Allowing mutations in maximal matches boosts genome compression performance

Yuansheng Liu [1], Limsoon Wong [2] and Jinyan Li [1,*]

[1] Advanced Analytics Institute, Faculty of Engineering and IT, University of Technology Sydney, Ultimo NSW 2007, Australia,
[2] School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417.

[*] To whom correspondence should be addressed.

Abstract

Motivation: A maximal match between two genomes is a contiguous non-extendable sub-sequence common in the two genomes. DNA bases mutate very often from the genome of one individual to another. When a mutation occurs in a maximal match, it breaks the maximal match into shorter match segments. The coding cost using these broken segments for reference-based genome compression is much higher than that of using the maximal match which is allowed to contain mutations.
Results: We present memRGC, a novel reference-based genome compression algorithm that leverages mutation-containing matches for genome encoding. MemRGC detects maximal matches between two genomes using a coprime double-window k-mer sampling search scheme, the method then extends these matches to cover mismatches (mutations) and their neighboring maximal matches to form long and mutation-containing matches. Experiments reveal that memRGC boosts the compression performance by an average of 27% in reference-based genome compression. MemRGC is also better than the best state-of-the-art methods on all of the benchmark data sets, sometimes better by 50%. Moreover, memRGC uses much less memory and de-compression resources, while providing comparable compression speed. These advantages are of significant benefits to genome data storage and transmission.
Availability and Implementation: https://github.com/yuansliu/memRGC
Contact: Jinyan.Li@uts.edu.au
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

The massive amount of genomic sequences produced by next-generation sequencing platforms presents great challenges to data storage and transmission (O'Leary *et al.*, 2015; Goodwin *et al.*, 2016). There is an imperative need to reduce the file size of these sequences by lossless compression to mitigate the associated concerns and issues (Chen *et al.*, 2002; Matos *et al.*, 2013; Hayashida *et al.*, 2014; Hernaez *et al.*, 2019). General-purpose text-compression tools such as gzip (www.gzip.org), bzip2 (www.bzip.org) and 7zip (www.7zip.org) can be directly applied to compress these data, but their compression performance is low as these methods are not specially designed to fully exploit the intrinsic patterns in this type of sequences (Zhu *et al.*, 2013; Deorowicz and Grabowski, 2013; Pinho and Pratas, 2014; Numanagić *et al.*, 2016). Over the past two decades, specialized algorithms have been proposed to compress raw sequencing data (Liu *et al.*, 2019b; Chandak *et al.*, 2019; Kowalski and Grabowski, 2020) and assembled genomes (Wandelt *et al.*, 2014). These methods include reference-based and *de novo* methods. A lot of evaluations demonstrated that reference-based compression can achieve much better performance than *de novo* compression (Zhu *et al.*, 2013; Numanagić *et al.*, 2016; Hosseini *et al.*, 2016). The basic idea of reference-based genome compression is to exploit the high similarities between two genomes of the same species (Wang and Zhang, 2011; Pinho *et al.*, 2011; Pratas *et al.*, 2017), e.g., the more than 99% similarity between human genomes (Lander *et al.*, 2001), to encode a target genome using the reference genome. Some of the reference-based genome compression methods, including DNAZip (Christley *et al.*, 2008), GenomeZip (Pavlichin *et al.*, 2013) and TGC (Deorowicz *et al.*, 2013), also make use of variation data of a genome relative to a reference genome. However, variation data are often unavailable, and generating them is a problem of high complexity. In this work, we focus on algorithms for reference-based genome compression without variation data (Wang and Zhang, 2011; Deorowicz and Grabowski, 2011; Ochoa *et al.*, 2015; Saha and Rajasekaran, 2015, 2016; Liu *et al.*, 2017; Shi *et al.*, 2018). Pairwise reference-based genome compression without variation data is also a foundation step for compressing genome collections (sets of genomes).

Exact matches have been used as an effective approach to measure similarities between long sequences (Kurtz *et al.*, 2004; Volfovsky *et al.*, 2001). For two sequences S_1 and S_2, an exact match (or simply a match) between S_1 and S_2 is a common sub-sequence in both S_1 and S_2. In reference-based genome compression problems, an exact match in the target genome can be concisely represented by a pair of integers (the start position of the match in the reference sequence and the length of the match). When lots of long matches exist between the reference and target genomes, the compression performance can be very high.

Existing methods have explored different ways to detect exact matches and use them to map the target sequence to the reference for compression. For example, Lempel-Ziv parsing has been widely used to find long prefix matches, e.g. in RLZ (Kuruppu *et al.*, 2010), RLZ-opt (Kuruppu *et al.*, 2011), GDC (Deorowicz and Grabowski, 2011), FRESCO (Wandelt and Leser, 2013), and GDC-2 (Deorowicz *et al.*, 2015). Most recent methods (Ochoa *et al.*, 2015; Saha and Rajasekaran, 2015; Liu *et al.*, 2017; Shi *et al.*, 2018) employ a left-to-right greedy mapping strategy. For example, ERGC (Saha and Rajasekaran, 2015) divides both the reference and the target genome into fixed-length blocks and the i-th block of the target genome is mapped to the i-th block of the reference genome from the left to right using a hash table. NRGC (Saha and Rajasekaran, 2016) improves the performance of ERGC. It designs a greedy placement scheme to place the i-th block of the target sequence to map to a good position at the reference genome rather than at the fixed i-th block of the reference genome. iDoComp (Ochoa *et al.*, 2015) generates a suffix array from the reference sequence and detects exact matches from the left to right of the target sequence; and then merges consecutive matches to form longer ones. HiRGC (Liu *et al.*, 2017) maps the target to the reference genome left-to-right using a hash table and a global greedy matching technique. HiRGC has a stable and robust performance, and it is not sensitive to the selection of the reference genome. Recently, HiRGC's overall performance was exceeded by SCCG (Shi *et al.*, 2018). SCCG combines the key ideas of ERGC and HiRGC, and estimates whether the block segmentation is needed or not before the global greedy matching. Nevertheless, SCCG's performance can be worse than iDoComp in some cases. So far, there is no all-win methods on widely used benchmark data sets, although they have made steady progress on overall compression performance.

These greedy mapping approaches have a common drawback — they do not aim to detect long matches. Instead of finding long matches, segments of them are detected and output separately. If a method aims to detect and use long matches, the encoding cost can be much smaller than the cost of using multiple shorter matches; cf. Fig. 1(a).

In this work, we propose to detect *maximal* matches between two genome sequences and extend these maximal matches to cover some mismatches (mutations) and their neighboring maximal matches for reference-based genome compression. A maximal match between two genomes is a sub-sequence in both genomes that is non-extendable at both ends. No existing genome compression methods (Numanagić *et al.*, 2016; Hernaez *et al.*, 2019) use this definition to detect matches. The extension of a maximal match to cover some mutations and its neighboring maximal matches is another novelty of our method. We define a mutation-containing match between two genomes as a string that is a sub-sequence in one of the two genomes and that is also a sub-sequence in the other genome after some mutations not at the beginning or ending position of the string. Use of mutation-containing matches for genome compression is effective because DNA base mutations or SNPs have been frequently observed in genomes. A mutation breaks a long exact match and divides the match into shorter matching segments. The coding cost by these broken segments for reference-based genome compression is much higher than the cost of using the maximal match when it is allowed to contain mutations; cf. Fig 1(b).

These ideas form the basis of our method memRGC, which stands for "maximal exact matches for reference-based genome compression". An improved version of our previous algorithm bfMEM (Liu *et al.*, 2019a) is developed here to detect maximal exact matches (MEMs). The method in the new version samples k-mers on both the reference and target

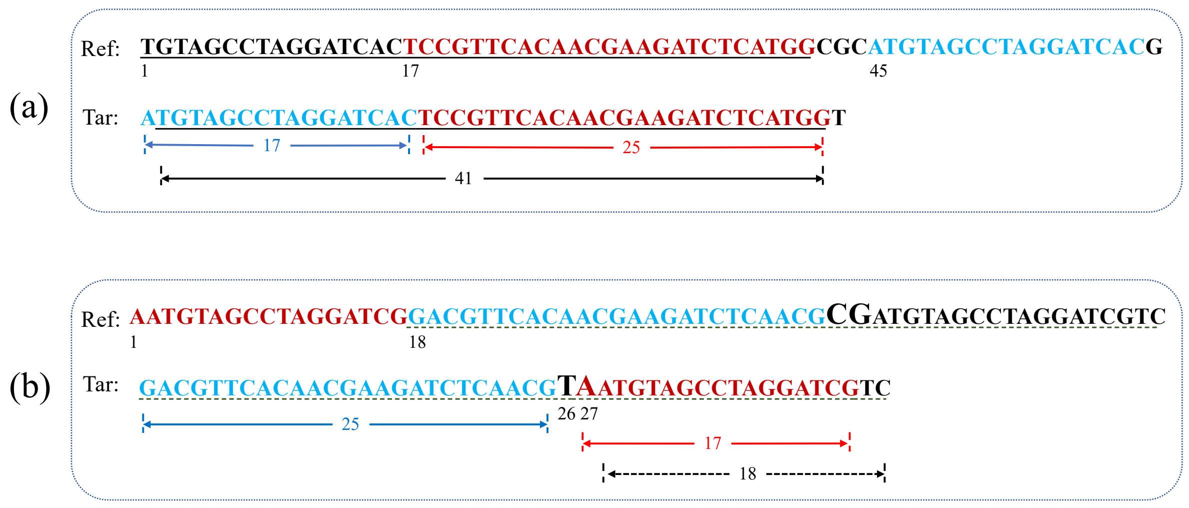
(a)
Ref: TGTAGCCTAGGATCACTCCGTTCACAACGAAGATCTCATGGCGCATGTAGCCTAGGATCACG
1
17
45
Tar: ATGTAGCCTAGGATCACTCCGTTCACAACGAAGATCTCATGGT
17
25
41
(b)
Ref: AATGTAGCCTAGGATCGGACGTTCACAACGAAGATCTCAACGCGATGTAGCCTAGGATCGTC
1
18
Tar: GACGTTCACAACGAAGATCTCAACGTAATGTAGCCTAGGATCGTC
25
26 27
17
18

Fig. 1. Genome encoding by different match-detection approaches. (a) The left-to-right mapping method detects two exact matches (highlighted in red and blue color) and the encoding of the target sequence is $\{(45, 17); (17, 25); \mathrm{T}\}$; The encoding can come shorter as $\{\mathrm{A}; (1, 41); \mathrm{T}\}$ when the long exact match (underlined) is detected. (b) The mutations at positions 26 and 27 break a long match into two shorter matches. The left-to-right method can find these two matches, i.e., the blue substring of length 25 and the red substring of length 17; and the encoding of the target sequence is $\{(18, 25), \mathrm{T}, (1, 17), \mathrm{TC}\}$. The mutation-containing match is labeled by a dotted underline and the encoding of the target sequence is $\{(18, 25), \mathrm{TA}, (18)\}$, where 18 is the length of the second match as the start position of the second match does not need to be encoded. The gain of the mutation-containing match encoding is no need of storing the start position in coding the second match.

sequences (Grabowski and Bieniecki, 2018) under a coprime window-length constraint, and implements a Bloom filter (Bloom, 1970) to filter some unnecessary k-mers for reducing the number of indexing k-mers. Mutation-containing matches are then determined by extending each MEM at its left- and right-end. In a mutation-containing match, the maximum length of the mismatching substring is set as 2 in this work and the minimal length of matching substring is set as 3. The gain of compression performance after the extension of maximal matches depends on these two thresholds (evaluation and experiments are shown in Supplementary file).

Attributed to the coprime double-window sampling technique and the Bloom filter indexing idea, memRGC stores only a small fraction of k-mers for mapping, and results in much less memory usage ($<$ 2 GB) than existing methods. For example, ERGC, HiRGC and SCCG have to construct a full index of the reference genome, demanding more than 6 GB RAM in practice for a human reference genome. iDoComp not only constructs a full suffix-array; it also needs to store exact matches in memory for merging. If large numbers of matches are detected, iDoComp requires a lot of memory.

2 Materials and methods

MemRGC proceeds in two stages (Fig. 2): (i) detection of mutation-containing matches from a pair of genomes for concisely mapping the target genome to the reference genome; (ii) alternative encoding, that selects a better encoding method to generate a smaller encoded file. Note that, memRGC performs lossless compression and does not limit the alphabet set in the two genomes.

Let $R = r_1r_2\cdots r_n$ and $T = t_1t_2\cdots t_m$ be the reference genome and the target genome respectively. We use notation S_i^L to denote a substring of length L starting at position i of a sequence S, i.e., $S_i^L = s_is_{i+1}\cdots s_{(i+L-1)}$. A maximal exact match (MEM) is represented by a tuple (i, j, L) satisfying $R_i^L = T_j^L$, $r_{i-1} \neq t_{j-1}$ and $r_{i+L} \neq t_{j+L}$, where we always set $r_0 \neq t_0$ and $r_{n+1} \neq t_{m+1}$. Note that exact matches are case-insensitive. MemRGC can deal with mixtures of upper cases and lowercases as described in the encoding stage.

Detection of mutation-containing matches from a pair of genomes

A mutation-containing match is formed after extending from a maximal exact match to cover mismatches and its neighboring maximal exact matches between the two genomes.

Detection of maximal exact matches

An improved version of our previous algorithm bfMEM (Liu *et al.*, 2019a) is developed to detect MEMs. The core idea is to sample k-mers on both R and T and to filter some unnecessary indexing k-mers using a Bloom filter. The k-mer sampling selects the last k-mer from a block of s number of contiguous k-mers, where s is the sampling interval (or the sampling window).

The original bfMEM (Liu *et al.*, 2019a) has three steps. (i) Sample k-mers on the target genome T with a sampling window s_2 and all the sampled k-mers are added into a Bloom filter f. (ii) All k-mers of the reference genome R are tested to see whether they are in f. Those k-mers that pass the Bloom filter test are inserted into a hash table $\mathcal{H}$ (k-mer as key and its position as value) for indexing. (iii) For all the sampled k-mers from T, they are used as the key to retrieve possible left and right extensions by querying $\mathcal{H}$. The original bfMEM samples k-mers on T only, and all k-mers of R are tested on the Bloom filter. Grabowski and Bieniecki (Grabowski and Bieniecki, 2018) found that sampling on both R and T can be applied if the two sampling windows are coprime. Taking this idea, we improve bfMEM by modifying the second step as: sampling k-mers on the reference genome R with a sampling window s_1 and all the sampled k-mers are tested to see whether they are in f. The details of our MEM detection are summarized in Algorithm 1. In our implementation, the tool BioBloom (Chu *et al.*, 2014) is used to create Bloom filter and ntHash (Mohamadi *et al.*, 2016) is used to calculate hash values for k-mers. Moreover, the false positive probability (FPP) of the Bloom filter is set as 0.01. Experiments shows that the FPP has little effect on compression performance (See supplementary file).

This coprime double-window k-mer sampling on R and T is sufficient to guarantee the completeness of the search space of MEMs with a length $\geq L$. Let the sampling window on the reference genome and on the target genome be s_1 and s_2 respectively, where the greatest common divisor of s_1 and s_2 is required to be 1 (i.e., coprime), the length of k-mer be k, and (i, j, L) be an MEM. We prove that this MEM is guaranteed to be generated from one of these k-mers.

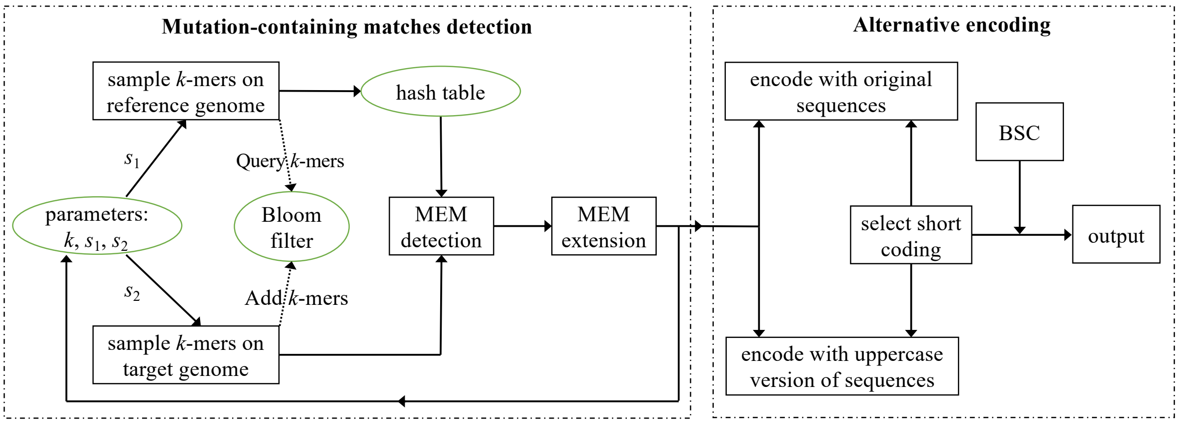
Mutation-containing matches detection
sample k-mers on reference genome
hash table
s_1
Query k-mers
parameters: k, s_1, s_2
Bloom filter
MEM detection
MEM extension
s_2
Add k-mers
sample k-mers on target genome
Alternative encoding
encode with original sequences
BSC
select short coding
output
encode with uppercase version of sequences

Fig. 2. Overview of our algorithm memRGC. It has two stages, viz. the mutation-containing matches detection stage and the alternative encoding stage. In the first stage, maximal exact matches (MEMs) are iteratively detected and extended by different settings of k-mer, and the sampling windows s_1 and s_2. In the encoding stage, two encoding schemes are employed in accordance to whether the original sequences or the uppercase version of the sequences is used. Then the shorter code is compressed by BSC (http://libbsc.com) as the final compressed file.

Algorithm 1: MEM detection by double k-mer sampling and Bloom filtering

```
Input: Reference sequence $R = r_1 r_2 \cdots r_n$ and target sequence
       $T = t_1 t_2 \cdots t_m$, size of $k$-mer $k$, sampling windows $s_1$
       and $s_2$
Output: MEMs

Function MemDetectionSketch$(R, T, k, s_1, s_2)$ begin
    $f \leftarrow$ empty Bloom filter of size $\left\lceil \frac{(m-k+1)/s_2 * \log_2 0.01}{\ln(1-0.01^{1/6})} \right\rceil$
    for $i \leftarrow s_2$ to $(m-k+1)$ by $s_2$ do                ▷ Sample on $T$
        Add $T_i^k$ to $f$
    $\mathcal{H} \leftarrow$ empty hash table
    for $i \leftarrow s_1$ to $(n-k+1)$ by $s_1$ do                ▷ Sample on $R$
        if $R_i^k \in f$ then
            Append $i$ to $\mathcal{H}[R_i^k]$
    $Z \leftarrow$ an empty array used to store MEMs
    for $i \leftarrow s_2$ to $(m-k+1)$ by $s_2$ do                ▷ Sample on $T$
        foreach $j \in \mathcal{H}[T_i^k]$ do
            $(b, e) \leftarrow (1, 1)$
            while $t_{i-b} = r_{j-b}$ do
                $b \leftarrow b + 1$;                              ▷ Left extension
            while $t_{i+e} = r_{j+e}$ do
                $e \leftarrow e + 1$;                              ▷ Right extension
            if $e + b \geq s_1 * s_2 + k$ then
                Append $(i-b, j-b, e+b-1)$ to $Z$
    return $Z$
end
```

First, the sampled k-mers in the substring R_i^L are its r-th k-mers, where $r = s_1 - ((i-1) \mod s_1) + s_1 * x, 1 \leq r \leq (L-k+1)$, $0 \leq x \leq \lfloor (L-k+1)/s_1 \rfloor$, and $\lfloor c \rfloor$ returns the largest integer not larger than c. Second, the sampled k-mers in the substring T_j^L are the t-th k-mers, where $t = s_2 - ((j-1) \mod s_2) + s_2 * y, 1 \leq t \leq (L-k+1)$ and $0 \leq y \leq \lfloor (L-k+1)/s_2 \rfloor$. According to the Chinese remainder theorem, given two coprime integers s_1 and s_2, for any integers a and b, there exists an integer $z \in [0, s_1 \times s_2)$ such that

$$\begin{cases} z \equiv a \pmod{s_1}, \\ z \equiv b \pmod{s_2}. \end{cases}$$

Therefore, for any integers i and j, the equation

$$\begin{cases} r = s_1 - ((i-1) \mod s_1) + s_1 * x, \\ t = s_2 - ((j-1) \mod s_2) + s_2 * y, \end{cases}$$

has a unique solution $r = t \in [0, s_1 \times s_2)$. Combining $1 \leq r, t \leq (L-k+1)$, the sampling method guarantees that at least one shared k-mer is sampled in both R_i^L and T_j^L iff $L \geq s_1 * s_2 + k - 1$. Any MEM having length $\geq (s_1 * s_2 + k - 1)$ including MEM (i, j, L) can be detected by some left- and/or right-extensions from such sampled k-mers.

Extension of an MEM to cover mismatches and neighboring MEMs

Let (i, j, L) be an MEM. Suppose $R_{i+L}^p \neq T_{j+L}^p$, $R_{i+L+p}^u = T_{j+L+p}^u$ and $R_{i+L+p+u} \neq T_{j+L+p+u}$, we define MEM $(i+L+p, j+L+p, u)$ as a neighboring MEM from the right-side extension of MEM (i, j, L). We also define $(i, j, L+p+u)$ as a mutation-containing match (MCM) which covers the mismatch sequence R_{i+L}^p and the neighboring MEM $(i+L+p, j+L+p, u)$. Here, the substring comparison $R_{i+L}^p \neq T_{j+L}^p$ is defined as: $r_{i+L+z} \neq t_{j+L+z}$ for $0 \leq z < p$.

The left-side extension can be conducted similarly as the above right-side extension. Suppose $R_{i-p}^p \neq T_{j-p}^p$, $R_{i-p-u}^u = T_{j-p-u}^u$ and $R_{i-p-u-1} \neq T_{j-p-u-1}$, we define MEM $(i-p-u, j-p-u, u)$ as a neighboring MEM from the left-side extension of MEM (i, j, L). We also define $(i-p-u, j-p-u, L+p+u)$ as a mutation-containing match (MCM) which covers the mismatch sequence R_{i-p}^p and the neighboring MEM $(i-p-u, j-p-u, u)$.

We set a maximum threshold for p and a minimum threshold for u. After several rounds of left-side and right-side extensions, a long mutation-containing match can be detected. In our implementation, the threshold for p is set as 2, and the threshold for u is set as 3. That means: the maximum length of a mismatching sequence is 2 and the minimum length of a neighboring MEM is 3. Compression performance can be affected by different settings of these thresholds. We performed some experiments on four reference-target genome pairs with the mismatch threshold ranging from 2 to 10, and the neighboring MEM length threshold ranging from 1 to 10 (see results in Supplementary file).

Iterative generation of MEMs and MCMs

We did not take a straightforward approach to detect all MEMs between R and T by setting one fixed minimum length L. Instead, we detect long MEMs first. After their extensions to form mutation-containing matches, we detect long MEMs from the remaining segments of R and T which have not been covered by the mutation-containing matches. The main reason of this adaption is because the extension step can detect many neighboring MEMs. If the straightforward approach is applied, these neighboring MEMs are redundantly detected.

We give an example to illustrate how the three parameters of MEM detection (the length of k-mer and two sampling windows s_1 and s_2) are iteratively set for detecting all the MEMs and mutation-containing matches. First, we set $k = 90$, $s_1 = 1,000$ and $s_2 = 999$ to detect long MEMs having minimum length $\geq (1,000 * 999 + 90 - 1) = 999,089$. These detected MEMs are sorted by their length in a descending order. Non-overlapping MEMs are selected and the substrings of these MEMs in the target genome T are mapped to the corresponding substrings of the reference genome. After mapping, the left-side and/or right-side extensions of these MEMs are carried out to detect mutation-containing matches (MCMs). Then, those segments in the target sequence T covered by the MCMs are labeled, and they are excluded from participating in subsequent rounds. In the second round, we set $s_1 = 1000$ and $s_2 = 499$ to detect MEMs having minimum length $\geq (1,000 * 499 + 90 - 1) = 499,089$, and then detect MCMs, and then label segments covered by these MCMs. The iteration terminates when short segments are not detected in T. Note that the setting of k affects not only the length of MEM but also the speed of MEM detection. For genomes of small size, the parameters k, s_1 and s_2 should be set small as well. For example, $k = 20, s_1 = 7, s_2 = 1$ are used in our implementation.

We note that it is difficult to optimize this iterative framework. It has slight effects on the compression performance and running time. Some evaluation results on this iterative framework are presented in Supplementary file.

Alternative encoding

We encode the target sequence left-to-right like HiRGC (Liu *et al.*, 2017). The basic procedure is: for an MEM, it is stored as two integers in a line, i.e., the start position in the reference sequence and its length; for mismatch bases or substrings, they are stored in a separate line. For two MEMs (i_0, j_0, L_0) and (i_1, j_1, L_1), they are in a mutation-containing match if $i_1 - (i_0 + L_0) = j_1 - (j_0 + L_0)$. The start position of the second MEM does not need to be stored as it is equal to the end position of the first MEM plus the length of the unmapped substring between these two

MEMs. Finally, the encoded file is compressed by the BSC compressor (http://libbsc.com).

To deal with upper cases and lowercases, the following two encoding schemes are employed:

- Encode with the uppercase version of the sequences. First, find the intervals of lowercase letters in T and save these intervals in the encoded file. Then, the bases in R and T are converted to uppercases. Finally, the mapped and mismatching bases are encoded as the above description.
- Encode with the original sequences. To store the case information, a match is divided into some fragments according to whether the cases are the same or different between the target and reference genome. Then, the MEM is stored as a series of numbers including the start position in the reference sequence and the lengths of those fragments. For instance, an MEM "ATTgcTAG" is divided into four fragments and encoded by 4 length integers "3 2 2 1" if the substring of this match at the reference genome is "ATTGCTAg".

The shorter coded one is stored in a file alone with a label of the encoding method.

3 Results

All experiments were carried out on a computing cluster running Red Hat Enterprise Linux 7.5 (64 bit) equipped with 2.7 GHz Intel® Xeon® 6150 (18 Cores), 384 GB RAM and 10 TB disk space. We compared memRGC with four state-of-the-art compression algorithms: iDoComp (Ochoa *et al.*, 2015), ERGC (Saha and Rajasekaran, 2015), HiRGC (Liu *et al.*, 2017) and SCCG (Shi *et al.*, 2018).

Eight human genomes including six reference genomes (viz., HG17, HG18, HG19, HG38, KOREF_20090131 (denoted by K131) and KOREF_20090224 (denoted by K224) (Ahn *et al.*, 2009)) and two individual genomes (YH (Wang *et al.*, 2008) and HuRef (Levy *et al.*, 2007)) were pairwise tested for reference-based genome compression by each of these methods. Moreover, two genome collections consisting of 50 and 200 human genome sequences from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015) were tested using an external reference for compression. Furthermore, we conducted experiments for *reference-free* compression of genome collections where internal reference-based genome compression is a foundation step. We collected 11 human reference genomes for the purpose of reference-free compression. In addition, genomes of some other species such as *C. elegans*, *S. cerevisiae*, *A. thaliana*, *O. sativa* and *C. lupus*, were tested. These data sets from a variety of species are widely used benchmark data sets (Ochoa *et al.*, 2015; Saha and Rajasekaran, 2015; Liu *et al.*, 2017; Shi *et al.*, 2018). They have very different features including the alphabets they contain, the number of chromosomes and the genome size. More details of these genomes are listed in Supplementary Table S1.

Performance comparison for pairwise reference-based genome compression

A total of $8 \times 7 = 56$ reference-target pairwise genome compressions were conducted. The file size of every target genome after compression with regard to each reference genome by iDoComp (Ochoa *et al.*, 2015), ERGC (Saha and Rajasekaran, 2015), HiRGC (Liu *et al.*, 2017), SCCG (Shi *et al.*, 2018) or our memRGC is presented in Supplementary Table S4 and part of results (24 reference-target pairs) is shown in Table 1. The table shows the *compression gain* of memRGC over the best result of these state-of-the-art methods. It is calculated by $(\frac{\text{compression rate of memRGC}}{\text{the best compression rate of the other methods}} - 1) * 100\%$, where the compression rate of a method on a data set is calculated as the original file size divided by the compressed file size (i.e., the compression fold). Note that the compressed file does not include the reference genome, strictly the same as the existing reference-based genome compression tools did. The numbers inside brackets in the compression gain column of the table are the absolute volume of file size reduction (in megabytes) from the second-best in the row.

Our memRGC achieves the best compression performance on all of these pairwise reference-based genome compressions (all-win). In particular, our method achieves around 10% to 50% compression gain compared to the best result of the other methods for 37 of the total 56 cases. There are other 12 cases, where the compression gain is even higher (more than 50%). For the remaining 7 cases, the compression gain is around 8%. Averaging over the 56 reference-based genome compressions, memRGC's compression gain is 27%. The best compression gain by memRGC is 83.57%, where the compressed file size of HG38 is further reduced from SCCG's 17.99 megabytes to only 9.80 megabytes.

SCCG wins the second-best performance 49 times. On the remaining 7 cases, iDoComp is the second-best algorithm. For these 7 cases, iDoComp is much better than SCCG and our memRGC further achieves at least 49% improvement. On 12 of the 56 cases, ERGC has very close (< 1 MB) performance to the second-best.

We also conducted experiments for random pairwise genomes from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015). MemGRC is always better than the other methods. These reference and target genomes are very similar, and the compression folds achieved are high. Some examples are shown in the last two rows of Table 1. We note that the performance of GDC-2 (Deorowicz *et al.*, 2015) is also very good at compressing these highly similar reference-target pairs. Although the compressed file size of NA12413 and that of NA20770 by GDC-2 are only 16.89 MB and 4.71 MB respectively, our memRGC's performance is still superior.

Performance comparison on memory usage

The memory usages by these methods for the 56 pairwise reference-based compressions are shown in Supplementary Figure S1. The memory consumption by memRGC is at least 6.8, 3.8 and 5.5 times smaller than ERGC, HiRGC and SCCG respectively. iDoComp's memory consumption is less than memRGC on 13 cases when the suffix-array (SA) generation is excluded. However, for other 32 cases, iDoComp's memory consumption is 6 times more than memRGC. If the memory used in the SA generation is included, memRGC always uses less memory than iDoComp. The memory usage saved is attributed to memGRC's double-sampling technique and the filtering method that stores only a small fraction of k-mers for indexing.

Performance comparison on compression speed

Supplementary Table S2 presents detailed running time (wall-clock) of the methods. The total wall-clock time is measured using `/usr/bin/time -v` Unix command. HiRGC is the fastest tool for the compression of all the 56 reference-target pairs. In most cases, HiRGC is about twice as fast as memRGC. However, memRGC is at least 1.5 times faster than ERGC. MemRGC is faster than SCCG except for two cases, and is 1.5 times faster in 27 cases. For iDoComp, if the time used to generate the SA is included, it is at least 2 folds slower than memRGC in 53 cases. If the time of the SA generation is not included, memRGC is slightly faster than iDoComp in 32 cases and iDoComp is much faster than memRGC in the other 24 cases.

Computational resources utilization for decompression

Supplementary Table S3 presents detailed running time (wall-clock) and memory usage of memRGC vs the others during decompression.

Table 1. Compressed file size (do not include the size of reference genome), compression gain, memory reduction and time reduction by different methods for the pairwise reference-target genome compression

Reference	Target	Raw size (MB)	File size (MB) after compression by					Compression gain	Memory	Time
			iDoComp	ERGC	HiRGC	SCCG	memRGC	(file size reduction)	reduction	reduction
YH	HG38	3004.12	64.82	580.54	27.04	*26.03*	**21.89**	18.91% (4.14)	74.57%	29.96%
	HuRef	2717.67	14.04	478.43	13.97	*13.34*	**11.50**	16.00% (1.84)	35.57%	22.94%
	K224	2986.70	18.10	11.57	16.05	*10.86*	**9.52**	14.08% (1.34)	61.97%	7.27%
HG17	HG19	3011.33	*8.23*	379.90	11.65	11.57	**5.33**	54.41% (2.90)	37.17%	65.74%
	HG18	2996.49	*3.37*	171.80	10.34	10.30	**2.25**	49.78% (1.12)	33.89%	38.97%
	HG38	3004.12	29.60	580.92	19.70	*19.46*	**13.39**	45.33% (6.07)	49.80%	53.58%
HG18	HG19	3011.33	*6.20*	299.45	10.66	10.62	**4.02**	54.23% (2.18)	35.36%	68.26%
	HG17	2992.95	*2.26*	87.61	9.43	9.42	**1.49**	51.68% (0.77)	34.53%	34.84%
	HG38	3004.12	28.12	582.28	18.99	*18.74*	**12.34**	51.86% (6.40)	48.83%	65.52%
HG19	HG38	3004.12	24.14	526.20	18.15	*17.99*	**9.79**	83.76% (8.20)	41.57%	59.21%
	HG17	2992.95	*6.18*	175.71	9.70	9.68	**3.88**	59.28% (2.30)	37.69%	49.86%
	HG18	2996.49	*5.26*	131.34	9.60	9.58	**3.32**	58.43% (1.94)	35.91%	57.83%
HG38	HG19	3011.33	*8.44*	184.49	11.55	11.43	**5.58**	51.25% (2.86)	35.17%	62.00%
	HG18	2996.49	11.45	266.51	11.25	*11.14*	**7.39**	50.74% (3.75)	44.05%	61.19%
	HG17	2992.95	11.75	333.13	11.09	*10.98*	**7.55**	45.43% (3.43)	45.39%	62.74%
K131	HG38	3004.12	237.04	603.86	28.56	*27.39*	**22.82**	20.03% (4.57)	74.40%	−17.97%
	HG19	3011.33	216.99	442.33	20.43	*19.42*	**17.24**	12.65% (2.18)	74.19%	41.00%
	YH	2986.68	32.62	8.97	13.33	*7.99*	**7.14**	11.90% (0.85)	57.80%	34.09%
K224	K131	2986.70	6.66	5.98	10.19	*5.84*	**4.40**	32.73% (1.44)	33.43%	52.83%
	HG38	3004.12	233.49	609.93	28.48	*27.33*	**22.73**	20.24% (4.60)	74.40%	23.68%
	HuRef	2717.67	27.34	483.47	14.43	*13.77*	**11.85**	16.20% (1.92)	37.71%	23.35%
HuRef	HG38	3004.12	171.50	619.79	41.78	*40.67*	**36.20**	12.35% (4.47)	74.57%	12.47%
	K131	2986.70	56.25	603.84	36.78	*32.16*	**28.80**	11.67% (3.36)	63.37%	0.22%
	K224	2986.70	53.22	602.50	35.26	*30.61*	**27.43**	11.59% (3.18)	62.45%	10.32%
HG00113	NA12413	2986.29	17.43	84.81	*16.78*	16.97	**15.39**	9.03% (1.39)	32.92%	32.26%
NA11919	NA20770	2986.50	*4.51*	70.72	4.99	4.91	**4.34**	3.44% (0.15)	32.88%	60.22%

Note: Bold font indicates the best result in the row. Italic font represents the second best. Memory reduction $= \left(1 - \frac{\text{memory usage of memRGC}}{\text{the smallest memory usage of other tools}}\right) * 100\%$. In the last column, the run time is only compared with the tool SCCG as it achieves the second best compression ratio in most cases. Time reduction is calculated by $\left(1 - \frac{\text{run time of memRGC}}{\text{run time of SCCG}}\right) * 100\%$.

MemRGC is consistently faster than the other methods for all of the cases. MemRGC is one order of magnitude faster than iDoComp in 32 cases and 3 times faster in the remaining 24 cases. It is at least 4.6 times faster than ERGC; a little faster than HiRGC; and 2.3 times faster than SCCG. In terms of memory usage for decompression, memRGC is an order of magnitude smaller than those of iDoComp, ERGC and SCCG. The memory requirement of HiRGC is 2 folds higher than that of memRGC. Thus memRGC achieves the fastest decompression speed with less memory than the other methods.

Using an external reference to compress genome collections from the 1000 Genomes Project

We compressed two genome collections containing 50 and 200 genomes from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015). The recently published human genome HG38 was chosen as the external reference genome. The genomes from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015) are stored in the VCF format and they were reconstructed to fasta files using vcf2fasta tool provided by GDC-2 (Deorowicz *et al.*, 2015). The raw size of the 50 genomes and of the 200 genomes are about 145.8 GB and 583.1 GB respectively. The method GDC-2 (Deorowicz *et al.*, 2015), specially designed to compress a collection of these genomes, is compared with out method on the performance.

The sizes of the compressed files are shown in Supplementary Table S5. In the compression of the 50 genomes, memRGC achieves a 1357-fold compression rate, with the file size of these genomes reduced to only 110 MB. The compressed file by GDC-2 is 4.8 times larger than memRGC's; and those by HiRGC and SCCG are at least 3 times larger.

Similarly in the compression of the 200 genomes, our method achieves the highest compression rate 1907-fold, where each genome is succinctly represented by a file of about 1.5 MB. The compressed files by HiRGC and SCCG are about 4 times larger than ours.

Another very recent method, HRCM (Yao *et al.*, 2019), is an improvement on HiRGC for genome collection compression by incorporating the two-level idea of FRESCO (Wandelt and Leser, 2013) and GDC-2 (Deorowicz *et al.*, 2015). Its performance on pairwise reference-based genome compression is not good, but HRCM achieves much progress in compressing a genome collection. The performance of HRCM is much better than the other four state-of-the-art methods; however, our memRGC still surpasses HRCM by about 10% compression gain.

No need of external reference for a compression of 11 human genomes

We considered a scenario of external-reference-free compression of a genome collection. Such a scenario relaxes the burden of selecting a good external reference genome to compress a genome collection. By memRGC, the first genome sequence in the collection is compressed by BSC, a block-sorting data compression tool. Before that it is used as a reference genome to compress one of the remaining genomes G2. This target genome G2 is identified through a search on the remaining genomes to find a genome that has the smallest code. Then we use this target genome (G2) as a reference genome to compress one of the remaining uncompressed genomes. Iteratively, memRGC constructs a chain structure of this genome collection for compression. This approach is different from the compression process of using the first genome stored in the genome collection as a fixed reference genome to compress all the remaining sequences. We note that for a large collection of human genomes, the search on the remaining genomes can be scoped to a small number such as 10 or 20.

This genome collection containing 11 genomes (32 GB) can be compressed down to only 748 MB by memRGC. The compressed file by GDC-2 has a size of 1216 MB, which is 1.6 times bigger than memRGC. For other methods such as HiRGC, SCCG and HRCM, they were not designed to compress the first reference genome. For a fair comparison, the volume of the first genome after compressed by BSC is added into their compressed files. The file size compressed by HiRGC or SCCG is 1016 MB or 987 MB respectively, much bigger than memRGC's. HRCM's compressed file is also 50 MB larger than memRGC's.

Compression performance on other species' genomes

To further assess the effectiveness and applicability of our memRGC, other species' genomes were tested as well. The experiments were conducted on 18 pairs of reference-target genomes and the compression results are presented in Supplementary Table S6. MemRGC achieves the best compression result for 17 out of the 18 pairs. On the only one remaining pair (where sacCer3 is used as reference and sacCer2 as target), the compressed file by memRGC is just 38 bytes larger than iDoComp's. MemRGC outperforms ERGC, HiRGC and SCCG on all of the pairs. Moreover, memRGC achieves at least 30% compression gain over iDoComp on 10 pairs. Compared with ERGC, memRGC has compression gains of at least 23% and up to 91% on 10 pairs. MemRGC outperforms HiRGC by an average of 34% (up to 96% in some single cases). MemRGC is also better than SCCG by at least 7% and up to 97% (29% on average).

Effect of the maximal match extension on compression performance

To demonstrate the impact of MEM extension, we ran experiments on 21 reference-target genome pairs without using the step of MEM extension. The results are presented in Supplementary Table S7. MemRGC without MEM extension has a lowered compression performance for all of the tested cases. For more than half of the reference-target pairs (12 cases), the impact of using the idea of MEM extension is very significant — the compressed files by memRGC without MEM extension is 20% larger than the standard memRGC.

4 Conclusion

Our extensive comparative experiments have demonstrated that memRGC is the best algorithm for reference-based genome compression. It is the first all-win method having better performance than current state-of-the-art methods on all the benchmark data sets of human genomes. The key and novel idea of our method is to detect maximal matches between genomes and then extend the maximal matches to cover some mutations and neighboring maximal matches for minimizing the codes of a target genome. The comparative experiments have also shown that the compression speed of memRGC is comparable to the existing methods, its de-compression speed is much faster, and its memory consumption is remarkably smaller. These advantages provided by our method will bring significant benefits to genome data storage and transmission. There remain several issues to overcome to improve the performance of memRGC. For example, the iterative procedure and the settings of k, s_1 and s_2 are coded in a brute-force way, and they have not been optimized for different reference-target pairs. As future work, it would be useful to consider mutation-containing matches for genome comparison and genome distance estimation. In the current implementation, memRGC needs to set up the reference-target genome pairs. It is cannot be directly used to compress genomes having large number of contigs such as white spruce (Warren *et al.*, 2015). We will adapt the current version for an efficient construction of the reference-target genome pairs when the genomes have large number of contigs.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of our manuscript.

Funding

This work has been partly supported by Australia Research Council Discovery Project ARC DP180100120.

References

Ahn, S.-M., Kim, T.-H., Lee, S., Kim, D., Ghang, H., Kim, D.-S., Kim, B.-C., Kim, S.-Y., Kim, W.-Y., Kim, C., *et al.* (2009). The first Korean genome sequence and analysis: full genome sequencing for a socio-ethnic group. *Genome Research*, **19**(9), 1622–1629.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7), 422–426.

Chandak, S., Tatwawadi, K., Ochoa, I., Hernaez, M., and Weissman, T. (2019). SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, **35**(15), 2674–2676.

Chen, X., Li, M., Ma, B., and Tromp, J. (2002). DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, **18**(12), 1696–1698.

Christley, S., Lu, Y., Li, C., and Xie, X. (2008). Human genomes as email attachments. *Bioinformatics*, **25**(2), 274–275.

Chu, J., Sadeghi, S., Raymond, A., Jackman, S. D., Nip, K. M., Mar, R., Mohamadi, H., Butterfield, Y. S., Robertson, A. G., and Birol, I. (2014). BioBloom tools: fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics*, **30**(23), 3402–3404.

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.

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

Deorowicz, S., Danek, A., and Niemiec, M. (2015). GDC 2: Compression of large collections of genomes. *Scientific Reports*, **5**, 11565.

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. and Bieniecki, W. (2018). copMEM: finding maximal exact matches via sampling both genomes. *Bioinformatics*, **35**(4), 677–678.

Hayashida, M., Ruan, P., and Akutsu, T. (2014). Proteome compression via protein domain compositions. *Methods*, **67**(3), 380–385.

Hernaez, M., Pavlichin, D., Weissman, T., and Ochoa, I. (2019). Genomic data compression. *Annual Review of Biomedical Data Science*, **2**(1), 19–37.

Hosseini, M., Pratas, D., and Pinho, A. (2016). A survey on data compression methods for biological sequences. *Information*, **7**(4), 56.

Kowalski, T. M. and Grabowski, S. (2020). PgRC: pseudogenome-based read compressor. *Bioinformatics*, **36**(7), 2082–2089.

Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, **5**(2), R12.

Kuruppu, S., Puglisi, S. J., and Zobel, J. (2010). Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In E. Chavez and S. Lonardi, editors, *String Processing and Information Retrieval*, pages 201–206, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kuruppu, S., Puglisi, S. J., and Zobel, J. (2011). Optimized relative lempel-ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference-Volume 113*, pages 91–98. Australian Computer Society, Inc.

Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., *et al.* (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), 860–921.

Levy, S., Sutton, G., Ng, P. C., Feuk, L., Halpern, A. L., Walenz, B. P., Axelrod, N., Huang, J., Kirkness, E. F., Denisov, G., *et al.* (2007). The diploid genome sequence of an individual human. *Plos Biology*, **5**(10), e254.

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

Liu, Y., Zhang, L. Y., and Li, J. (2019a). Fast detection of maximal exact matches via fixed sampling of query k-mers and bloom filtering of index k-mers. *Bioinformatics*, **33**(22), 4560–4567.

Liu, Y., Yu, Z., Dinger, M. E., and Li, J. (2019b). Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression. *Bioinformatics*, **35**(12), 2066–2074.

Matos, L. M., Pratas, D., and Pinho, A. J. (2013). A compression model for DNA multiple sequence alignment blocks. *IEEE Transactions on Information Theory*, **59**(5), 3189–3198.

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

Mohamadi, H., Chu, J., Vandervalk, B. P., and Birol, I. (2016). ntHash: recursive nucleotide hashing. *Bioinformatics*, **32**(22), 3492–3494.

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., and Weissman, T. (2015). iDoComp: a compression scheme for assembled genomes. *Bioinformatics*, **31**(5), 626–633.

O'Leary, N. A., Wright, M. W., Brister, J. R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D., *et al.* (2015). Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Research*, **44**(D1), D733–D745.

Pavlichin, D. S., Weissman, T., and Yona, G. (2013). The human genome contracts again. *Bioinformatics*, **29**(17), 2199–2202.

Pinho, A. J. and Pratas, D. (2014). MFCompress: a compression tool for FASTA and multi-FASTA data. *Bioinformatics*, **30**(1), 117–118.

Pinho, A. J., Pratas, D., and Garcia, S. P. (2011). GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, **40**(4), e27–e27.

Pratas, D., Hosseini, M., and Pinho, A. J. (2017). Substitutional tolerant Markov models for relative compression of DNA sequences. In *International Conference on Practical Applications of Computational Biology & Bioinformatics*, pages 265–272. Springer.

Saha, S. and Rajasekaran, S. (2015). ERGC: An efficient referential genome compression algorithm. *Bioinformatics*, **31**(21), 3468–3475.

Saha, S. and Rajasekaran, S. (2016). NRGC: a novel referential genome compression algorithm. *Bioinformatics*, **32**(22), 3505–3412.

Shi, W., Chen, J., Luo, M., and Chen, M. (2018). High efficiency referential genome compression algorithm. *Bioinformatics*, **35**(12), 2058–2065.

Volfovsky, N., Haas, B. J., and Salzberg, S. L. (2001). A clustering method for repeat analysis in DNA sequences. *Genome Biology*, **2**(8), research0027–1.

Wandelt, S. and Leser, U. (2013). FRESCO: Referential compression of highly similar sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **10**(5), 1275–1288.

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

Wang, C. and Zhang, D. (2011). A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Research*, **39**(7), e45–e45.

Wang, J., Wang, W., Li, R., Li, Y., Tian, G., Goodman, L., Fan, W., Zhang, J., Li, J., Zhang, J., *et al.* (2008). The diploid genome sequence of an Asian individual. *Nature*, **456**(7218), 60–65.

Warren, R. L., Keeling, C. I., Yuen, M. M. S., Raymond, A., Taylor, G. A., Vandervalk, B. P., Mohamadi, H., Paulino, D., Chiu, R., Jackman, S. D., *et al.* (2015). Improved white spruce (Picea glauca) genome assemblies and annotation of large gene families of conifer terpenoid and phenolic defense metabolism. *The Plant Journal*, **83**(2), 189–212.

Yao, H., Ji, Y., Li, K., Liu, S., He, J., and Wang, R. (2019). HRCM: an efficient hybrid referential compression method for genomic big data. *BioMed Research International*. doi: 10.1155/2019/3108950.

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.