB) registered during the encoding and decoding processes, for all the compressors, on all the files of each dataset. Lowest memory usage, for each dataset, both for encoding and decoding, are bold-faced.

during the encoding and decoding processes (in the case of $\text{RENANO}_{\text{IND}}$ the artificial reference is loaded during decoding). Also, in the case of $\text{RENANO}_{\text{IND}}$, to generate the artificial reference during encoding, the algorithm needs to load information of each atomic alignment obtained from the PAF file, which makes the memory usage grow with the number of alignments. However, even for the alignment of the *hss* file of 286 GB, the memory usage is of 10.4 GB, which is manageable by a personal computer with 16 GB of RAM.

3.4.2 Comparison against Genozip

We compare RENANO and $\text{RENANO}_{\text{IND}}$ against the tool Genozip [15], which has two modes: a reference-free compression mode (which we refer a to as *Gen*), and a reference-based compression mode (which we refer to as *Gen-ref*). For this comparison we run the selected tools on the nanopore datasets specified in Table 2. We execute both modes of Genozip configured to maximize compression (i.e., in their default configurations). In the case of the reference-based compression mode, we use the reference genome files presented in Table 3. In Table 9 we present the read base call strings, and total, compression ratios obtained by running both modes of Genozip, RENANO, and $\text{RENANO}_{\text{IND}}$, on the selected datasets.

With respect to the results obtained by the two modes of Genozip, the reference-free mode achieved better results than the reference-based mode in all the datasets. Note also that the reference-based mode failed to compress the *hss* and the *sor* datasets. Therefore, to evaluate the performance of RENANO and $\text{RENANO}_{\text{IND}}$ we compare them against the reference-free mode of Genozip (Gen).

The results show that both RENANO and $\text{RENANO}_{\text{IND}}$ significantly outperform Genozip in each of the datasets. Specifically, regarding read base call strings compression, RENANO and $\text{RENANO}_{\text{IND}}$ improve on Genozip by 40.5% and 33.1% on average, respectively, and in total compression by 17.7% and 15.3% on average, re-

File from	Read base call strings				Total			
	Gen	Gen-ref.	RENANO	$\text{RENANO}_{\text{IND}}$	Gen	Gen-ref.	RENANO	$\text{RENANO}_{\text{IND}}$
hss	0.233	-	**0.117 (-49.5)**	0.123 (-46.9)	0.476	-	**0.393 (-17.5)**	0.396 (-16.9)
bra	0.236	0.282	**0.141 (-40.2)**	0.202 (-14.5)	0.370	0.400	**0.307 (-17.2)**	0.337 (-9.0)
sor	0.244	-	**0.161 (-34.0)**	0.171 (-30.1)	0.390	-	**0.332 (-14.8)**	0.337 (-13.6)
fly	0.244	0.282	**0.117 (-51.9)**	0.123 (-49.4)	0.385	0.400	**0.292 (-24.0)**	0.295 (-23.2)
yst	0.236	0.286	**0.173 (-27.0)**	0.178 (-24.8)	0.313	0.339	**0.266 (-14.8)**	0.269 (-14.0)
S. average	0.239	0.283	**0.142 (-40.5)**	0.159 (-33.1)	0.387	0.370	**0.318 (-17.7)**	0.327 (-15.3)
W. average	0.237	0.282	**0.132 (-44.0)**	0.145 (-38.7)	0.436	0.394	**0.362 (-16.9)**	0.368 (-15.4)

Table 9: Read base call strings, and total, compression ratios for both modes of Genozip, RENANO, and $\text{RENANO}_{\text{IND}}$, on all the datasets. The percentage relative difference of RENANO and $\text{RENANO}_{\text{IND}}$ with respect to *Gen* is presented in parenthesis. Best results for each dataset are bold-faced. The table also shows the simple (S.) and weighted (W.) averages of the results. The symbol '-' indicates the tool failed to compress the dataset.

spectively.

References

[1] J. Arram, M. Pflanzer, T. Kaplan, and W. Luk. Fpga acceleration of reference-based compression for genomic data. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 9–16, 2015.

[2] James K. Bonfield and Matthew V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PLOS ONE*, 8(3):1–10, 2013.

[3] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2009.

[4] Stéphane Deschamps, Yun Zhang, Victor Llaca, Liang Ye, Abhijit Sanyal, Matthew King, Gregory May, and Haining Lin. A chromosome-scale assembly of the sorghum genome using nanopore sequencing and optical mapping. *Nature Communications*, 9(1):4844, 2018.

[5] Guillermo Dufort y Álvarez, Gadiel Seroussi, Pablo Smircich, José Sotelo, Idoia Ochoa, and Álvaro Martín. ENANO: Encoder for NANOpore FASTQ files. *Bioinformatics*, 36(16):4506–4507, 2020.

[6] Aníbal Guerra, Jaime Lotero, José Édinson Aedo, and Sebastián Isaza. Tackling the challenges of fastq referential compression. *Bioinformatics and Biology Insights*, 13:1177932218821373, 2019.

[7] Mikel Hernaez, Dmitri Pavlichin, Tsachy Weissman, and Idoia Ochoa. Genomic data compression. *Annual Review of Biomedical Data Science*, 2(1):19–37, 2019.

[8] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res*, 21(5):734–740, 2011.

[9] Zhi-An Huang, Zhenkun Wen, Qingjin Deng, Ying Chu, Yiwen Sun, and Zexuan Zhu. Lw-fqzip 2: a parallelized reference-based compression of fastq files. *BMC Bioinformatics*, 18(1):179, 2017.

[10] Benjamin Istace, Anne Friedrich, Léo d'Agata, Sébastien Faye, Emilie Payen, Odette Beluche, Claudia Caradec, Sabrina Davidas, Corinne Cruaud, Gianni Liti, Arnaud Lemainque, Stefan Engelen, Patrick Wincker, Joseph Schacherer, and Jean-Marc Aury. De novo assembly and population genomic survey of natural yeast isolates with the Oxford Nanopore MinION sequencer. *GigaScience*, 6(2), 2017.

[11] Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, Sunir Malla, Hannah Marriott, Tom Nieto, Justin O'Grady, Hugh E Olsen, Brent S Pedersen, Arang Rhie, Hollian Richardson, Aaron R Quinlan, Terrance P Snutch, Louise Tee, Benedict Paten, Adam M Phillippy, Jared T Simpson, Nicholas J Loman, and Matthew Loose. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36:338 EP –, 2018.

[12] Daniel C. Jones, Walter L. Ruzzo, Xinxia Peng, and Michael G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*, 40(22):e171, 2012.

[13] Carl Kingsford and Rob Patro. Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928, 2015.

[14] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing genomic sequence fragments using SlimGene. *J Comput Biol*, 18(3):401–413, 2011.

[15] Divon Lan, Ray Tobler, Yassine Souilmi, and Bastien Llamas. Genozip - A Universal Extensible Genomic Data Compressor. *Bioinformatics*, 2021.

[16] Eric S. Lander et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[17] Jeremy Leipzig. A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, 18(3):530–536, 2016.

[18] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[19] DJ Lipman and WR Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[20] M. M. Malmberg, G. C. Spangenberg, H. D. Daetwyler, and N. O. I. Cogan. Assessment of low-coverage nanopore long read sequencing for snp genotyping in doubled haploid canola (brassica napus l.). *Scientific Reports*, 9(1):8688, 2019.

[21] Danny E. Miller, Cynthia Staber, Julia Zeitlinger, and R. Scott Hawley. Highly contiguous genome assemblies of 15 drosophila species generated using nanopore sequencing. *G3: Genes, Genomes, Genetics*, 8(10):3131–3141, 2018.

[22] National Center for Biotechnology Information (NCBI). `https://www.ncbi.nlm.nih.gov/`. Accessed: 2020-08.

[23] J. Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664, 1983.

[24] Jorma Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[25] Grégory F. Schneider and Cees Dekker. DNA sequencing with nanopores. *Nature Biotechnology*, 30(4):326–328, 2012.