Bioinformatics
xxxxxx
Advance Access Publication Date: Day Month Year
Original Paper

Sequence analysis

Index suffix-prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression

Yuansheng Liu[1], Zuguo Yu[2,3], Marcel E. Dinger[4,5] and Jinyan Li[1,*]

[1]Advanced Analytics Institute, Faculty of Engineering and IT, University of Technology Sydney, PO Box 123, Broadway, NSW 2007, Australia, [2]Key Laboratory of Intelligent Computing and Information Processing of Ministry of Education, Hunan Key Laboratory for Computation and Simulation in Science and Engineering, Xiangtan University, Hunan 411105, China, [3]School of Electrical Engineering and Computer Science, Queensland University of Technology, GPO Box 2434, Q4001, Australia, [4]Garvan Institute of Medical Research, Sydney, Australia and [5]St Vincents Clinical School, Faculty of Medicine, UNSW, Sydney, Australia.

[*]To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Advanced high-throughput sequencing technologies have produced massive amount of reads data, and algorithms have been specially designed to contract the size of these data sets for efficient storage and transmission. Reordering reads with regard to their positions in *de novo* assembled contigs or in explicit reference sequences has been proven to be one of the most effective reads compression approach. As there is usually no good prior knowledge about the reference sequence, current focus is on the novel construction of *de novo* assembled contigs.
Results: We introduce a new *de novo* compression algorithm named `minicom`. This algorithm uses large k-minimizers to index the reads and subgroup those that have the same minimizer. Within each subgroup, a contig is constructed. Then some pairs of the contigs derived from the subgroups are merged into longer contigs according to a (w, k)-minimizer indexed suffix-prefix overlap similarity between two contigs. This merging process is repeated after the longer contigs are formed until no pair of contigs can be merged. We compare the performance of `minicom` with two reference-based methods and four *de novo* methods on 18 data sets (13 RNA-seq data sets and 5 whole genome sequencing data sets). In the compression of single-end reads, `minicom` obtained the smallest file size for 22 of 34 cases with significant improvement. In the compression of paired-end reads, `minicom` achieved 20-80% compression gain over the best state-of-the-art algorithm. Our method also achieved a 10% size reduction of compressed files in comparison with the best algorithm under the reads-order preserving mode. These excellent performances are mainly attributed to the exploit of the redundancy of the repetitive substrings in the long contigs.
Availability and Implementation: https://github.com/yuansliu/minicom
Contact: Jinyan.Li@uts.edu.au
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

The vast quantity of reads data produced by high-throughput sequencing technologies presents great challenges to data storage and transmission (Koboldt *et al.*, 2010; Goodwin *et al.*, 2016). Traditional text data compression tools, e.g., gzip (`http://www.gzip.org`), bzip2 (`http://www.bzip.org`) and 7zip (`http://www.7zip.org`), have weak performance on these data sets due to their insufficient exploit of the complicated redundancy in the reads (Zhu *et al.*, 2013; Deorowicz and Grabowski, 2013; Wandelt *et al.*, 2014). Over the last decade, specially designed algorithms for raw sequence data compression have been making constant progresses and improvement, and have shown much better compression performance than the general-purpose compression tools (Numanagić *et al.*, 2016).

Raw sequence data are usually stored in the FASTQ format. A FASTQ file contains two main parts: nucleotide sequences (i.e., *reads*) along with their corresponding quality scores indicating the reliability of every bases. Recently, lossy compression studies for the quality scores compression demonstrated that this lossy compression did not affect the downstream analysis on the reads data (Cánovas *et al.*, 2014; Malysa *et al.*, 2015; Yu *et al.*, 2015; Ochoa *et al.*, 2016; Greenfield *et al.*, 2016). Under this context, we focus our study on the *de novo* lossless compression of reads data in FASTQ files.

Our novel *de novo* algorithm (named `minicom`) is designed with two key ideas: minimizer-based indexing for reads and the suffix-prefix overlap between two contigs. The minimizer of a string is the lexicographically smallest k-mer of the string (Roberts *et al.*, 2004). Given a data set, our algorithm determines the k-minimizer for every read and index all of the reads to subgroup those that have the same minimizer. Within each subgroup of the reads, a contig is constructed and then some pairs of the contigs are merged into longer contigs according to our (w, k)-minimizer indexed suffix-prefix overlap between two contigs. This merging process is repeated after the longer contigs are formed until no pair of contigs can be merged. The parameter k can be untraditionally set by our algorithm as a large number to greatly enhance the compression performance.

This approach is quite different from the existing minimizer-based reads compression algorithms `ORCOM` (Grabowski *et al.*, 2014), `Mince` (Patro and Kingsford, 2015) and `FaStore` (Roguski *et al.*, 2018). `ORCOM` and `Mince` take minimizers to bin reads. However, `ORCOM` or `Mince` does not detect our defined suffix-prefix overlaps within any subgroup of the reads to generate long contigs. Although `FaStore` merges neighboring contigs in the re-clustering stage, it is limited that the merged reads cover only a narrow segment of the genome. Our contigs merging approach is more general and is able to cover much wider segments of the genome. Furthermore, small k-mers of the minimizers (e.g., 8, 10 or 15) have to be used by these three algorithms to limit the number of subgroups to obtain low-fidelity matches. However, our `minicom` is allowed to adopt large k-mer (e.g., 31 for the reads of length 100bp) in order to detect long and high-fidelity overlaps (Roberts *et al.*, 2004). We also define (w, k)-minimizers in the detection of effective suffix-prefix overlaps between two contigs. A (w, k)-minimizer is defined as the lexicographically smallest k-mer in the $(w + k - 1)$-long string (Roberts *et al.*, 2004; Li, 2016). The suffix-prefix overlaps can be used to determine an approximate optimal position of reads for improving compression ratio. With the above two key ideas, `minicom` is enabled to exhibit superior performance for the compression of both single-end reads and paired-end reads over the existing reads compression algorithms.

In fact, the existing reads compression algorithms can be classified into two categories: reference-based and *de novo* algorithms. We review them in detail to highlight more about the background and our novelties. The reference-based approach exploits the similarity between the target sequences and the reference sequence. It aligns the target sequences to the reference genome and then stores the position information and base differences for rebuilding the targets. The reference-based approach has demonstrated superior performances for compressing different kinds of sequences such as aligned reads (e.g., BAM-format files) (Bonfield and Mahoney, 2013; Hach *et al.*, 2014), unaligned short reads (Kingsford and Patro, 2015; Zhang *et al.*, 2015) and genomes (Deorowicz and Grabowski, 2011; Liu *et al.*, 2017). Unfortunately, reference-based algorithms have three disadvantages (Patro and Kingsford, 2015): (i) An appropriate reference sequence is not always available; (ii) The reference sequence used for compression must be taken as a copy to the receiver; and (iii) Sequence alignment is a time-consuming procedure.

There are two interesting algorithms different from the traditional reference-based algorithm. Kingsford and Patro (2015) proposed a reference-based compression algorithm `PathEnc`. Though it uses a compressed transcriptome as the shared reference, it does no alignment. The reference is only used to generate a model in a fixed-order context with an adaptive arithmetic coder. `Quark` (Sarkar and Patro, 2017) is a reference-asymmetric compression algorithm, i.e., the reference is only used for compression and not for decompression. The algorithm is designed specially for RNA-seq data compression. After mapping reads to the transcriptome, a set of islands, which is a small subset of the transcriptome, is constructed by merging the reference sequence overlaps of the mappings. The islands are stored in the final compressed file.

On the other hand, *de novo* algorithms compress reads data without use of a reference sequence (Tembe *et al.*, 2010; Jones *et al.*, 2012; Cox *et al.*, 2012). They make use of similarities between the individual reads themselves. The most prevailing idea of *de novo* algorithms is to reorder reads so that reads having a large overlap can be grouped. These algorithms usually consist of three stages: (1) Reordering; (2) Encoding; (3) Compression. Many advanced techniques are employed in the reordering and encoding stages, and the third stage is limited to general-purpose compression tools such as gzip, 7zip and BSC (`https://github.com/IlyaGrebnov/libbsc`).

In the reordering stage, `SCALCE` (Hach *et al.*, 2012) buckets reads according to the longest core substring, which is derived by a locally consistent parsing method. `Mince` (Patro and Kingsford, 2015) improves the bucketing idea of `SCALCE` by a data-dependent bucketing scheme. Each bucket is labeled with a minimizer, which is equivalent to the core substring of `SCALCE`. `ORCOM` (Grabowski *et al.*, 2014) groups reads via the use of signatures, namely carefully selected minimizers. `FaStore` (Roguski *et al.*, 2018) follows the same clustering method of `ORCOM` and a re-clustering method is proposed to merge clusters of high similarity. The basic idea of `HARC` (Chandak *et al.*, 2018) is to find the maximum overlaps between reads so that they can be reordered according to their approximate positions in the genome.

In the encoding stage, `SCALCE` just sorts reads based on the lexicographical order with respect to the position of the core substring. The split-swap read transformation is proposed by `Mince`. It extracts common core substrings and sorts reads according to the offsets of the core substrings. `ORCOM` and `FaStore` sort reads in each disk bin with the lexicographical order of the strings obtained by swapping the left part and right part of the beginning position of the signature for the current reads. `ORCOM` generates several streams by aligning a read to its m previous reads. `FaStore` builds a matching graph by finding the matches in its m previous reads, and traverses each sub-graph to assemble reads into contigs. `HARC` builds a contig for matched reads based on a majority rule at each position. Each matched read is encoded referring to the contig.

Chandak *et al.* (2018) had a theoretic analysis on the reads compression problem from the point of entropy, and proved bounds on the fundamental limits of reads compression. The analysis demonstrates that an algorithm can achieve the best compression ratio if reads can be reordered according to their position in the original genome. If a good reference sequence exists, mapping reads to the reference can decide an optimal order. However, the reference is not available in the *de novo* compression scenario. The intuitive idea is to assemble reads as contig by *de novo* assembly techniques. Traditional *de novo* genome assembly is extreme computationally intensive. The most difficult part is to construct de Bruijn graphs and search Eulerian paths in the graphs. Progresses have been made to resolve the complexity. For example, `Quip` (Jones *et al.*, 2012) and `LEON` (Benoit *et al.*, 2015) both construct probabilistic de Bruijn graphs using the Bloom filter. After that, `Quit` uses a simple greedy approach to assemble contigs. While `LEON` stores the graph instead of producing contigs, where the reads are represented by a k-mer anchor and a list of bifurcation choices. Most recently, Ginart *et al.* (2018) developed a compression algorithm `Assembltrie` via light assembly. `Assembltrie` assembles reads into a compact data structure, called read forest, instead of assembling the reads into independent contigs. It selects potential maximum prefix-suffix overlaps *between reads* greedily in the tries. The ideas for finding longer overlaps by `HARC` and `Assembltrie` are extremely similar. `HARC` uses the overlap information to construct contig, while a read forest is generated and stored by `Assembltrie`. It should be noted that `HARC` and `Assembltrie` are designed for genomic data but not well-tested for compressing RNA-seq data.

We benchmark the performance of `minicom` in comparison with seven state-of-the-art algorithms on various RNA-seq and WGS (whole genome sequencing) data sets. On the single-end reads data sets, `minicom` produces the smallest compressed files on 22 of 34 cases and better than `Quark`, `ORCOM`, `Mince` and `Assembltrie` on all the cases. On the paired-end data sets, `minicom` achieves the best results always with 1.2-1.8 times compression gain for all of the cases. Furthermore, `minicom` achieves 1 GB size reduction from the compressed files by `HARC` in the reads-order preserving mode. The compression time and memory usage of `minicom` are better than that of `PathEnc`, `Mince` and `Assembltrie`.

2 Methods

Let $S = s_1 s_2 \cdots s_n$ be a DNA sequence, where $s_i \in \Sigma = \{\text{A, C, G, T}\}$. Its length is $|S| = n$. For a symbol $a \in \Sigma$, we use $\overline{a}$ to denote its Watson-Crick complement. The reverse complement of S is denoted by $\widehat{S} = \widehat{s}_1 \widehat{s}_2 \cdots \widehat{s}_n = \overline{s}_n \overline{s}_{n-1} \cdots \overline{s}_1$. An encoding function ϕ (i.e., $\phi(\text{A}) = 0, \phi(\text{C}) = 1, \phi(\text{G}) = 2$ and $\phi(\text{T}) = 3$) is used to map a k-mer to a distinct $(2 \times k)$-bit integer. Our method can deal with the letter 'N' in reads, see details in Sec. 2.6.

A *minimizer* of a string is its lexicographically smallest k-mer. If there are more than one such k-mers in a string, the first one is defined as its minimizer. For a read string S, we also consider the minimizer of its reverse complement strand $\widehat{S}$. The pseudocode to compute minimizers is described in Supplementary Algorithm 1. A list of (w, k)-minimizers from a string can be derived from the all possible $(w + k - 1)$-long substrings shifting from the beginning of the string to the end. We use Supplementary Algorithm 2 to compute the first τ minimizers in such a list. The concept of minimizers was initially proposed by Roberts *et al.* (2004) to reduce memory consumption and processing time for biological sequence comparison. Here we use minimizers to index reads. As observed by (Li, 2016), an invertible hashing function can be used to perform a random ordering of the k-mers instead of a lexicographic ordering, for better performance (Marçais *et al.*, 2017).

2.1 Reads indexing and iterative contigs merging

Our *de novo* assembled long contigs are generated with two stages: the initial basic contigs generation and the subsequent contigs merging (in an iterative way).

Initial basic contigs: First, we compute the minimizers for all of the reads. Large k-mers are used in our minimizers in expect to obtain long and high-fidelity overlaps between two contigs. Reads are clustered (sub-grouped) by hashing their minimizers to a hash table. The key of the hash table is the minimizer and the value of the hash table is a set of the read index, the position of the minimizer in the read and the strand label (0 representing the read itself or 1 representing its reverse complement strand). The value of a minimizer can be very large for a large k-mer. For example, the value of the minimizer is a 62-bit integer if $k = 31$. In implementation, we limit the entries of the hash table to reduce the memory usage of the hash table. If there is a collision to hash minimizers to the limited entries, we sort the entry array independently according to the minimizers after collecting all the minimizers. Therefore, we can easily distinguish different clusters by comparing two contiguous minimizers in the entry. If there are more than one reads in one cluster, we sort the reads in this cluster through the position of the minimizers. Then, a contig is constructed. The details are described in Algorithm 1. As an example, a reads cluster from dataset SRR490961 is shown in Supplementary Figure 1.

Each base of the contig for the cluster of reads is the base with the highest frequency. Once the contig is constructed, we compute the Hamming distance of every read in this cluster by aligning the read to the contig without indels. If the Hamming distance exceeds a pre-defined

Algorithm 1: Generation of the initial basic contigs

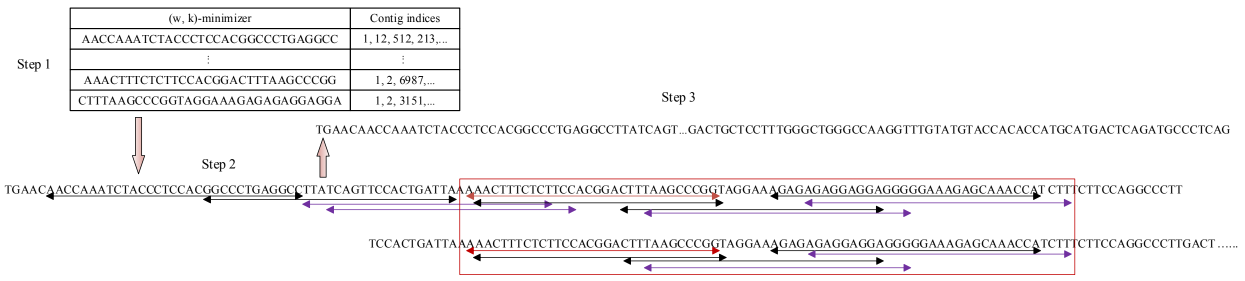
```
Input: Set of reads $\mathcal{S} = \{S_1, \cdots, S_n\}$, size of $k$-mer $k$, size of
       hash table $b$ and difference threshold $e$
Output: A set of contigs

Function InitialContigsSketch($S, k, b, e$) begin
    $\mathcal{H}[1..b] \leftarrow$ empty hash table    ▷ Each entry $\mathcal{H}[i]$ is an array;
    Each element of $\mathcal{H}[i]$ is a tuple composed of minimizer, position
    of minimizer, strand label and reads index
    for $t \leftarrow 1$ to $n$ do
        $(h, p, r) \leftarrow$ MinimizerSketch$(S_t, k)$
        Append $(h, p, r, t)$ to $\mathcal{H}[h \% b]$
    $\mathcal{W} \leftarrow \emptyset$    ▷ A set of reads clusters
    for $e \leftarrow 1$ to $b$ do
        Sort $\mathcal{H}[e] = [(h, p, r, t)]$ by the 1st item $h$
        $i \leftarrow 1$
        for $j \leftarrow 1$ to $|\mathcal{H}[e]|$ do
            if $j = |\mathcal{H}[e]|$ or $\mathcal{H}[e][j].h \neq \mathcal{H}[e][j+1].h$ then
                if $j - i > 1$ then    ▷ More than one reads
                    $\mathcal{G} \leftarrow$ empty array
                    for $c \leftarrow i$ to $j$ do
                        Append $(\mathcal{H}[c].p, \mathcal{H}[c].r, \mathcal{H}[c].t)$ to $\mathcal{G}$
                    Sort $\mathcal{G} = [(p, r, t)]$ by the 1st item $p$ of tuples
                    $(R, \mathcal{F}) \leftarrow$ ConstructContigSketch$(\mathcal{S}, \mathcal{G}, k, e)$
                    if $|\mathcal{F}| > 1$ then
                        $\mathcal{W} \leftarrow \mathcal{W} \cup \{(R, \mathcal{F})\}$
                $i \leftarrow j + 1$
    return $\mathcal{W}$
```

threshold e, this read is removed from the cluster. Finally, we determine the alignment position of every read in each cluster and record their strand label. Supplementary Algorithm 3 details the procedure to construct the contig for a cluster of reads.

Actually, a minimizer is a sampling presentation of DNA sequence. It is very sensitive to k-mer size. An example involving two reads, which have the same 30-minimizer but different 31-minimizer, from dataset SRR1294116 is shown in Supplementary Figure S2. To cluster more reads, the remaining reads are then processed by the above procedure with some smaller k-mer sizes.

Contigs merging via suffix-prefix overlaps between two contigs: The length of the basic contigs ranges from L to $(2 \cdot L - k)$, where L stands for the length of reads. To construct longer contigs, we merge these basic contigs. Given a contig, we find a contig such that its prefix is approximately matched with the suffix of the given contig. By "approximately matched", we mean that the Hamming distance between the prefix and suffix is smaller than a threshold. To index the prefixes of the basic contigs, we compute the first τ number of (w, k)-minimizers of each contig and insert these minimizers to a hash table. By default, we set $w = L/2 - k$ to ensure the length of the overlap is at least $L/2$. Strategies (Li, 2016) such as, (i) limit the entries of hash table; (ii) sort each entry; and (iii) store the intervals of minimizers on the sorted array, are used to reduce the memory consumption and fast indexing. For each contig, we enumerate all (w, k)-minimizers and search contigs sharing the same minimizers in the hash table. Given a minimizer of this contig, a next contig is detected if the two (w, k)-minimizers have the same strand and the Hamming distance of the overlap substring is smaller than λ. By default $\lambda = 2 \cdot e$. We concatenate these two contigs to form a new contig. Then, reads corresponding to these two contigs are merged into a new cluster and we subsequently update their alignment position against this long contig.



(w, k)-minimizer	Contig indices
AACCAAATCTACCCTCCACGGCCCTGAGGCC	1, 12, 512, 213,...
⋮	⋮
AAACTTTCTCTTCCACGGACTTTAAGCCCGG	1, 2, 6987,...
CTTTAAGCCCGGTAGGAAAGAGAGAGGAGGA	1, 2, 3151,...

Step 1
Step 2
Step 3
TGAACAACCAAATCTACCCTCCACGGCCCTGAGGCCTTATCAGT...GACTGCTCCTTTGGGCTGGGCCAAGGTTTGTATGTACCACACCATGCATGACTCAGATGCCCTCAG
TGAACAACCAAATCTACCCTCCACGGCCCTGAGGCCTTATCAGTTCCACTGATTAAAAACTTTCTCTTCCACGGACTTTAAGCCCGGTAGGAAAGAGAGAGGAGGAGGGGGAAAGAGCAAACCAT CTTTCTTCCAGGCCCTT
TCCACTGATTAAAAACTTTCTCTTCCACGGACTTTAAGCCCGGTAGGAAAGAGAGAGGAGGAGGGGGAAAGAGCAAACCATCTTTCTTCCAGGCCCTTGACT......

Fig. 1. An illustration of contigs merging via suffix-prefix overlaps. The double-ended arrows represent $(19, 31)$-minimizers. Step 1: The first τ (w, k)-minimizers of each contig are indexed into a hash table. Step 2: For a contig, we enumerate its all $(19, 31)$-minimizers and search for the same minimizers in the hash table to find a suffix-prefix overlap. When processing the fifth $(19, 31)$-minimizer, we find the same $(19, 31)$-minimizer in the second contig. Both of them are highlighted with a red line. A suffix-prefix overlap, which contains 98 bases, is determined by the $(19, 31)$-minimizer. Step 3: Merging two contigs to form a new long contig.

An illustration of searching suffix-prefix overlap is shown in Fig. 1. The details of this contig merging process are presented in Algorithm 2.

Algorithm 2 merges two contigs only. We repeat this procedure many rounds to merge more contigs, including those merged in the previous rounds. The iteration stops when the number of merged contigs is smaller than 100.

Our minicom is quite different from the key idea of HARC: (i) Our initial basic contigs generation always groups duplicate reads into the same cluster, but HARC is unable to guarantee this because it only searches limited number of entries of the hash table; (ii) Our contigs merging procedure makes use of the suffix-prefix overlaps between contigs, but HARC finds the next read via suffix-prefix overlap between *reads*.

2.2 Realignment of singleton reads

It is possible that there can exist reads which are not covered by any cluster (remaining as singletons) after the above contig generation stage. To move some of these singleton reads into some clusters, we increase the difference threshold e to align more reads to the contigs, as similarly handled by HARC (Chandak *et al.*, 2018). In detail, these singleton reads are indexed into μ different hash tables by the substrings (length is by default set to 17 for reads length >80bp). Given a difference threshold, we enumerate all substrings of a contig and align the singleton reads to the contig by searching the hash tables. For a potential match, we apply Supplementary Algorithm 4 to compute the final difference string. If $|E| \leq 0.4 \cdot L$, we then group this singleton read to the cluster. To find more accurate alignment, we increase the difference threshold by a step size η (default as e) each round. We set the maximum difference threshold as $L/2$.

2.3 Encoding

For each cluster of reads, we store its contig sequence, the alignment position of every read referring to the contig, the strand label and the difference string between the read and the contig. We first sort this cluster of reads through their alignment positions into an ascending order. The alignment positions are further encoded by delta encoding. We use 1-bit to record whether the read is encoded in the original or its reverse complement strand. If reads have the same alignment position, we sort them through their strand labels. The labels 0 or 1 can be arranged more compactly, leading to a smaller size of the final compressed file. The alignment positions and strand labels are written in a binary file separately. We employ 2-bit encoding to encode the contig sequence and store as a binary file as well. The difference file consists of the matched lengths and mismatched characters between the reads and the contig (an ASCII text file). All of these files are compressed independently using the BSC compressor. Supplementary Algorithm 4 presents the details of the difference encoding.

Algorithm 2: Merge contigs

```
Input: Set of contigs $\mathcal{W}$, window size $w$, size of $k$-mer $k$, number
       of minimizer $\tau$, size of hash table $b$ and threshold $\lambda$
Output: A set of contigs
Function MergeContigSketch($\mathcal{W}, w, k, \tau, b, \lambda$) begin
    $\mathcal{H}[1..b] \leftarrow$ empty hash table        ▷ Each entry $\mathcal{H}[i]$ is an array
    foreach $(R_t, \mathcal{F}_t) \in \mathcal{W}$ do
        $\mathcal{M} \leftarrow$ MinimizersSketch$(R_t, w, k, \tau)$
        foreach $(h, p, r) \in \mathcal{M}$ do
            Append $(h, p, r, t)$ to $\mathcal{H}[h\%b]$
    for $e \leftarrow 1$ to $b$ do
        Sort $\mathcal{H}[e]$ by the 1st item
    $(\mathcal{W}', v_i) \leftarrow (\emptyset, 0)$        ▷ $v$ is an array
1   for $i \leftarrow 1$ to $|\mathcal{W}|$ do
        if $v_i = 0$ then        ▷ i-th contig is not visited
            $\mathcal{M} \leftarrow$ MinimizersSketch$(R_i, w, k, \infty)$
            foreach $(h, p, r) \in \mathcal{M}$ do
                foreach $(h', p', r', t) \in \mathcal{H}[h\%b]$ do
                    if $i \neq t$ then        ▷ exclude itself
                        if $h = h'$ and $v_t = 0$ and $r = r'$ then
                            $\xi \leftarrow$ the Hamming distance of the overlap
                              substring of two contigs $R_i$ and $R_t$
                            if $\xi <= \lambda$ then
                                Merge $R_i$ and $R_t$ to a long contig $R'$
                                $\mathcal{F}' = \mathcal{F}_i \cup \mathcal{F}_t$
                                Update the 1st item of $\mathcal{F}' = [(p, r, t)]$
                                $\mathcal{W}' = \mathcal{W}' \cup \{(R', \mathcal{F}')\}$
                                $(v_i, v_t) \leftarrow (1, 1)$        ▷ Label visited
                                goto 1
    return $\mathcal{W}'$
```

2.4 Reads-order preserving mode

An optional requirement for reads compression algorithms is to maintain the original order of the reads after decoding. It is called reads-order preserving mode. Similarly as proposed by HARC (Chandak *et al.*, 2018), we additionally store the order information of the reads in the FASTQ file along with the above information. As a FASTQ file usually contains more than 10 million reads, the file storing the order information is a big volume in the compressed file. To reduce the size of the position files, we sort reads in each cluster according to the position of reads if the reads have the same alignment position. Delta encoding is applied to encode the positions of reads having the same alignment position. Finally, the positions are encoded as a binary file and it is then compressed by BSC as well.

2.5 Handling paired-end reads

Paired-end reads contain long range positional information, keeping this information while compressing is compulsory. `PathEnc` (Kingsford and Patro, 2015) and `Mince` (Patro and Kingsford, 2015) handle paired-end reads by merging the two ends of the pair. To keep the same strand, if necessary, the two methods can generate a reverse complement for one of the ends before merging. Our method `minicom` can preserve the pairing information by recording a permutation rather than concatenating the two ends. In detail, our method generates contigs, realign singleton reads and encode as single reads. Comparing with `PathEnc` and `Mince`, we create two more files to recover the paired-end information. `Minicom` uses 1-bit to record whether a read is from the left or right end. If a read is from the left end, we record the permutation mapping before and after reordering. Later, we only save the permuted position of the reads from the right end. The two files are encoded as binary files and compressed by BSC and 7zip respectively.

2.6 Other considerations

There are special subtypes of reads. For example, all bases in a read are ‘A’, all bases are ‘T’ or all bases are ‘N’. From the right end of SRR635193, we have observed 1324, 60 and 1085118 reads respectively for these special subtypes. For these special reads, we only store the number of reads. Another three subtypes of reads are: most bases in a read are ‘A’, most bases are ‘T’ and most bases are ‘N’. We set a threshold (length of reads minus difference threshold e) to detect these three subtypes of reads. Supplementary Algorithm 3 is used to encode these subtypes of reads by replacing the contig with the sequence of pure corresponding bases. In the reads-order preserving mode, these six subtypes of special reads are sorted according to their position of reads in the FASTQ file and then delta encoding is used to encode the position values. If there are many such special reads, this technique is effective to achieve the highest compression ratio for these reads. Otherwise, only 4 bytes are wasted for each subtype.

To simplify reads processing, we classify reads containing ‘N’ into two kinds: the number of ‘N’ small than σ and others. If reads in the first kind, we first convert ‘N’ to the most frequency bases in the corresponding reads. Then, those reads are used to generate contigs as usual. Before encoding, those changed bases are translated to ‘N’ again. The second kind contains reads having many ‘N’s. We store these reads as an ASCII text file and compress it by BSC.

We had parallel implementation for our algorithm. The initial contigs generation is paralleled by the separate processing of the entries of the hash table. After that, all procedures are paralleled by the separate processing of the clusters.

3 Results and analysis

We tested the proposed algorithm on various real reads data. The performance was compared with two recently published reference-based algorithms `PathEnc` (Kingsford and Patro, 2015) and `Quark` (Sarkar and Patro, 2017), and with four *de novo* compression algorithms `Mince` (Patro and Kingsford, 2015), `ORCOM` (Grabowski *et al.*, 2014), `HARC` (Chandak *et al.*, 2018) and `Assembltrie` (Ginart *et al.*, 2018). All the experiments were carried out on a computing cluster running Red Hat Enterprise Linux 6.7 (64 bit) with 2×2.33 GHz Intel® Xeon® E5-2695 v3 (14 Cores) and 128 GB RAM. All algorithms were run with 24 threads under their default/recommended parameters.

3.1 Datesets

A total 18 sequencing data sets, including 13 RNA-seq data sets and 5 whole genome sequencing (WGS) data sets, are used in this work to benchmark the performance of the seven algorithms. 12 data sets of them are single-end reads data and 6 of them are paired-end reads data. The read lengths of these data sets are various, ranging from 44 to 108. Some of these RNA-seq data sets are benchmark data sets widely used in the literature (Kingsford and Patro, 2015; Patro and Kingsford, 2015; Sarkar and Patro, 2017); the other RNA-seq data sets have never been tested by the state-of-the-art methods. The WGS data sets are compiled by the MPEG HTS working group (Numanagić *et al.*, 2016) for benchmarking. It covers a wide range of organisms (human metagenomic, bacterial and plant genome) and coverage. Details of these data sets are provided in Supplementary Table S1.

3.2 Compression performance

To test the robustness of these algorithms, the two end files of paired-end reads are compressed independently as per HARC (Chandak *et al.*, 2018) and their concatenated version is tested as well. In total, 34 single FASTQ files are compressed and the sizes of these compressed files are presented in Tab. 1. Our method `minicom` achieves the best compression result for 22 of the 34 cases and obtains the smallest total size. On the remaining 12 cases, our `minicom` has very close performance (always the second best) to the best compression result. In particular, `minicom` performs better than `Quark`, `ORCOM`, `Mince` and `Assembltrie` for all of the 34 cases. The compressed files by the two reference-based algorithms `PathEnc` and `Quark` are 39% and 15% larger than ours respectively. Seven cases by `PathEnc` are even twice larger than ours. Though `PathEnc` wins on two cases as the best algorithm, our `minicom` achieves very close results within 5 MB in total. In comparison with the four *de novo* algorithms, the size of `minicom`’s compressed files is about 21% smaller than that of `ORCOM` and is 25% smaller than that of `Mince`. `HARC` is more competitive to `minicom` and its total size of compressed files is about 519 MB larger than ours. `HARC` wins on 10 cases as the best algorithm on the WGS data sets. `Minicom` is better than `HARC` on all of the RNA-seq data sets and wins 3 times on the WGS data sets. All of the compressed files by `Assembltrie` are at least 18% and up to 66% (or on average 35%) larger than ours.

There are many features of reads which can affect the compression ratio, such as the reads length, sequencing depth, quality of reads and duplication rate. Table 1 (the first 21 rows) indicates that our method `minicom` can make better compression rate on RNA-seq data sets than on WGS data sets (see last 13 rows). The reason is probably that RNA-seq data and WGS data have different characteristics in reads. For example, about 34% of reads in the RNA-seq data set SRR445724 are duplicate. While only 4.5% reads are duplicate in the WGS data set SRR174310_1. We did an experiment which removed duplicate reads from four data sets to understand the performance before and after removing the duplicate reads. Results are shown in Supplementary Table S4. The change is significant—the compression ratio decreases by 28% and 33% after the removal of duplicated reads on the RNA-seq data sets SRR490976 and SRR445724 respectively.

Ginart *et al.* (2018) have observed that more than 40% of the reads in SRR870667_1 (reads length is 108) do not share an overlap (length > 21) with any other reads even the Hamming distance is set as 4. In such a case, it is difficult to generate long contigs with high-fidelity overlaps. After testing different parameters, we found that $e = 18$ achieves the best compression ratio. A larger threshold can result in more reads grouped in clusters. But, this would cause wider difference between the reads and the contigs. On this data set, the compressed files by `HARC` and `Assembltrie` are > 1.9 times larger than ours. The length of reads in SRR870667_2 is 74 which is

Table 1. Sizes (in byte) of the compressed files in the compression of single-end FASTQ files

Dataset	Reference-based methods		*de novo* methods				
	`PathEnc`	`Quark`	`ORCOM`	`Mince`	`HARC`	`Assembltrie`	`minicom`
SRR1294116	185,505,576	185,502,660	203,910,373	199,944,420	165,560,320	244,190,064	**156,446,720**
SRR1294122	195,043,844	184,186,666	212,361,271	204,841,681	172,175,360	248,586,788	**159,528,960**
SRR490961	176,903,711	160,646,768	177,165,498	174,581,268	133,068,800	210,139,010	**122,552,320**
SRR490962	161,331,188	143,681,017	158,777,607	156,291,137	118,056,960	190,222,802	**109,219,840**
SRR490976	188,202,449	167,241,067	195,389,895	173,776,052	158,044,160	235,760,055	**146,636,800**
SRR445718	159,916,858	148,637,597	172,040,669	162,963,826	138,792,960	193,122,455	**126,064,640**
SRR445719	152,110,548	140,260,951	163,741,508	156,071,908	130,938,880	180,481,550	**119,541,760**
SRR445724	262,188,846	273,093,903	296,775,789	287,989,612	272,742,400	366,182,608	**245,585,920**
SRR445726	237,646,946	243,123,204	263,623,124	256,387,742	238,776,320	329,297,287	**215,572,480**
SRR635193_1	**49,733,548**	57,670,400	69,152,406	67,964,900	53,616,640	88,363,940	53,125,120
SRR635193_2	**57,893,202**	64,642,313	77,003,657	74,360,498	60,395,520	275,101,281	58,716,160
SRR635193*	93,612,837	100,437,552	116,015,715	115,342,258	90,880,000	171,342,552	**88,647,680**
SRR689233_1	59,686,615	55,076,392	54,028,762	51,396,163	36,720,640	56,197,400	**33,669,120**
SRR689233_2	67,730,999	61,475,515	62,384,804	57,565,475	44,451,840	66,507,677	**40,417,280**
SRR689233*	104,302,169	95,806,542	95,638,561	89,548,209	66,549,760	113,831,509	**60,395,520**
SRR1265495_1	72,587,827	74,772,026	87,770,548	87,717,853	118,824,960	99,009,575	**67,829,760**
SRR1265495_2	73,357,330	75,338,821	88,706,224	87,163,316	115,548,160	92,204,870	**68,659,200**
SRR1265495*	126,828,480	119,668,224	137,793,482	137,274,758	165,734,400	162,352,256	**101,775,360**
SRR1265496_1	65,790,698	70,430,526	79,936,259	79,893,390	108,523,520	82,211,910	**65,105,920**
SRR1265496_2	70,510,753	73,397,947	85,417,582	83,697,854	110,612,480	89,349,347	**67,676,160**
SRR1265496*	116,842,428	113,010,822	127,395,119	126,830,440	149,596,160	140,667,778	**98,119,680**
SRR554369_1	13,482,645	—	10,731,797	10,038,174	**5,652,480**	7,657,499	5,969,920
SRR554369_2	14,013,314	—	11,180,413	10,571,589	**6,021,120**	8,205,937	6,379,520
SRR554369*	20,150,328	—	15,859,984	15,353,447	**8,294,400**	12,158,577	8,683,520
SRR327342_1	45,917,884	—	35,992,863	36,892,806	**18,780,160**	33,059,162	19,353,600
SRR327342_2	58,336,493	—	49,474,721	49,324,493	29,655,040	47,214,209	**27,770,880**
MH0001.081026_1	55,169,938	—	51,469,749	50,345,988	**41,789,440**	54,640,426	43,161,600
MH0001.081026_2	60,521,876	—	57,032,946	55,881,683	46,848,000	59,224,619	**46,540,800**
MH0001.081026*	99,662,933	—	92,093,916	92,053,436	**74,516,480**	100,107,376	75,479,040
SRR870667_1	843,131,165	—	826,261,168	687,536,058	1,372,682,240	1,703,038,318	**686,796,800**
SRR870667_2	447,501,976	—	315,154,162	349,309,795	**222,197,760**	715,207,920	236,482,560
ERR174310_1	3,102,765,421	—	1,797,748,923	1,954,851,290	**1,399,818,240**	1,837,682,723	1,497,856,000
ERR174310_2	3,131,250,155	—	1,830,332,018	1,985,838,750	**1,456,107,520**	1,900,332,740	1,542,533,120
ERR174310*	×	—	2,536,425,475	2,992,003,750	**1,500,846,080**	×	1,911,060,480
Total size	—	—	10,554,786,988	11,121,604,019	8,832,819,200	—	8,313,354,240

Notes: Bold font indicates the best result in the row. A shadowed text indicates that our method achieves the second best result. A '*' indicates these files are obtained by concatenating two corresponding FASTQ files. A '×' means that the method cannot compress it using the limited RAM. `Quark` was not tested on the WGS data as it is specially designed for the compression of RNA-seq data.

Table 2. Size (in byte) of compressed files for paired-end reads

Dataset	Reference-based methods		*de novo* methods				
	`PathEnc`	`Quark`	`ORCOM`	`Mince`	`HARC`	`Assembltrie`	`minicom`
SRR635193	222,826,708	151,368,409?	410,021,128	240,283,883	263,720,960	410,673,387	**154,112,000**
SRR689233	180,116,872	191,811,478	423,041,099	176,948,801	168,407,040	401,463,383	**93,296,640**
SRR554369	25,839,093	—	70,448,471	31,101,469	17,397,760	54,376,622	**11,601,920**
MH0001.081026	143,359,035	—	226,753,969	174,843,659	144,189,440	201,121,008	**113,479,680**
ERR174310	×	—	8,940,636,785	6,058,152,443	2,998,538,240	7,711,063,859	**2,508,769,280**
Total size	—	—	10,070,901,452	6,681,330,255	3,592,253,440	8,778,698,259	2,881,259,520

Notes: Bold font indicates the best result in the row. A '?' indicates the decompressed reads are not identical to the original (see Supplementary Figure S3).

31% smaller than that of SRR870667_1. Supplementary Table S5 lists the size of different parts of compressed files about these two data sets. We can see that the difference file of SRR870667_1 is $>$ 6 times larger than that of SRR870667_2. These observations probably partly explain why the seven methods achieved very different compression ratios on these two data sets.

Comparison results of the seven algorithms on the five paired-end reads data sets are shown in Tab. 2. `Minicom` achieves significant compression improvement over the existing *de novo* methods. The total size of compressed files by `ORCOM`, `Mince` and `Assembltrie` are at least twice larger than ours. Comparing with `ORCOM`, our `minicom` obtains compression gain varying from 2.0 to 6.0 times. `Minicom` achieves a compression gain from 1.5 to 2.6 times in comparison with `Mince`. The improvement over `Assembltrie` by `minicom` is ranging from 1.7 to 4.3 times. The size of `minicom`'s compressed files is about 710 MB smaller than that of `HARC`. `Minicom` achieves 1.2-1.8 times better compression than `HARC`. Compared with the two reference-based methods, `minicom` also achieves compression gain >1.2 times except one case by `Quark`. However, we note that the decompressed result by `Quark` on the paired-end data set SRR635193 is not identical to the original data set. `Quark` is specially designed only for RNA-seq dataset, not effective for WGS data sets.

Of the seven algorithms, only our method `minicom` and `HARC` are capable of preserving the reads order. The comparison results are shown in Tab. 3. Our `minicom` achieves superior compression performance to `HARC` on 25 of the 30 cases (sometimes with >1.4 times compression gain). On the remaining 5 cases, the compressed file by `minicom` is <5% larger than that of `HARC`. The compressed files by `HARC` are 1 GB larger than ours in total.

Moreover, we compared the performance with a new tool named `Spring` (`https://github.com/shubhamchandak94/SPRING`). It should be pointed out that `Spring` is a revised version of `HARC`. As `Spring` has not been formally published in the scientific literature to date, we cannot understand its technical details and cannot do in-depth comparison with it. We only run the tool to compare its performance with `HARC` and our method (see Supplementary Tables S2 and S3). In the compression of single-end reads, our method `minicom` is better than `Spring` on 20 cases. `Spring` wins 13 times—most of them are on the WGS data sets. On the RNA-seq data sets, `minicom` performs better than `Spring` on 18 of the 21 cases, and `Spring` is worse than `HARC` on 12 cases. On the WGS data sets, `minicom` is better than `Spring` on 2 cases. In the compression of paired-end reads, `Spring` is better than `HARC` and `minicom` on 4 cases and `minicom` wins on one case. For the reads-order preserving mode, `minicom` performs better than `Spring` on 21 of the 29 cases.

We found that `FaStore` (Roguski *et al.*, 2018) is a tool for compressing the whole FASTQ file (all identifiers, all reads and all quality scores), there is no option specially set to ignore the quality scores. We used the compression mode '–max', which applies a Q-score binary thresholding and ignores read identifiers, to run the tool. This compression performance is shown in Supplementary Table S9. We note that this performance cannot be used to compare with `minicom` directly.

3.3 Comparison on computational resources

`ORCOM` is the fastest algorithm for the compression of these data sets. `Assembltrie` had unstable compression speeds from a few minutes to 43 hours. `Minicom` is always faster than `PathEnc`. In most cases, `minicom` is significantly faster than `PathEnc` and `Assembltrie`. `Minicom` is faster than `Quark` except from other 6 cases. For 29 of the 39 cases, `minicom` is faster than `Mince` and sometimes twice faster. In most cases, `minicom` is a little slower than `HARC`. Detailed comparison of the compression time costs by these algorithms is shown in Supplementary Table S6.

Supplementary Table S7 describes the memory usage of these algorithms in compression. `HARC` uses the least RAM for all of the cases. The RAM consumption by `minicom` is much less than `PathEnc` and `Assembltrie`. `Minicom` takes less RAM than `Quark` and `Mince` as well. As `minicom` maintains all the processed reads and assembled contigs in memory, it can be well understood that more memory is required than by `ORCOM` or `HARC`.

The comparison of decoding time is presented in Supplementary Table S8. `ORCOM` is much faster than all the other methods. The decompression speed of `minicom` is at least one order of magnitude faster than `PathEnc`; about twice faster than `Quark`; and about 2.5 times faster than `mince`. Though `HARC` achieves competitive decompression speed as ours in most cases, its total decompression time cost is 1.8 times more. `Assembltrie` performs a little bit faster than our methods on some cases. However, `Assembltrie` performs extremely poor on 6 cases.

Table 3. Sizes (in byte) of compressed files in the reads-order preserving mode

Dataset	`HARC`	`minicom`
SRR1294116	$312,115,200$	$\mathbf{291,491,840}$
SRR1294122	$297,256,960$	$\mathbf{273,018,880}$
SRR490961	$291,010,560$	$\mathbf{270,100,480}$
SRR490962	$264,325,120$	$\mathbf{245,923,840}$
SRR490976	$259,246,080$	$\mathbf{235,571,200}$
SRR445718	$239,636,480$	$\mathbf{220,538,880}$
SRR445719	$224,778,240$	$\mathbf{207,462,400}$
SRR445724	$430,469,120$	$\mathbf{386,488,320}$
SRR445726	$391,946,240$	$\mathbf{351,764,480}$
SRR635193_1	$138,485,760$	$\mathbf{129,259,520}$
SRR635193_2	$142,141,440$	$\mathbf{132,864,000}$
SRR689233_1	$85,544,960$	$\mathbf{80,721,920}$
SRR689233_2	$92,395,520$	$\mathbf{87,080,960}$
SRR1265495_1	$146,995,200$	$\mathbf{95,272,960}$
SRR1265495_2	$151,562,240$	$\mathbf{96,020,480}$
SRR1265495	$244,736,000$	$\mathbf{155,965,440}$
SRR1265496_1	$127,324,160$	$\mathbf{89,374,720}$
SRR1265496_2	$129,536,000$	$\mathbf{91,944,960}$
SRR1265496	$217,569,280$	$\mathbf{146,759,680}$
SRR554369_1	$\mathbf{10,004,480}$	$10,178,560$
SRR554369_2	$\mathbf{10,373,120}$	$10,547,200$
SRR327342_1	$64,143,360$	$\mathbf{63,395,840}$
SRR327342_2	$74,629,120$	$\mathbf{71,505,920}$
MH0001.081026_1	$73,134,080$	$\mathbf{71,178,240}$
MH0001.081026_2	$77,056,000$	$\mathbf{76,308,480}$
SRR870667_1	$1,540,474,880$	$\mathbf{906,864,640}$
SRR870667_2	$\mathbf{454,359,040}$	$460,574,720$
ERR174310_1	$\mathbf{2,105,876,480}$	$2,209,443,840$
ERR174310_2	$\mathbf{2,158,960,640}$	$2,246,676,480$
Total size	$10,756,085,760$	$9,714,298,880$

4 Conclusion and future work

We have introduced `minicom`, a new *de novo* algorithm for reads data compression. The idea of (w, k)-minimizers is the first time used for reads compression by our `minicom`. It is an effective idea to find a suffix-prefix overlap between two contigs and to determine a sub-optimal order of reads. The experiment results obtained from various benchmark data sets confirm that `minicom` can provide better compression performance than both the state-of-the-art reference-based and *de novo* compression algorithms.

Although `minicom` has already exhibited excellent compression performance, there are some other facets worth of investigation for further improvement. First, its performance heavily depends on some parameters such as the length of k-mer, the difference threshold e. The current default setting of these parameters is assessed through the performance analysis on some selected data sets. We will establish a robust and automated parameter selection procedure (e.g., through feature extraction from reads) in near future. Second, the realignment procedure for the singleton reads takes up a large proportion of the total compression time cost. We are going to work out a more time-efficient realignment procedure to accelerate the compression process. Third, a disk-based mode, like ORCOM, for contigs generation can be added to reduce the RAM usage, convenient for a standard PC. Fourth, we will develop a more practical compression tool for FASTQ format file by incorporating lossy compression of the quality scores. Last, we will investigate if the minimizer-indexed iterative contigs generation can be used to speed up *de novo* genome or transcriptome assembly and reads alignment.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of our manuscript. We thank Prof. Carl Kingsford for providing the reference transcripts. We are grateful to Prof. Rob Patro and Dr. Hirak Sarkar for running Quark. We thank Prof. Sebastian Deorowicz and Dr. Łukasz Roguski for running FaStore. We are grateful to Hui Peng and Chaowang Lan for their insightful discussion.

Funding

This project was supported by the Australia Research Council (ARC) Discovery Project DP180100120, and partly supported by the National Natural Science Foundation of China (Grant No. 11871061) and Collaborative research project for Overseas Scholars (including Hong Kong and Macau) of National Natural Science Foundation of China (Grant No. 61828203).

References

Benoit, G., Lemaitre, C., Lavenier, D., Drezen, E., Dayris, T., Uricaru, R., and Rizk, G. (2015). Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph. *BMC Bioinformatics*, **16**(1), 288.

Bonfield, J. K. and Mahoney, M. V. (2013). Compression of FASTQ and SAM format sequencing data. *PloS One*, **8**(3), e59190.

Cánovas, R., Moffat, A., and Turpin, A. (2014). Lossy compression of quality scores in genomic data. *Bioinformatics*, **30**(15), 2130–2136.

Chandak, S., Tatwawadi, K., Weissman, T., and Birol, I. (2018). Compression of genomic sequencing reads via hash-based reordering: algorithm and analysis. *Bioinformatics*, **34**(4), 558–567.

Cox, A. J., Bauer, M. J., Jakobi, T., and Rosone, G. (2012). Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, **28**(11), 1415–1419.

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

Deorowicz, S. and Grabowski, S. (2013). Data compression for sequencing data. *Algorithms for Molecular Biology*, **8**, 25.

Ginart, A. A., Hui, J., Zhu, K., Numanagić, I., Courtade, T. A., Sahinalp, S. C., and David, N. T. (2018). Optimal compressed representation of high throughput sequence data via light assembly. *Nature Communications*, **9**(1), 566.

Goodwin, S., McPherson, J. D., and McCombie, W. R. (2016). Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, **17**(6), 333–351.

Grabowski, S., Deorowicz, S., and Roguski, Ł. (2014). Disk-based compression of data from genome sequencing. *Bioinformatics*, **31**(9), 1389–1395.

Greenfield, D. L., Stegle, O., and Rrustemi, A. (2016). GeneCodeq: quality score compression and improved genotyping using a Bayesian framework. *Bioinformatics*, **32**(20), 3124–3132.

Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. (2012). SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, **28**(23), 3051–3057.

Hach, F., Numanagic, I., and Sahinalp, S. C. (2014). DeeZ: reference-based compression by local assembly. *Nature Methods*, **11**(11), 1082.

Jones, D. C., Ruzzo, W. L., Peng, X., and Katze, M. G. (2012). Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*, **40**(22), e171–e171.

Kingsford, C. and Patro, R. (2015). Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, **31**(12), 1920–1928.

Koboldt, D. C., Ding, L., Mardis, E. R., and Wilson, R. K. (2010). Challenges of sequencing human genomes. *Briefings in Bioinformatics*, **11**(5), 484–498.

Li, H. (2016). Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, **32**(14), 2103–2110.

Liu, Y., Peng, H., Wong, L., and Li, J. (2017). High-speed and high-ratio referential genome compression. *Bioinformatics*, **33**(21), 3364–3372.

Malysa, G., Hernaez, M., Ochoa, I., Rao, M., Ganesan, K., and Weissman, T. (2015). QVZ: lossy compression of quality values. *Bioinformatics*, **31**(19), 3122–3129.

Marçais, G., Pellow, D., Bork, D., Orenstein, Y., Shamir, R., and Kingsford, C. (2017). Improving the performance of minimizers and winnowing schemes. *Bioinformatics*, **33**(14), i110–i117.

Numanagić, I., Bonfield, J. K., Hach, F., Voges, J., Ostermann, J., Alberti, C., Mattavelli, M., and Sahinalp, S. C. (2016). Comparison of high-throughput sequencing data compression tools. *Nature Methods*, **13**(12), 1005–1008.

Ochoa, I., Hernaez, M., Goldfeder, R., Weissman, T., and Ashley, E. (2016). Effect of lossy compression of quality scores on variant calling. *Briefings in Bioinformatics*, **18**(2), 183–194.

Patro, R. and Kingsford, C. (2015). Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, **31**(17), 2770–2777.

Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M., and Yorke, J. A. (2004). Reducing storage requirements for biological sequence comparison. *Bioinformatics*, **20**(18), 3363–3369.

Roguski, Ł., Ochoa, I., Hernaez, M., and Deorowicz, S. (2018). FaStore: a space-saving solution for raw sequencing data. *Bioinformatics*, **34**(16), 2748–2756.

Sarkar, H. and Patro, R. (2017). Quark enables semi-reference-based compression of RNA-seq data. *Bioinformatics*, **33**(21), 3380–3386.

Tembe, W., Lowey, J., and Suh, E. (2010). G-SQZ: compact encoding of genomic sequence and quality data. *Bioinformatics*, **26**(17), 2192–2194.

Wandelt, S., Bux, M., and Leser, U. (2014). Trends in genome compression. *Current Bioinformatics*, **9**(3), 315–326.

Yu, Y. W., Yorukoglu, D., Peng, J., and Berger, B. (2015). Quality score compression improves genotyping accuracy. *Nature Biotechnology*, **33**(3), 240.

Zhang, Y., Li, L., Yang, Y., Yang, X., He, S., and Zhu, Z. (2015). Light-weight reference-based compression of FASTQ data. *BMC Bioinformatics*, **16**(1), 188.

Zhu, Z., Zhang, Y., Ji, Z., He, S., and Yang, X. (2013). High-throughput DNA sequence data compression. *Briefings in Bioinformatics*, **16**(1), 1–15.