Bioinformatics, 2017, 1–9
doi: 10.1093/bioinformatics/btx412
Advance Access Publication Date: 23 June 2017
Original Paper

OXFORD

Sequence analysis

High-speed and high-ratio referential genome compression

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

[1]Advanced Analytics Institute, University of Technology Sydney, Broadway, NSW 2007, Australia and [2]School of Computing, National University of Singapore, Singapore 117417

*To whom correspondence should be addressed.

Associate Editor: Inanc Birol

Received on March 15, 2017; revised on May 31, 2017; editorial decision on June 19, 2017; accepted on June 21, 2017

Abstract

Motivation: The rapidly increasing number of genomes generated by high-throughput sequencing platforms and assembly algorithms is accompanied by problems in data storage, compression and communication. Traditional compression algorithms are unable to meet the demand of high compression ratio due to the intrinsic challenging features of DNA sequences such as small alphabet size, frequent repeats and palindromes. Reference-based lossless compression, by which only the differences between two similar genomes are stored, is a promising approach with high compression ratio.

Results: We present a high-performance referential genome compression algorithm named HiRGC. It is based on a 2-bit encoding scheme and an advanced greedy-matching search on a hash table. We compare the performance of HiRGC with four state-of-the-art compression methods on a benchmark dataset of eight human genomes. HiRGC takes <30 min to compress about 21 gigabytes of each set of the seven target genomes into 96–260 megabytes, achieving compression ratios of 217 to 82 times. This performance is at least 1.9 times better than the best competing algorithm on its best case. Our compression speed is also at least 2.9 times faster. HiRGC is stable and robust to deal with different reference genomes. In contrast, the competing methods' performance varies widely on different reference genomes. More experiments on 100 human genomes from the 1000 Genome Project and on genomes of several other species again demonstrate that HiRGC's performance is consistently excellent.

Availability and implementation: The C ++ and Java source codes of our algorithm are freely available for academic and non-commercial use. They can be downloaded from https://github.com/yuansliu/HiRGC.

Contact: jinyan.li@uts.edu.au

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Next-generation sequencing (NGS) technologies have produced huge amounts of sequence reads and many assembled genomes over the past decade (Goodwin *et al.*, 2016). The swift decline of sequencing costs has made whole-genome sequencing commonplace (http://www.genome.gov/sequencingcosts). The accumulated data and the continuing genome generation provide new foundations to gain deeper insights into genomic structures, functions and evolutions (Mardis, 2008). For example, >100 million human genomes are expected to be sequenced by 2025 for precision medicine research (Stephens *et al.*, 2015). As an uncompressed haploid human genome needs roughly three gigabytes (GB) of physical memory (Deorowicz *et al.*, 2015; Ochoa *et al.*, 2015), this big-data has created computational challenges for storage (Kahn, 2011), retrieval (Williams and Zobel, 2002) and privacy (Huang *et al.*, 2016). Efficient and high-ratio data compression is needed to address the

exponential growth of NGS genomes (Numanagić *et al.*, 2016). Traditional text data compression tools such as gzip (http://www.gzip.org/) and bzip2 (http://www.bzip.org/) are inefficient for compressing genome data, because they have not exploited the intrinsic features of DNA sequences such as small alphabet size, frequent repeats and palindromes (Ochoa *et al.*, 2015; Zhu *et al.*, 2013). Recently proposed compression algorithms specialized for genome data have achieved higher compression ratios than the traditional algorithms by making use of such biological features (Zhu *et al.*, 2013; Giancarlo *et al.*, 2014).

Referential genome compression is a preferred option for genome data lossless compression (Wandelt *et al.*, 2014). The key idea of this technique is to store the differences between the target genome and the reference by replacing the common long substrings with the corresponding positions at the reference sequence as well as their lengths. This compression approach is motivated by the high similarity between the target genomes and the reference of the same species, e.g. the >99% similarity for human genomes (Lander *et al.*, 2001). Since the first referential genome compression algorithm DNAZip (Christley *et al.*, 2009) was published, the topic has attracted intensive research interests (Pavlichin *et al.*, 2013; Deorowicz *et al.*, 2013). One main limitation of these algorithms is that the variation data must be provided (Deorowicz *et al.*, 2015; Ochoa *et al.*, 2015). Since genomes can be obtained from different experiments with different reference genomes and also can be *de novo* assembled, the variation data are not always available. This specific setting limits their applications in practice. Here, we focus on referential genome compression when just a reference genome is given (i.e. the variation data is not needed).

Our algorithm HiRGC is a high-performance referential genome compression algorithm. Its speed and compression ratio are both better than the state-of-the-art algorithms (Ochoa *et al.*, 2015; Deorowicz *et al.*, 2015; Saha and Rajasekaran, 2015, 2016). A novel idea in the preprocessing stage is to purify the raw data to get a sequence containing characters only in $\Psi = \{A, C, G, T\}$ by separating the auxiliary information in the FASTA-format file. Then, we use a 2-bit encoding scheme to encode the pure base sequence to a 2-bit integer sequence. These preprocessing steps are different from those used by existing methods. And, to our best knowledge, these preprocessing steps are explicitly described and employed for the first time in referential genome compression. After these preprocessing steps, the subsequent main computation is a greedy matching on a hash table constructed from the reference integer sequence.

The matching strategy is new, and quite different from the mapping generation step of iDoComp (Ochoa *et al.*, 2015) and is also different from the alignment step of ERGC (Saha and Rajasekaran, 2015): (i) Our matching method takes 2-bit integer sequences as input. iDoComp and ERGC have to deal with complex strings constructed from large character sets. (ii) Our matching method runs in a single round. ERGC requires iterations with different lengths of substrings and needs to calculate the edit-distance of the unaligned substrings; iDoComp requires storing matches in memory and then requires combining the consecutive matches, which are very memory- and time-consuming; and GDC-2 (Deorowicz *et al.*, 2015) performs two-level parsing. (iii) Our matching method operates on a global hash table. ERGC uses a partial hash table; (iv) Each match of our algorithm contains at least k integers (a parameter of our algorithm). iDoComp may generate very short matches with only one or two characters. (v) A mismatch element by our algorithm can be an integer or a subsequence. A mismatched character by iDoComp must be related to a match.

The first two differences imply that our greedy matching strategy (a single-round computation without extra cost) is *simpler* than the other methods, contributing to HiRGC's *fast* compression speed. The third point implies that HiRGC can perform better than ERGC when structural variants with translocations exist in genomes. Furthermore, the partitioning strategy of ERGC, which divides long sequences into fixed-size blocks, may result in the splitting of long matches into two or more short ones, whereas HiRGC uses a global-matching approach which has less match information to store and thus saves space. The last two points suggest that iDoComp requires *more space* to store short matches and mismatched characters, whereas HiRGC largely manages to avoid this. Experiments on benchmark genome datasets have confirmed the effectiveness of our new ideas for improving both speed and compression ratio.

2 Related works

The most difficult part of referential genome compression is to map the target genome to the reference genome; i.e. how to optimize a subsequence of the target to be represented by a pair of integers—a position number and the length of the subsequence—with regard to the reference genome. Existing algorithms have used various data structures to generate the mapping. There are three basic categories. The first category exploits the structure of suffix array or suffix tree to find the longest subsequence. For instance, RLZ (Kuruppu *et al.*, 2010) maintains a self-indices structure on the reference sequence, where the self-indices are evolved from a suffix array (Navarro and Mäkinen, 2007). Then, LZ77 parsing (Ziv and Lempel, 1977) is used to find the longest prefix of the target sequence with the help of the self-indices structure. An improved version of RLZ (named RLZ_opt (Kuruppu *et al.*, 2011)) uses a non-greedy parsing approach and suffix array. Particularly, it incorporates a local look-ahead optimization, a short-factor encoding and longest increasing subsequence computation to improve performance. Wandelt and Leser (2012) proposed an algorithm to trade between memory consumption and compression speed. The algorithm divides the reference sequence into blocks of fixed size and then finds the longest prefix match by a depth-first search on a suffix tree constructed from the reference block. FRESCO (Wandelt and Leser, 2013) parses the target sequence to the reference sequence by a compressed suffix tree. Reference selection, reference rewriting and second-order compression techniques are also employed by FRESCO to further improve its compression ratio. iDoComp (Ochoa *et al.*, 2015) parses the target to the reference by a suffix array. Its strategy is similar to Chern's algorithm (Chern *et al.*, 2012), where some adjacent matches are combined into an approximate match and all the mappings are further compressed by an entropy encoder.

The second category of algorithms all use hash tables to parse a target sequence to a reference sequence. GDC (Deorowicz and Grabowski, 2011) is an algorithm similar to RLZ_opt. It first divides a sequence into blocks containing 8192 symbols. Then, GDC performs LZ77 parsing by hashing instead of using suffix array. In addition, GDC selects a reference sequence by a heuristic method and provides fast random access. Xie *et al.* (2015) proposed an algorithm CoGI that conducts alignment based on a local no-collision hash table. The primary difference is that CoGI can be used as a reference-free compression algorithm. GDC-2 (Deorowicz *et al.*, 2015) performs two-level LZ77 parsing to compress a large collection of genomes. Different contextual encoding strategies are employed to encode positions and lengths of the two-level matches. ERGC (Saha and Rajasekaran, 2015) is a greedy compression algorithm. It divides the reference sequence

and the target sequence to fixed-size blocks, and a target block is aligned to a corresponding reference block using a hash table constructed from all k-mers of the reference block. NRGC (Saha and Rajasekaran, 2016) is a recently improved version of ERGC with a scoring-based placement technique.

The third category of algorithms are not related to the above two data structures (suffix array and hash table). RGS (Wang and Zhang, 2011) uses a modified UNIX diff program to find the longest common subsequences. GReEn (Pinho *et al.*, 2012) is a compression tool based on arithmetic coding. It has a copy model related to the reference sequence for predicting the characters. Chern *et al.* (2012) developed an algorithm to generate the mapping by an adaptive sliding window-based string matching technique.

FASTA file is a widely adopted file format for storing genome sequences. It contains an identifier and lines of sequence data. The standard IUB/IUPAC amino acids and nucleic acid codes constitute the sequence data. Any loss of information in the file would cause unpredictable outcome in personalized medicine. GReEn and CoGI ignore the unique sequence identifier. GRS and RLZ-opt only consider limited upper-case letters, i.e. $\Omega = \Psi \cup \{N\}$, and CoGI deals with the letters $\Sigma = \{a, t, g, c, n\}$. Despite the significant enlargement of character sets, the implementation of iDoComp still processes an superset of $(\Omega \cup \Sigma)$ rather than all the legal characters. Some characteristics of these algorithms are summarized in Supplementary Table S1.

Algorithms like GDC, ERGC and NRGC divide long sequences into fixed-size blocks to improve compression speed. Such a partitioning strategy may result in the splitting of long matches into two or more short ones. ERGC searches for matches in a local block and can achieve high compression ratio only when the reference and target sequence are highly similar. When the variation between them is large, ERGC has limitation due to many unmatched subsequences. To mitigate this limitation, NRGC introduces a new technique called scoring-based placement to detect better alignments via a precomputed scoring system. However, it is a complicated system. Users must have prior knowledge about sequence similarity to obtain a high compression ratio.

3 Materials and methods

Our algorithm HiRGC is modularized into three stages: preprocessing, greedy matching and post-processing. A schematic diagram of HiRGC is depicted in Figure 1.

3.1 Preprocessing steps

In this stage, we first separate the FASTA file into two parts: auxiliary information and a sequence that contains letters only in $\Psi = \{A, C, G, T\}$, and then encode the corresponding sequence to a 2-bit integer sequence. The auxiliary information consists of the identifier, the length of short sequences, position intervals of lower-case letters and the letter 'N', and characters not in $\Omega = \Psi \cup \{N\}$ as well as their corresponding positions. We use other characters to represent characters not in Ω unless otherwise specified.

The detailed procedure for preprocessing a given target genome FASTA file is as follows:

Step 1. Read the identifier from this FASTA file and save it as a string *id*.

Step 2. Read the short sequences in each line and save their lengths to an array *seq_len* successively. Then the sequences are concatenated as a long sequence L.

Step 3. Find the intervals of lower-case letters in L and save positions of the beginning of these intervals as well as their lengths to the arrays *low_pos* and *low_len*, respectively. Then, convert lower-case letters to upper-case to obtain a new sequence L_1 consisting entirely of upper-case letters.

Step 4. Find the intervals of the letter 'N' in L_1 and save positions of the beginning of these intervals as well as their lengths to arrays *N_pos* and *N_len*, respectively. Then, remove all N's from L_1 to obtain a sequence L_2.

Step 5. Find all other non-Ω characters in L_2 and save their positions and characters in arrays *oth_pos* and *oth_ch*, respectively. Remove all these characters from L_2 to obtain a sequence L_3.

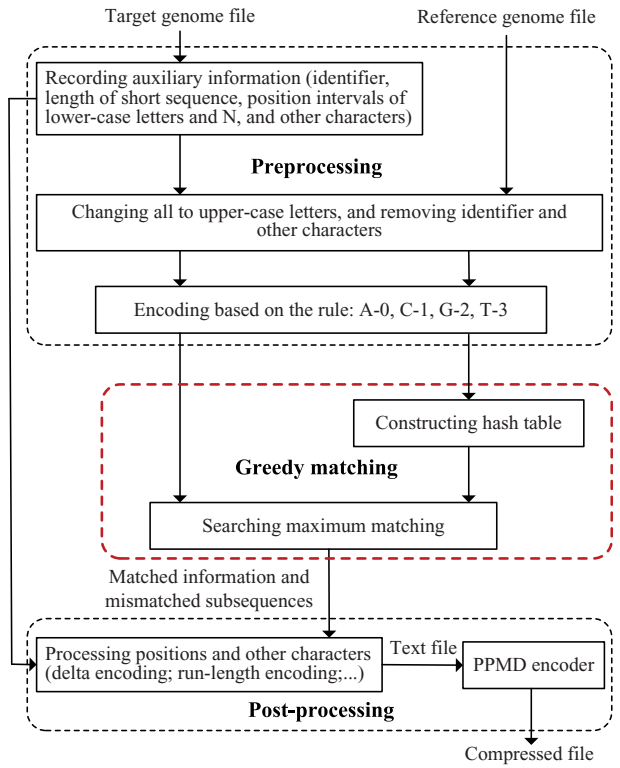

Fig. 1. Schematic diagram of our algorithm HiRGC

After these, the auxiliary information of the target genome is represented as $\Delta = \{id, seq\_len, low\_pos, low\_len, N\_pos, N\_len, oth\_pos, oth\_ch\}$, which is required during decompression. Later, we use some advanced encoding techniques to compress Δ in the post-processing stage.

Similarly, for the reference genome FASTA file, we obtain a sequence containing characters only in Ψ for the next stage. Note that the auxiliary information of the reference genome file can be discarded. The above preprocessing stage reduces the alphabet size significantly, and makes it possible to encode every character by a 2-bit scheme.

We use a 2-bit encoding method to translate genome sequences into 2-bit integer sequences for referential genome compression. By 2-bit encoding, each nucleotide A, C, G and T is encoded as a 2-bit integer. Considering the Watson-Crick base-pairing rule and that 0 and 1 are complementary in the binary system, the following 2-bit encoding

$$E(x) = \begin{cases} 0; & \text{if } x = A; \\ 1; & \text{if } x = C; \\ 2; & \text{if } x = G; \\ 3; & \text{if } x = T; \end{cases}$$

translates our preprocessed sequences (containing characters only in Ψ) into 2-bit integer sequences.

For such an integer sequence, we use k-tuple to denote k contiguous integers. Given an integer sequence $\{u(i)\}_{i=1}^{n}$, where $u(i) \in \{0,1,2,3\}$, there exist $(n-k+1)$ overlapping k-tuples. We use an integer of $(2 \times k)$ bits, called a tuple value, to represent the i-th tuple. It is computed by

$$V_i^u = \sum_{j=0}^{k-1} \left(u(i+j) \times 4^j\right), \tag{1}$$

where $i \in [1, n-k+1]$.

3.2 Advanced greedy matching in a global hash table

First, a hash table is constructed from the tuple values of a reference sequence. It consists of two data structures: an array of entries $p(\cdot)$ and an array of header pointers $h(\cdot)$. We assume that an entry in the hash table contains a tuple, position of this tuple, its hash value and a next pointer. In our implementation, only the next pointer is stored. The entry array links the entries which have the same hash value by the next pointer. An element in the header pointer array and the chain of entries it points to in the array of entries form a bucket that stores all the entries having the same hash value. The details of this step are described in lines 1–4 of Algorithm 1.

In the hash table, our greedy matching finds the maximum match starting at position 1 of the target sequence. Suppose we are processing the i-th tuple in the target sequence. We first calculate the corresponding tuple value V_i^t. With the corresponding hash value $\left(V_i^t \bmod s\right)$, we detect its corresponding bucket that stores the entries having a hash value equal to $\left(V_i^t \bmod s\right)$. If there is no entry, the i-th tuple in the target sequence is a mismatched one, and skip to the $(i+1)$-th tuple. For each entry in this bucket, if its tuple is identical to the current tuple, we extend this match until a mismatched integer is met. The longest one is the final match. If there are more than one longest matches, we select the first one. The subsequence between the last match and the i-th is a mismatched subsequence. The position and length of the match is the matched information. Then, we skip all the tuples in this match. Since most variations between genomes are single nucleotide polymorphism, the integer after a match is always treated as unmatched. Lines 5–26 of Algorithm 1 state the details of this procedure.

Example 1. Suppose we are given the following reference sequence and target sequence:

Reference : 0032 3022301003002021

Target : 00321022301213002022

The output of our matching algorithm is

$$\mathcal{M} = [(1,4),(1),(6,6),(21),(14,5),(0),(16,2)]$$

where a match is represented by a pair of values representing the position and length, and a mismatch is represented by a subsequence. The reference sequence and $\mathcal{M}$ suffice to reconstruct the target sequence.

3.3 Post-processing

We use delta encoding (Smith *et al.*, 1997) to encode most of the data derived the first three stages, including the positions of matches, position intervals of lower-case letters and the letter 'N', and positions of other characters. Furthermore, run-length encoding (Held and Marshall, 1991) is employed to encode the lengths of short sequences. The other characters are mapped to digit numbers from 0. For example, if there are three other characters, say K, M and S, these three are converted to integers 0, 1 and 2, respectively. If the other characters are <10 in type, we save all these characters as a string. Otherwise, we combine them as a sequence of 32-bit integers. Then, the auxiliary information, matched information and the mismatched subsequences are stored as an ASCII text file.

We use the PPMD compression algorithm, a variant of prediction by partial matching data compression algorithm (Cleary and Witten, 1984; Moffat, 1990), to compress the text file. 7-zip (http://www.7-zip.org/) provides an implementation of PPMD and we use it in this stage. When there are more than one chromosomes, we compress their text files as a whole.

Algorithm 1. The hash-based greedy matching

Input: a reference sequence $R = \{r(i)\}_{i=1}^{n_r}$, a target sequence $T = \{t(i)\}_{i=1}^{n_t}$, size of hash table s and the tuple length k.
Initialize the header pointer, that $h(i) = 0$ for $\forall i \in \{1, \ldots, n_r\}$.
1: **for** $i = 1$ **to** $(n_r - k + 1)$ **do**
2: Calculate $V_i^r = \sum_{j=0}^{k-1} (r(i+j) \times 4^j)$.
3: Update the hash table:
$$p(i) = h(V_i^r \bmod s), h(V_i^r \bmod s) = i.$$
4: **end for**
5: Set $i = 1$ and $p^* = 1$.
6: **repeat**
7: Calculate $V_i^t = \sum_{j=0}^{k-1} (t(i+j) \times 4^j)$.
8: Set $j = h(V_i^t \bmod s)$, $p_{\max} = 0$ and $l_{\max} = 0$.
9: **while** $j \neq 0$ **do**
10: Set $l = 0$.
11: **while** $r(j+l) = t(i+l)$ **do**
12: Update $l = l + 1$.
13: **end while**
14: **if** $l \geq k$ **and** $l > l_{\max}$ **then**
15: $p_{\max} = j$, $l_{\max} = l$.
16: **end if**
17: Update $j = p(j)$.
18: **end while**
19: **if** $l_{\max} > 0$ **then**
20: Subsequence $t(p^*, \ldots, i-1)$ is a mismatched one.
21: $(p_{\max}, l_{\max})$ is the matched information.
22: Update $p^* = i + l_{\max}$.
23: **end if**
24: Update $i = i + l_{\max} + 1$.
25: **until** $i > (n_t - k + 1)$
26: The remaining subsequence is a mismatched subsequence.
Output: matched information and mismatched subsequences.

3.4 Decompression

Usually referential compression and decompression are asymmetric. Our decompression procedure is just a simple reversion of the compressed one except that the subsequences are recovered from the reference sequence via the positions and lengths of the matches. PPMD first decompresses the final 7z file. Since PPMD is a lossless compression algorithm (Cleary and Witten, 1984), and both delta encoding (Smith *et al.*, 1997) and run-length encoding (Held and Marshall, 1991) are invertible transformation, HiRGC can recover the auxiliary information, matched information and the mismatched subsequences without any information loss during decompression. According to the greedy matching procedure, the reference

sequence, the matched information and the mismatched subsequences can be used to precisely reconstruct the target sequence. The 2-bit encoding is an invertible operation as well. Thus, the sequence L_3 can be exactly recovered. As the auxiliary information records all the necessary information of changes we made on the original target genome FASTA file, we can precisely reconstruct the original FASTA file by combing the auxiliary information and the sequence L_3. Therefore, HiRGC is a lossless compression algorithm.

4 Results and performance analysis

We report compression ratio, speed, and memory usage of the algorithms in this section. The performance of our algorithm HiRGC is compared to four recently published algorithms: GDC-2 (Deorowicz *et al.*, 2015), iDoComp (Ochoa *et al.*, 2015), ERGC (Saha and Rajasekaran, 2015), and NRGC (Saha and Rajasekaran, 2016). All the experiments were carried out on a computing cluster running Red Hat Enterprise Linux 6.7 (64 bit) with 2 × 3.33 GHz Intel Xeon X5680 (6 cores) and 48 GB RAM.

4.1 Genome datasets and their disk file size

We thoroughly compared the performance of the five algorithms on eight human genomes (Homo sapiens): hg38, hg19, hg18, hg17, the genome of J. Craig Venter (HuRef) (Levy *et al.*, 2007), the genome of a Han Chinese known as YH (Wang *et al.*, 2008), KOREF_20090131 (denoted by K131), and KOREF_20090224 (denoted by K224) (Ahn *et al.*, 2009). Data of these genomes are complete, all containing chromosomes 1–22 and the X and Y sex-chromosomes. Each of the eight datasets has a disk file size of around 3.0 GB. All these genomes are benchmark genomes widely used in the performance evaluation of various state-of-the-art algorithms for referential genome compression (Pinho *et al.*, 2012; Ochoa *et al.*, 2015; Saha and Rajasekaran, 2016; Deorowicz *et al.*, 2016). Some other information of these genomes are provided in Supplementary Table S2.

In addition, we compared the five algorithms on 100 human genomes randomly selected from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015) with the help of the script vcf2fasta provided by GDC-2 (Deorowicz *et al.*, 2015). Finally, genomes of some other species such as three versions of *Caenorhabditis elegans* (viz. ce6, ce10 and ce11), two versions of *Saccharomyces cerevisiae* (viz. sacCer2 and sacCer3), two versions of *Arabidopsis thaliana* (viz. TAIR9 and TAIR10) and three versions of *Oryza sativa* (viz. TIGR5.0, TIGR6.0 and TIGR7.0), are also used to assess the performance of the five algorithms. The disk file sizes of these genomes range from tens to hundreds megabytes (MB), much smaller than human genomes.

4.2 High compression ratios on the eight benchmark human genomes

To test the robustness and effectiveness of the algorithms, each of the eight benchmark human genomes was taken iteratively as the reference genome and the remaining seven were all considered as target genomes (i.e. no bias in the target genome selection). In total, $8 \times 7 = 56$ reference-target genome pairs were used to test every algorithm. For each reference genome, the total file size of the seven target genomes is around 21.0 GB. The reference genome is not included in any compressed file as per tradition (Ochoa *et al.*, 2015; Saha and Rajasekaran, 2015, 2016).

Our HiRGC algorithm could compress the 21.0 GB target genomes to a file size <260 megabytes (MB) in the worst case when HuRef was set as the reference genome. While for the other reference genomes, the HiRGC compressed files are all <130 MB. These compressed files are about 82 to 217 times smaller than the original files. However, most of the compressed files by GDC-2 are at least one order of magnitude larger than ours except for one case (6 times); cf. the last row of Table 1. All of the compressed files by ERGC are at least one order of magnitude (up to 21 times) larger than ours. The compressed files by NRGC are at least 9.2 times larger than ours, and those by iDoComp are at least 1.9 times (up to 7.7 times) larger.

Detailed results on individual reference-target genome pairs show that HiRGC can indeed very often make huge compression improvement. For the other cases, HiRGC has competitive or close performance with the state-of-the-art algorithms. HiRGC did not make any obviously worse (5 MB less) performance than the existing methods except on one pair (viz. when hg18 is set as reference and hg17 as target); cf. Table 2. In fact, HiRGC achieved better results than GDC-2 in 54 cases. In particular, on 32 cases, HiRGC delivered one order of magnitude improvement. HiRGC performed better than NRGC on 53 of the 56 cases; on the remaining three cases, HiRGC and NRGC were very close (within 2.5 MB range in performance). NRGC exhibited extremely poor performance when hg38 or HuRef was set as the reference or the target genome—NRGC even sometimes could not compress. Compared with iDoComp, HiRGC achieved better performance on 44 cases; on 8 other cases, their performance were competitive (within 5 MB range); on the remaining four reference-target genome pairs when

Table 1. Overall comparison of compressed size (MB) and relative compression gain for different algorithms under different reference genomes

Reference	Target set size (MB)	Compressed file size (MB) by					Relative compression gain			
		GDC-2	iDoComp	ERGC	NRGC	HiRGC	GDC-2 (%)	iDoComp (%)	ERGC (%)	NRGC (%)
YH	20, 897	1508.37	255.75	1660.97	1831.10	**128.74**	91	49	92	92
hg17	20, 887	1631.53	\	2158.26	2044.15	**102.82**	93	\	95	94
hg18	20, 887	1630.61	392.85	1475.71	898.21	**96.89**	94	74	94	86
hg19	20, 872	1684.31	469.58	1897.89	1194.08	**95.73**	94	79	94	91
hg38	20, 879	1732.57	483.82	2020.22	1289.24	**95.73**	94	80	95	92
K131	20, 897	1489.35	961.54	1801.59	1456.77	**124.10**	91	87	93	91
K224	20, 897	1458.61	924.78	1827.6	1610.96	**124.80**	91	86	93	92
HuRef	20, 972	1606.78	846.88	4280.67	4812.53	**251.23**	84	70	94	94

Note: iDoComp could not compress some reference-target genome pairs within 24 h. A '\' means that the result was not available within 24 h.

Table 2. Detailed comparison between HiRGC and the state-of-the-art methods

Reference	Target	File size (MB) after compressed by				
		GDC-2	iDoComp	ERGC	NRGC	HiRGC
YH	hg17	338.45	51.02	125.06	600.79	**17.64**
	hg18	340.95	37.76	**17.02**	27.80	17.55
	hg19	354.17	40.67	433.31	33.56	**18.86**
	hg38	376.52	64.84	580.42	492.59	**27.04**
	K131	45.08	20.25	**12.73**	27.50	17.61
	K224	39.56	18.12	**11.27**	26.64	16.06
	HuRef	**13.64**	22.09	481.16	622.22	13.98
hg17	YH	379.02	\	202.57	649.45	**12.92**
	hg18	11.72	\	171.65	**9.83**	10.34
	hg19	34.72	\	379.77	17.38	**11.65**
	hg38	66.90	\	580.82	666.40	**19.71**
	K131	388.11	367.14	174.89	36.36	**18.74**
	K224	384.78	363.79	187.09	37.22	**17.29**
	HuRef	366.28	406.66	461.47	627.41	**12.17**
hg18	YH	380.47	65.89	**7.35**	27.25	11.94
	hg17	10.68	**2.28**	87.46	9.83	9.43
	hg19	29.11	**6.20**	299.32	16.03	10.67
	hg38	66.95	28.13	582.17	596.73	**18.99**
	K131	389.42	76.45	**13.15**	36.36	17.79
	K224	386.10	74.96	**11.73**	37.22	16.34
	HuRef	367.88	138.94	474.53	174.79	**11.73**
hg19	YH	393.59	80.99	190.40	31.31	**12.18**
	hg17	33.47	**7.69**	175.57	19.08	9.70
	hg18	28.49	**5.28**	131.21	13.35	9.60
	hg38	47.13	24.16	526.10	430.69	**18.15**
	K131	402.18	90.29	248.04	42.41	**18.01**
	K224	398.88	89.06	268.14	41.41	**16.57**
	HuRef	380.57	172.11	358.43	615.82	**11.52**
hg38	YH	398.00	70.40	268.16	79.76	**13.65**
	hg17	54.58	18.02	332.98	209.31	**11.09**
	hg18	52.06	11.47	266.33	82.75	**11.26**
	hg19	32.83	**8.46**	191.03	63.09	11.55
	K131	407.11	79.93	342.02	106.22	**19.41**
	K224	403.78	78.45	339.56	97.51	**17.96**
	HuRef	384.21	217.09	280.14	650.60	**10.81**
K131	YH	43.13	32.65	**8.77**	22.26	13.34
	hg17	335.06	210.16	231.46	177.29	**19.20**
	hg18	337.47	211.59	**18.75**	30.90	19.16
	hg19	350.61	217.01	442.22	37.57	**20.44**
	hg38	370.90	237.06	603.73	545.48	**28.56**
	K224	16.18	6.93	**4.52**	7.14	8.88
	HuRef	36.00	46.14	491.85	636.13	**14.52**
K224	YH	35.68	29.31	**8.59**	21.87	13.14
	hg17	331.58	206.41	258.27	248.02	**19.12**
	hg18	333.98	207.85	**18.60**	30.83	19.06
	hg19	347.22	213.33	443.40	34.96	**20.36**
	hg38	367.54	233.51	609.79	614.73	**28.48**
	K131	13.20	6.68	**5.79**	7.77	10.20
	HuRef	29.41	27.68	483.16	652.78	**14.44**
HuRef	YH	**28.16**	29.36	599.02	660.69	31.70
	HG17	351.94	185.08	622.93	673.49	**34.39**
	hg18	355.11	162.28	608.91	673.49	**35.20**
	hg19	367.68	151.31	624.43	696.29	**36.10**
	hg38	388.72	171.52	619.68	713.23	**41.79**
	K131	58.70	94.09	603.52	698.02	**36.78**
	K224	56.47	53.24	602.18	697.32	**35.27**

Note: Bold indicates the best result in the row. A shadowed text, called the second best result, indicates the difference between this performance and the best is < 5 MB. A '\' means that the result was not available within 24 h.

hg17 was set as reference, iDoComp could not finish compressing the files within 24 h. HiRGC was better than ERGC on 44 of the 56 target genomes (by one order of magnitude mostly); on the remaining 12 cases, the compression performance of HiRGC and ERGC were close to each other within a 5 MB range. Table 2 also lists the best and the second best algorithms on every target genome. HiRGC won 35 times as the best algorithm, and was essentially the second best in all the other cases except one. ERGC won 12 times as the best algorithms; iDoComp won five times as the best algorithm and four times as the second best algorithms; NRGC and GDC-2 got only one and two best cases, respectively.

A box plot of the sizes of the 56 compressed files by each algorithm is shown in Figure 2. The size variation of the compressed files by the four state-of-the-art methods is in a large range (e.g. from 2.28 to 406.66 MB by iDoComp; from 5.79 to 624.43 MB by ERGC). In contrast, the output file size of HiRGC is very stable, ranging from 8.88 to 41.79 MB over the 56 cases. Thus, our algorithm is so robust that it can handle complex input datasets, where the state-of-the-art methods cannot compress well. The average sizes of all the 56 compressed files by the five algorithms are also plotted in Figure 2. Our algorithm HiRGC obtained an average size of 18.3 MB, which is 12.5, 16.6 and 14.8 times improvement over GDC-2's 227.5 MB, ERGC's 303.7 MB and NRGC's 279.3 MB, respectively. iDoComp achieved an average size 83.3 MB (ignoring the four uncompleted cases), 5.9 times larger than HiRGC's.

We also assessed the relative compression gain by comparing our algorithm with the state-of-the-art methods; cf. Table 1. The relative compression gain is defined as:

$$\text{Gain} = \left(1 - \frac{\text{Compressed size of HiRGC}}{\text{Compressed size of other method}}\right) \times 100\%.$$

The gain quantifies the improvement of our method compared against existing ones. For example, our algorithm achieved the gain of 91% against GDC-2 under the reference genome YH. It means that the size of the compressed file using our method is only $(1 - 91\%) = 9\%$ of that of GDC-2. The relative compression gain shows that the percentage improvement HiRGC achieved with respect to GDC-2 and ERGC is at least 91%. Comparing with iDoComp, our HiRGC obtained relative compression gain varying from 49% to 87%. The improvement over NRGC by HiRGC is >86%.

For HiRGC, only one parameter (the tuple length k) influences compression result. To explore the best choice of k, we tuned k within the range [10, 25]. The compression ratio is used to measure compression performance. We used YH and hg18 as the reference genome, and tested eight pairs of reference-target genomes under different values of k. The effect of the tuple length k on the compression ratio is shown in Supplementary Figure S1. HiRGC achieved the best compression ratio on different pairs of reference-target genomes when k was set as 20. When k was smaller than 20, lower performance was observed. There was also a slightly decrease in performance when k was set bigger than 20 in most cases. Therefore we set $k = 20$ here and also recommend this to users.

We ran all the state-of-the-art algorithms with their default parameters. However, we found that NRGC failed to compress some chromosomes under default parameters. Following the instruction of NRGC, we increased the division length which is one of the parameters of NRGC. Because the parameter setting is complex, we cannot guarantee finding the optimum. For a chromosome, if we could not find a proper division length, we had no other choice but to assert that NRGC could not compress it. We then

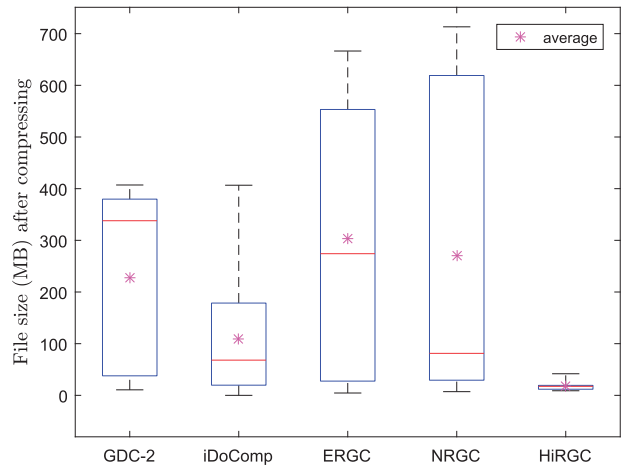

Fig. 2. Box plots of compressed file sizes by different methods

simply compressed the original file with the PPMD compression algorithm.

4.3 Time complexity and speed performance

Speed is another important factor to measure the performance of a compression algorithm. For HiRGC, only the size of hash table affects compression speed when the tuple length k is fixed. If the size of hash table s is set to $2^{2\times 20}$, the buckets will have no hash collision. Therefore, each bucket has the minimum number of entries. But, considering the limited RAM, we need to use a small s. It is difficult to work out the best size of hash table from theory due to the non-uniform distribution of DNA sequences. We performed some experiments on different reference-target pairs with a different s ranging from 26 to 32. The results are shown in Supplementary Figure S2. We found that HiRGC has a good compression speed when the size of hash table s is set to 2^{30}. In the following, we first conduct a complexity analysis of HiRGC, and then compare the running speed of HiRGC with other algorithms.

The time complexity of the first and the third stage of HiRGC are linear. Here, we focus on the time complexity analysis of the greedy matching stage. Let n_r and n_t represent the length of reference sequence and target sequence after preprocessing. First, the complexity of constructing a hash table is linear time, taking $O(n_r)$ time. Let $\widehat{n}$ represent the number of mismatched integers. Then the length of total matched subsequence is $(n_t - \widehat{n})$. Let m_i represent the length of the i-th matched subsequence and $\tilde{n}$ denote the number of matched subsequences. Then, it can be derived that $\sum_{i=1}^{\tilde{n}} m_i = (n_t - \widehat{n})$. Let h_i be the number of entries of the selected buckets related to the i-th matching subsequence. Then, the number of elementary operations of the greedy matching process is

$$\widehat{n} + \sum_{i=1}^{\tilde{n}} h_i \times m_i \le \widehat{n} + \max_i\{h_i\} \times \sum_{i=1}^{\tilde{n}} m_i$$
$$\le \max_i\{h_i\} \times \left(\widehat{n} + \sum_{i=1}^{\tilde{n}} m_i\right) = \max_i\{h_i\} \times n_t.$$

Therefore, the best-case time complexity is $O(n_t)$ when $h_i = 1$, and the worst-case time complexity is $O(d \times n_t)$, where d is the maximum value of h_i. Because frequent repeats exist in DNA sequence, there are many identical k-tuples. In our datasets, the maximum value of d can reach a hundred thousands. In practice, we do not meet the extreme case where all values of h_i are very large. Next, we analyze the average time complexity of our algorithm on real genomes.

Our above analysis has already indicated that the number of entries of each bucket can significantly affect the time complexity. To investigate the distribution of entries, we count the number of entries in different buckets in the hash table constructed from chromosome 1 of some real genome (chromosome 1 is the longest chromosome). We calculate the percentage of number of buckets for different thresholds as follows:

$$P(x) = \frac{\text{Number of bucket having at most } x \text{ entries}}{\text{Total number of bucket having at least 1 entry}} \times 100\%.$$

Because $P(x)$ is observed to increase slowly, we only plot data between 1 and 15; this should not affect our analysis and conclusion. The result shows that about 75% of the buckets has no collision and >90% of the buckets has at most 2 entries. When the threshold increases to 15, only about 3‰ of the buckets has >15 entries. We conclude that most of buckets have very few entries in practice. We then calculate the average number of entries, which is about 1.7 entries per bucket. Finally, the average time complexity of our greedy matching algorithm is estimated as $O(1.7 \times n_t)$, which is much better than the worst-case. The result is depicted in Supplementary Figure S3.

The actual compression speed of our algorithm is significantly faster than current state-of-the-art methods. To minimise the influence of programming languages, we provide the result for both C++ and Java implementations of our algorithm. They had similar speed performance. The compression time of our algorithm was <7 min when using C++ (for most of the cases, <5 min). For GDC-2, the fastest compression time was about half an hour, and on some datasets it took >3 h. Since the time to generate the suffix array is expensive, iDoComp spent about half an hour in most cases. There were four cases which could not be compressed by iDoComp within 24 h. ERGC achieved unstable compression speeds from 10 min to half an hour. It heavily relies on the similarity between the reference genome and the target genome. When NRGC could not compress some chromosomes under the default parameter, it spent much time to choose suitable parameters for those chromosomes. Detailed compression time by the different methods is given in Supplementary Table S3.

As described in Section 3.4, the decompression procedure is much simpler than the compression procedure, and it is a linear time operation. Because NRGC could not compress many chromosomes, the decompression time of NRGC is omitted here. The comparison of decompression time is shown in Supplementary Table S4. We observed that GDC-2 and HiRGC performed better than the other methods. GDC-2 and HiRGC achieved stable decompression time, whereas iDoComp heavily depended on the dataset. HiRGC was faster than these methods for most of the cases.

The authors of iDoComp pointed out that the suffix array can be saved once it is generated and can be reused many times. In our algorithm, the hash table obtained from a reference genome can also be reused to compress many target genomes. We ran another experiment and the comparison result is reported in Supplementary Table S5. For this comparison, iDoComp did not count the time of generating the suffix array, and loaded the suffix array from disk, while our algorithm created the hash table once and compressed the other seven genomes. Though the time required to generate the suffix array was ignored, iDoComp still took about more than an hour to compress the genome set. Our algorithm does not consider saving the hash table because it can be generated very fast. From Supplementary Table S5, we see that HiRGC needed <30 min to

compress a genome set containing seven genomes of about 21 GB. Moreover, HiRGC spent <20 min to decompress the genome set, while the decompression time by iDoComp was three times more than HiRGC.

4.4 Memory usage by HiRGC

Storage of the hash table requires $(4 \times n_t + 2^{32})$ bytes of RAM, where the first and second terms account for the memory requirements of the entry array and the header pointer array, respectively. For the longest chromosome, we have $n_t < 2^{28}$. The highest RAM used to store the hash table is <5 GB. Moreover, we require storage for the two integer sequences and the auxiliary information. About 2 GB memories are used in practice. It should be pointed out that none of the positions, the lengths of matches or the mismatched subsequences need to be stored in RAM. Thus, HiRGC can be suitable for the personal laptop with 8 GB of RAM.

4.5 Compressing 100 genomes from the 1000 Genomes Project

We used HG38 as the reference genome to compress 100 genomes randomly selected from the 1000 Genomes Project. The ID numbers of the 100 genomes are listed at Supplementary Table S6. The raw size of the 100 genomes is about 292 GB. ERGC could not compress these genome datasets, because an infinite loop occurred when compressing chromosome 1. GDC-2 could not compress this genome set within 120 h even when 16 threads were used. iDoComp compressed the 100 genomes to 9.7 GB in 1519 min. HiRGC took 210 min to compress the 100 genomes into a file size of 711 MB. The decompression time by iDoComp and HiRGC was 886 and 109 min, respectively. Overall, our performance is much better than the existing methods on compression ratio, compression speed, and decompression time for these 100 genomes.

4.6 Performance on compressing the genomes of some plants and microbial species

The file size of these genomes are much smaller than human genomes. The algorithms were tested on 16 pairs of reference-target genomes. All the algorithms have competitive or close performance except ERGC and NRGC which did not perform well for the genomes of *Oryza sativa*. ERGC, GDC-2, HiRGC and iDoComp achieved 2, 3, 4 and 7 best cases, respectively. Detailed compression results are reported at Supplementary Table S7. A separate issue is that iDoComp could not achieve *lossless* decompression for some cases (see the '?' symbol in Supplementary Table S7). This also happened to iDoComp for some reference-target pairs of the eight benchmark genomes (see the '?' symbol in Supplementary Table S4).

5 Conclusion and future work

We have proposed a novel algorithm HiRGC for referential genome compression. The key idea is to search maximum matches from the integer sequence on a hash table by an advanced greedy matching strategy. The compression results obtained from real benchmark datasets show that HiRGC significantly outperforms the state-of-the-art algorithms in terms of compression ratio. In addition, analysis of the time complexity on real datasets shows that the average time complexity of our algorithm is linear in practice; and HiRGC has achieved substantial improvement on compression speed. HiRGC is also stable and robust to deal with different reference genomes.

Although HiRGC has achieved high compression ratio and fast compression/decompression speed, there are some other aspects that are worth consideration for further improvement. First, HiRGC is currently not capable of verifying the reference genome provided by users. To verify the reference genome, there are two possible solutions: (i) Store some information about the reference sequence in the compressed file; or (ii) store a unique ID in both the reference file and the compressed file. Second, HiRGC does not exploit complex structural variants with inversions. If we make full use of such complex structural variants in the greedy matching, a further improvement on compression ratio is possible. Last, HiRGC does not support indexing and searching for the compressed files. There are some research works proposed for searching and indexing on large genome collections (Danek *et al.*, 2014; Holley *et al.*, 2016). It is interesting that little compression ratio can be sacrificed to achieve the high speed search in the compressed files (Wandelt *et al.*, 2013). For example, the algorithm GDC (Deorowicz and Grabowski, 2011) can support random access. We will investigate these issues to provide new functionalities on top of current HiRGC.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of our manuscript. We are also grateful to Dr Yu Zhang of City University of Hong Kong for his insightful discussion.

Funding

This work was partly supported by Australia Research Council Discovery Project DP130102124.

Conflict of Interest: none declared.

References

Ahn,S.-M. *et al.* (2009) The first Korean genome sequence and analysis: full genome sequencing for a socio-ethnic group. *Genome Res.*, **19**, 1622–1629.

Chern,B. *et al.* (2012) Reference based genome compression. *Information Theory Workshop (ITW), 2012 IEEE*, 427–431.

Christley,S. *et al.* (2009) Human genomes as email attachments. *Bioinformatics*, **25**, 274–275.

Cleary,J., and Witten,I. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **32**, 396–402.

Danek,A. *et al.* (2014) Indexes of large genome collections on a PC. *PLoS One*, **9**, e109384.

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

Deorowicz,S. *et al.* (2013) Genome compression: a novel approach for large collections. *Bioinformatics*, **29**, 2572–2578.

Deorowicz,S. *et al.* (2015) GDC 2: compression of large collections of genomes. *Sci. Rep.*, **5**, 11565.

Deorowicz,S. *et al.* (2016) Comment on: 'ERGC: An efficient referential genome compression algorithm'. *Bioinformatics*, **32**, 1115–1117.

Giancarlo,R. *et al.* (2014) Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Brief. Bioinformatics*, **15**, 390–406.

Goodwin,S. *et al.* (2016) Coming of age: ten years of next-generation sequencing technologies. *Nat. Rev. Genet.*, **17**, 333–351.

Held,G., and Marshall,T. (1991). *Data Compression; Techniques and Applications: Hardware and Software Considerations*. John Wiley and Sons, Inc., New York.

Holley,G. *et al.* (2016) Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.*, **11**, 3.

Huang,Z. *et al.* (2016) A privacy-preserving solution for compressed storage and selective retrieval of genomic data. *Genome Res.*, **26**, 1687–1696.

Kahn,S.D. (2011) On the future of genomic data. *Science*, **331**, 728–729.

Kuruppu,S. *et al*. (2010) Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *International Symposium on String Processing and Information Retrieval*, pp. 201–206. Springer.

Kuruppu,S. *et al*. (2011) Optimized relative Lempel-Ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference-Volume 113*, pp. 91–98. Australian Computer Society, Inc.

Lander,E.S. *et al*. (2001) Initial sequencing and analysis of the human genome. *Nature*, **409**, 860–921.

Levy,S. *et al*. (2007) The diploid genome sequence of an individual human. *PLoS Biol*., **5**, e254.

Mardis,E.R. (2008) The impact of next-generation sequencing technology on genetics. *Trends Genet*., **24**, 133–141.

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

Moffat,A. (1990) Implementing the PPM data compression scheme. *IEEE Trans. Commun*., **38**, 1917–1921.

Navarro,G., and Mäkinen,V. (2007) Compressed full-text indexes. *ACM Comput. Surv*., **39**, 2.

Numanagić,I. *et al*. (2016) Comparison of high-throughput sequencing data compression tools. *Nat. Methods*, **13**, 1005–1008.

Ochoa,I. *et al*. (2015) iDoComp: a compression scheme for assembled genomes. *Bioinformatics*, **31**, 626–633.

Pavlichin,D.S. *et al*. (2013) The human genome contracts again. *Bioinformatics*, **29**, 2199–2202.

Pinho,A.J. *et al*. (2012) GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Res*., **40**, e27–e27.

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

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

Smith,S.W. *et al*. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA.

Stephens,Z.D. *et al*. (2015) Big data: astronomical or genomical?. *PLoS Biol*., **13**, e1002195.

Wandelt,S., and Leser,U. (2012) Adaptive efficient compression of genomes. *Algorithms Mol. Biol*., **7**, 30.

Wandelt,S., and Leser,U. (2013) FRESCO: Referential compression of highly similar sequences. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, **10**, 1275–1288.

Wandelt,S. *et al*. (2013) RCSI: scalable similarity search in thousand(*s) Genomes. In Proceedings of the VLDB Endowment*, Vol. **6**, pp. 1534–1545.

Wandelt,S. *et al*. (2014) Trends in genome compression. *Curr. Bioinformatics*, **9**, 315–326.

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

Wang,J. *et al*. (2008) The diploid genome sequence of an Asian individual. *Nature*, **456**, 60–65.

Williams,H.E., and Zobel,J. (2002) Indexing and retrieval for genomic databases. *IEEE Trans. Knowl Data Eng*., **14**, 63–78.

Xie,X. *et al*. (2015) CoGI: towards compressing genomes as an image. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, **12**, 1275–1285.

Zhu,Z. *et al*. (2013) High-throughput DNA sequence data compression. *Brief. Bioinformatics*, **16**, 1–15.

Ziv,J., and Lempel,A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, **23**, 337–343.