SCIENTIFIC REP✸RTS

GDC 2: Compression of large collections of genomes

Sebastian Deorowicz[1], Agnieszka Danek[1] & Marcin Niemiec[2]

The fall of prices of the high-throughput genome sequencing changes the landscape of modern genomics. A number of large scale projects aimed at sequencing many human genomes are in progress. Genome sequencing also becomes an important aid in the personalized medicine. One of the significant side effects of this change is a necessity of storage and transfer of huge amounts of genomic data. In this paper we deal with the problem of compression of large collections of complete genomic sequences. We propose an algorithm that is able to compress the collection of 1092 human diploid genomes about 9,500 times. This result is about 4 times better than what is offered by the other existing compressors. Moreover, our algorithm is very fast as it processes the data with speed 200 MB/s on a modern workstation. In a consequence the proposed algorithm allows storing the complete genomic collections at low cost, e.g., the examined collection of 1092 human genomes needs only about 700 MB when compressed, what can be compared to about 6.7 TB of uncompressed FASTA files. The source code is available at http://sun.aei.polsl.pl/REFRESH/index.php?page=projects&project=gdc&subpage=about.

The genome sequencing technology has recently become so cheap that it started to be considered as a useful tool in medicine. Companies like Illumina offer whole human genome sequencing for medical purposes for five thousand U.S. dollars[1]. There are also large scale projects designed to find the common differences between individual genomes. One of the most famous is the 1000 Genome Project[2] which aims at sequencing the genomes of several thousand humans and determining the genetic variants with at least 1% frequency. There are, however, even broader attempts for human genome sequencing, to mention the UK10K project[3], the Personal Genomes Project[4], and the Million Veteran Project (MVP)[5]. The planned number of sequenced genomes are 10 K, 100 K, and 1 M, respectively. Large collections of genomes are built also for other species. E.g., in the 1001 Genomes Project (1001 GP)[6,7] about 1000 of genomes of *Arabidopsis thaliana* are to be sequenced.

The sequencing is of course challenging, but due to the large amounts of produced data, the pure storage and transfer of the results becomes a challenge too. The recent papers[8,9] show that the IT costs are (or will be soon) comparable to the sequencing costs. Due to the slow progress in reducing the IT prices, the effective ways of representing genomic data in compact form are intensively investigated. Several subproblems can be identified here. The first is the compression of raw sequencing reads[10–13] The second is the compression of reads after mapping onto reference genomes[10,14,15]. The third is the compression of results of variant calling[16–18]. The fourth is the compression of complete genomic sequences[19–22]. These subproblems are related, nevertheless require different approaches. The recent surveys discuss most of the existing algorithms[9,23,24].

In this paper we deal with the last of the mentioned tasks, i.e., storage of collections of genomes. We propose Genome Differential Compressor 2 (GDC 2), a utility for compression of large sets of genomes of the same species. Since such genomes are highly similar, e.g., it was estimated that two humans have their genomes identical in 99.5 percent[25], it is clear that when compressing a collection of genomes one can obtain better compression ratios than when compressing the sequences separately. Initially, the researchers tried to use the similarity between a sequence to be compressed and a reference sequence.

[1]Institute of Informatics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland. [2]Nubitech, 40-684 Katowice, Poland. Correspondence and requests for materials should be addressed to S.D. (email: sebastian.deorowicz@polsl.pl)

The first impressive result was by Christley *et al.*[16]. They showed that the description of differences between James Watson's genome and the reference genome can be stored in as little as 4.1 MB. Taking into account that the complete haploid human genome is of size 3.1 Gbases, this translates to ~750-fold compression. This result was recently improved by Pavlichin *et al.*[17] who reduced the space for the JW genome to about 2.5 MB (compression ratio ~1250).

Such large compression ratio was possible since the data were preprocessed, i.e., precise information of all variants were available. This is not always the case, as the genomes can be obtained in different experiments with different reference genomes or the genomes can be *de novo* assembled. In such situations the data to be compressed are collections of complete genomic sequences. This significantly complicates the compression task, as the differences between sequences are not given explicitly; they have to be found, e.g., by multiple complete genome alignment, which is a very complex problem. Moreover, for technological reasons, the differences between *de novo* assembled genomes are usually larger than between the reassembled genomes.

Several papers for the problem of compression of collections of genomic sequences were published[19,20,22,26,27]. In majority of them, each single sequence is compressed separately, by identifying the differences between it and a single reference genome. This allowed to obtain compression ratios for human genomes up to 400, much poorer than ~1250 obtained by Pavlichin *et al.*[17]. This is the price for the lack of prior knowledge about the compressed data. The most successful attempts at obtaining higher compression ratios were possible by exploring the knowledge of similarities between more sequences in the collection. Since such approaches are the real competitors to the proposed algorithm, we will describe them a little more.

The first attempt in this direction was GDC-ultra[19]. It takes a single reference sequence and constructs a search structure (namely, hash table) for it. Then it compresses the first sequence of the collection by looking for similarities between this sequence and the reference. When the sequence is processed, it is used as an additional reference sequence for further sequences, so a separate search structure is constructed for it. The same is for the following sequences, so for example, the 25th input sequence of the collection is compressed by looking for the differences between it and: the main reference sequence, the formerly processed 24 sequences of the collection. The number of additional reference sequences is limited to 39 (for technical reasons only, mainly to keep the necessary amount of memory at a reasonable level). If the collection consists of more than 39 sequences, the 40th, 41st, etc. sequence is compressed with the 40 references only. The differences between the current sequence and the referential sequences is finally Huffman coded. Such approach proved to be promising, since the collection of 69 human genomes were compressed with ratio ~1000.

A different approach was used by Wandelt *et al.*[21] in their FRESCO algorithm. They investigated several variants, and below we will describe the one that gave the best results. The collection is divided into two sets: (*i*) additional references, (*ii*) remaining sequences. FRESCO constructs a search structure (suffix tree) for the main reference sequence. Then it looks for similarities between the additional reference sequences and the main reference performing classical Ziv–Lempel parsing[28] of additional reference sequences. As a result it obtains for each additional reference a sequence of triples (position in the main reference, length of the identical part, next symbol). For the Ziv–Lempel-parsed additional reference sequences a search structure (hash table) is built. After that FRESCO is ready to perform the compression of the remaining sequences from the collection. Each sequence is Ziv–Lempel-parsed against the main reference sequence. Then, the sequence of triples is compressed using the additional Ziv–Lempel-parsed reference sequences serving as the second-level reference. The obtained compression ratios are impressive as they are approximately 3000 for the collection of about 1000 haploid genomes of the 1000 GP, when 70 additional reference sequences were used.

The best compression ratios for the genomic collection was obtained by TGC algorithm[18]. It is, however, from a different category, since as an input it takes a Variant Call Format (VCF)[29] file describing the differences between genomes and the reference sequence, so it processes essentially the same data as Pavlichin *et al.*[17]. In this work we deal with complete genomes stored in FASTA format. In theory it is possible to convert FASTA files into VCF files, but it would require making a close to optimal alignment of many complete genomes (i.e., finding the smallest set of differences between these genomes), which is far from being trivial, especially due to a presence of long structural variants. Nevertheless, comparing the obtained results with TGC will be interesting, as it will allow us to see how far we are from the top algorithm for the similar problem. The main idea of TGC is to split the VCF file into two files. The first (dictionary of variants) stores a description of each variant (i.e., its type, position, alternative alleles, etc.). The second file stores the binary representation of presence/absence of each single variant in each single sequence. The bit vectors (one for each sequence) are compressed using a specialized Ziv–Lempel-based algorithm. The dictionary file is also compressed using a specialized algorithm. The compression ratios of TGC for the collection of 1092 diploid human genomes (when taking only 1 reference sequence) is about 15,500.

Methods

Definitions. For precise description of the proposed algorithm let us define some terms. As an input we have a single reference sequence R and a collection of genome sequences $\mathcal{S} = \{S^1, S^2, ..., S^n\}$. Each

sequence is composed of symbols from some alphabet Σ, i.e, $S^k = s_1^k s_2^k \dots s_{|S^k|}^k$ for each $1 \leq k \leq n$, where $s_i^k \in \Sigma$ for each valid i and $|S^k|$ denotes the length of S^k. Also $R = r_1 r_2 \dots r_{|R|}$, where $r_i \in \Sigma$ for each valid i and $|R|$ denotes the length of R. For any sequence X (a reference or from the collection) $X_{i,j} = x_i x_{i+1} \dots x_j$.

For the DNA sequences the alphabet should ideally contain only 4 symbols (**A**, **C**, **G**, **T**), but in practice **N** (unknown) symbols are quite frequent. Moreover, sometimes also other IUPAC codes appear. Thus in the work we assume only that the symbols are letters from the ASCII code (we also distinguish between lower- and uppercase letters).

Compression algorithm. At the beginning, the compression algorithm reads the reference sequence R and constructs a search structure HT^R (namely, hash table with linear probing[30]) for it. The hash value is computed for each h_{1m}-symbol long substring of R ($h_{1m} = 15$ by default, but a different value can be specified by a user), i.e., for all $R_{i,i+h1m-1}$, where $1 \leq i \leq |R| - h_{1m} + 1$. After that, the main processing of the collection $\mathcal{S}$ starts. The compression algorithm is two-level.

At the first level, we perform the Ziv–Lempel factoring of all sequences from the collection $\mathcal{S}$. This means that for each sequence S^k from $\mathcal{S}$ we produce a sequence L^k composed of tuples (the first symbol of a tuple, denoted as f_x (where x indicates the type of the tuple), is an identifier, as it will become clear later in the text). To this end, we start from $i = 1$ and look for the longest common substring $S_{i,j}^k$ present in R. Since the search structure HT^R contains substrings of length h_{1m} it is not possible to find shorter matches. There are two possibilities here:

- No match of length at least h_{1m} is found. Then, we append a tuple describing single symbol s_i^k, i.e., $\left\langle f_{\text{literal}}, s_i^k \right\rangle$ to L^k and update the current sequence position: $i \leftarrow i+1$.
- Otherwise we have a match $S_{i,j}^k = R_{p,p+j-i}$ of length $j-i+1$. We encode it by appending the tuple $\left\langle f_{\text{match\_1st\_lev}}, p, j-i+1 \right\rangle$ to L^k. Then, we update the current sequence position: $i \leftarrow j+1$.

There is, however, some exception to the general rule that no shorter than h_{1m} symbols match can be found. Genomic sequences often differ by single nucleotide polymorphism (SNPs) or short indels (a few symbols long insertions or deletions). Thus, when some match is found, before looking for another match in R using the hash table HT^R, we do 3 (or 5, depending on the user-specified option) simple verifications. We check whether the next symbol(s) after the current match is just a single nucleotide mutation or a single-symbol (or double-symbol) indel. We allow matches found after such variation to be of length h_{1e} (equal to 4 by default). The rationale for such decision is two-fold. Firstly, it speeds up the searching as for the verification we do not need to query the hash table HT^R. Secondly, such matches (even if they are short) can be quite efficiently encoded as the match position is easy to predict (encoding of Ziv–Lempel parsing results is described below). Thus, even if the sequence L^k will be longer when such short matches are allowed, the final compression ratio can be better.

At the second level, the algorithm performs a similar Ziv–Lempel factoring of the collection $\mathcal{L} = \{L^1, L^2, ..., L^n\}$ to obtain the collection $\mathcal{D} = \{D^1, D^2, ..., D^n\}$. We will use here similar notations as for the sequences $\mathcal{S}$, i.e., l_i^k is the ith tuple of sequence L^k, $L_{i,j}^k$ is $l_i^k l_{i+1}^k \dots l_j^k$. Additionally we define the *weight* of a substring $L_{i,j}^k$ as the sum of weights of the tuples it is composed of, where the weight of a literal tuple is 1 and the weight of a match tuple is 7 (values chosen experimentally). A search structure HT^L (namely, hash table with linear probing) is used here to look for matches in $\mathcal{L}$. At the beginning HT^L is empty, but we update it by adding the already processed sequences of $\mathcal{L}$, i.e., when processing L^k the hash table HT^L contains only all substrings of tuples of weights "close" to $h_2 = 11$ of $L^1, L^2, ..., L^{k-1}$. (For each position i in the tuple sequence L^u we take the shortest substring (in terms of the number of tuples) $L_{i,j}^u$ of weight not smaller than h_2.) The substrings of tuples of L^k are added to the hash table after L^k is processed.

Now, when we process L^k starting from $i = 1$ to obtain D^k, we look for the match of the largest weight $L_{i,j}^k = L_{p,p+j-i}^u$. There are two possible situations here:

- No match of weight at least h_2 is found. In this case we append the tuple l_i^k (describing the first-level literal or the first-level match) to D^k and update the current sequence position: $i \leftarrow i+1$.
- Match $L_{i,j}^k = L_{p,p+j-i}^u$ is found. In this case we append the tuple $\left\langle f_{\text{match>\_2nd>\_lev}}, u, i, j-i+1 \right\rangle$ to D^k and update the current sequence position: $i \leftarrow j+1$.

The sequence D^k is composed of tuples of three kinds: first-level literal (pair), first-level match (triple), second-level match (quadruple). Since when processing L^1 the search structure HT^L is empty, $D^1 = L^1$.

The reason for using two-level Ziv–Lempel factoring is that the genome sequences are usually highly similar, so in the whole collection the same series of matches and literals between the current sequence and the reference sequence can be found. Thus, instead of storing the series of tuples many times, it is beneficial to encode them once and only reference to them for other sequences. Figure 1 shows how the two-level factoring is performed.

input data

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
R	A	C	T	G	A	C	C	G	T	C	G	A	T	T	T	A	A	C	C	C		
S^1	A	A	C	T	G	A	C	C	G	T	A	A	G	A	T	T	T	A	A	T	G	C
S^2	A	C	T	A	A	C	C	G	T	C	C	A	T	T	T	A	A	T	G	C		
S^3	A	C	T	A	A	C	C	G	T	C	A	A	T	T	T	A	A	T	G	C		
S^4	A	A	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	G	T	G	C	
S^5	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	C	C	C				
S^6	A	C	T	G	A	T	C	G	T	C	G	T	T	A	A	G	C	C				

first level factoring

L^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
L^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^3	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, A) (L1M, 12, 6) (L1L, T) (L1L, G) (L1L, C)
L^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1L, T) (L1L, G) (L1L, C)
L^5	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 5)
L^6	(L1M, 1, 5) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L1M, 19, 2)

second level factoring

D^1	(L1M, 16, 3) (L1M, 3, 7) (L1L, A) (L1L, A) (L1M, 11, 7) (L1L, T) (L1L, G) (L1L, C)
D^2	(L1M, 1, 3) (L1L, A) (L1M, 5, 6) (L1L, C) (L1M, 12, 6) (L2M, 1, 6, 3)
D^3	(L2M, 2, 1, 3) (L1L, A) (L2M, 2, 5, 4)
D^4	(L1L, A) (L1M, 16, 3) (L1M, 3, 3) (L1L, T) (L1M, 7, 5) (L1M, 13, 2) (L1M, 16, 2) (L1L, G) (L2M, 1, 6, 3)
D^5	(L1M, 1, 5) (L2M, 4, 4, 3) (L1M, 16, 5)
D^6	(L2M, 5, 1, 4) (L2M, 4, 7, 2) (L1M, 19, 2)

Figure 1. Example of first- and second-level factoring in GDC 2 algorithm, where: $h_{1m} = 3$, $h_{1e} = 2$, $h_2 = 3$, weight of a literal tuple is 1 and weight of a match tuple is 2. Blue and green colors are used only to distinguish between adjacent first-level matches. The red underline is to point the second-level matches. The used abbreviations: L1L — $f_{literal}$, L1M — $f_{match\_1st\_lev}$, L2M — $f_{match\_2nd\_lev}$.

The collection $\mathscr{D}$ is a succinct representation of the input collection $\mathscr{S}$. Nevertheless, it has potential to be compressed even more if we use an arithmetic coder[31]. What is important, instead of encoding the tuples as they are, we predict some of their values (e.g., matching positions) and encode only the differences between our predictions and the real values. The successive fields of the tuples are arithmetically encoded as follows.

Flags.   There are only 3 different flags distinguishing between the tuple types. We encode them contextually, where the context is composed of two recently encoded flags.

Codes of symbols in the first-level literals.   Codes of symbols are encoded contextually, where the context is the recently encoded symbol.

Positions of the first-level matches.   These positions can be from a broad range, i.e., between 1 and $|R| - h_{1m} + 1$. Since the genomic sequences are similar, the position of the current match is likely to be close to the position of the previous match increased by the number of symbols encoded in the meantime. Thus, before encoding the position *pos* we estimate its value *expected_pos* and encode only the difference *relative_pos = expected_pos − pos*. The *expected_pos* is calculated by increasing the recently encoded *pos* by: (*i*) the length of the last match, (*ii*) the number of literals encoded since the last match, (*iii*) the number of symbols encoded as the second-level matches seen from the recent first-level match. Then, the estimation is classified as: *perfect* (*relative_pos = 0*), *good*, ($0 < |relative\_pos| < 2^6$), *poor* (other values). Finally, the estimation type is encoded without a context and the necessary number of bytes (0, 1, or 4) of *relative_pos* are encoded with context being the estimation type and number of encoded byte. (Please note that various decisions on boundaries between, e.g., classes of estimations, length of matches, are based on preliminary experiments on parts of the input data and the exact influence of these decisions on the overall compression ratio is not presented.)

Lengths of the first-level matches. Each length is classified as: *short* (not longer than 2^8 symbols), *long* (of length between 2^8 and $2^{16} + 2^8$ symbols), *very long* (longer than $2^{16} + 2^8$ symbols). Then, the length type is encoded (without a context). Finally, the necessary number of bytes (1, 2, or 4) of the length are encoded with context being the length type and the number of encoded byte.

Sequence ids of the second-level matches. The value *id* is split into two integers: $\lceil id/256 \rceil$ (prefix) and $id - 256 \times \lceil id/256 \rceil$ (suffix). The prefix is encoded without a context. The context of the suffix is the prefix.

Positions of the second-level matches. Similarly like the positions of the first-level matches, these values can be from a broad range. Thus, instead of encoding them as they are, we estimate the position and encode only the difference. Let us assume the current sequence is L^k. We need some auxiliary array $A[1..k]$ to make the estimations possible. Now we will discuss how A is maintained when processing L^k Then, we will show how it is used to estimate the positions of the second-level matches.

Let us assume that the we have a match in the sequence L^u. After encoding it we store in $A[u]$ the pair $\langle p^A, s^A \rangle$, where p^A is the match position in L^u and s^A is the number of symbols of S^k processed before the current match.

Thus, the encoding of the match positions is made as follows. For a match in the sequence L^u we calculate the difference d between the current position in S^k and the position S^A stored in $A[u]$. Then, we advance the position p^A (stored in $A[u]$) of L^u as long as the number of the symbols covered by the first-level literals and matches is not larger than d. What we obtain is the expected position in L^u for the current match.

Then, we calculate the difference between the expectation and the value of the current tuple. The estimations are classified as: *perfect* (difference is 0), *good* (absolute value of the difference between 1 and 16), *moderate* (absolute value of the difference between 16 and 256), and *poor* (other values). Finally, the estimation type is encoded without a context and the necessary number of bytes of the difference are encoded with context being the estimation type and the number of the encoded byte.

Lengths of the second-level matches. The lengths are classified according the their value to: *short* (not longer than 2^4 tuples), *medium-sized* (between 2^4 and $2^5 + 2^4$ tuples), *long* (between $2^5 + 2^4$ and $2^7 + 2^5 + 2^4$ tuples), *very long* (between $2^7 + 2^5 + 2^4$ and $2^8 + 2^7 + 2^5 + 2^4$ tuples), *extremely long* (the rest). Then, the length type is encoded (without a context). Finally, the necessary number of bytes is encoded, where the context is the length type and additionally (for extremely long lengths) also the number of encoded byte.

Decompression algorithm. Decompression is straightforward. At the beginning the $\mathscr{D}$ collection is obtained by arithmetically decoding the compressed file. Then, the collection $\mathscr{L}$ is decoded. Finally, the sequences of $\mathscr{S}$ are constructed from $\mathscr{L}$ and R.

Access to a single compressed sequence. A drawback of the proposed algorithm is that to decompress S^n we need to decompress (at least at the second level) all other sequences. More precisely, to obtain S^m we need to have $L^1, L^2, \ldots, L^{m-1}$ as they must be known to obtain L^m. Then, we can obtain S^m from L^m and R. This can be important especially when m is large. To partially solve this problem we implemented a variant of the compression algorithm in which we allow to set by the user (during compression) the fraction of the sequences that can be used as the second-level references. Thus, when this parameter is, e.g., 30%, in the worst case only 30% of $\mathscr{L}$ must be decompressed. This deteriorates the compression ratio, so this is rather a compromise than a perfect solution.

Real implementation. To increase the speed of the compression and decompression we designed the compressor in a multithreaded fashion. There are several (user-defined) threads performing the first-level compression (and decompression) and a single thread performing the second-level compression (and decompression). For example, in the compression, each of the first-level threads reads a sequence S^k from a queue of sequences to compress and performs the Ziv–Lempel factoring of S^k according to R. The results L^k are stored in an in-memory queue Q. The second-level-compression thread reads sequences L^k from Q, performs the Ziv–Lempel factoring of it according to the already processed part of sequences from $\mathscr{L}$ obtaining D^k and finally performs also the entropy coding of D^k. (We use a popular and fast arithmetic coding variant by Schindler, also known as a range coder (http://www.compressconsult.com/rangecoder/).) The queue Q has FIFO (first in first out) organization, so there is no guarantee in which order the sequences of $\mathscr{L}$ will be processed (it depends on the processing time of the sequences by the first-level threads). Thus, the compression ratios can slightly differ between the executions of the algorithm.

The parallel design of the decompression algorithm is similar.

The compression output is composed of three files. The one with extension gdc2_desc stores file names, sequence sizes, and ids of the multi-FASTA sequences. It is small, but to provide the best possible compression ratio of the whole algorithm, it is compressed using popular zlib library. The file with

extension gdc2_rc contains the compressed representation of the collection $\mathcal{S}$. Finally, the file with extension gdc2_ref stores the compressed reference sequence R. As it is not a part of the collection to be compressed, its size is not counted in the experimental results. Nevertheless, we decided to compress it for the situations in which the user is interested in storing both the reference R and the collection $\mathcal{S}$ in a single place in a compact form. This file is compressed by gathering symbols in triples and encoding them arithmetically.

Relation of the proposed compressor to the existing works. The proposed compressor bares some similarities to the existing works. The main concept of two-level Ziv–Lempel factoring is an extension of what was done in FRESCO[21]. There are, however, significant differences between these two approaches. FRESCO uses LZ77 factoring[28], in which the sequences are divided into triples ⟨match position, match length, next symbol after the match⟩, while GDC 2 uses LZSS factoring[32] (which encodes the sequence as a list of matches and literals as described in the previous section). Moreover, FRESCO looks for longer matches, while GDC 2 allows very short matches if they are close to the previous matches. These different approaches have impact on the next stage of the compression, as in FRESCO there are less factors to be entropy encoded, but each needs more bits. The concept of looking for short matches after some longer ones is an extension of what was made in our previous work[19]. In GDC 2 we, however, permit not only single-letter mismatches, but also short indels. We also do not limit the number of short matches in a series.

In FRESCO, the collection of sequences is split into two sets: additional references and the remaining sequences. The additional references are compressed only according to the main reference sequence. The remaining sequences are compressed only according to the main and the additional references. Such design decision of FRESCO means that the maximal number of additional reference sequences is, in some way, limited. The reason is that if there are too many additional references, they could occupy a significant amount of space and the gain in better compression of the remaining sequences could not compensate that. In GDC 2, we do not split the collection into two sets. We just use all of the already processed sequences as the additional references for the current sequence, with significant boost in the compression ratio.

The most important difference between FRESCO and GDC 2 is, however, in the compression of tuple fields. FRESCO estimates the position of the first-level matches in a similar way as GDC 2, but the positions and lengths of the second-level matches are processed as they are, without estimations. All the integers are stored using specific byte code. The whole resulting byte sequence is then Huffman coded (with no contextual statistics). In GDC 2, the positions of the second-level matches are estimated in a complex way, which allows to store only small values (differences between estimations and real positions). Moreover, the second-level matches are also taken into account for the estimation of the first-level matches. Finally, GDC 2 uses a better entropy coder and the fields are compressed contextually, to reduce the redundancy even more. The way the tuples are encoded using an arithmetic coder, especially the calculation of the expected positions for the first- and second-level matches, is novel in this context.

GDC 2 and FRESCO differs also in the design of the internal data structures used for indexing sequences, which influences the processing speed.

The multithreaded design of GDC 2 was not used by existing multi-reference genome compressors.

Results

Our compressor, GDC 2, was implemented in C++11 language using C++ built-in concurrency mechanisms. The test machine was equipped with Intel i7 4930 K CPU (6 cores, clocked at 3.4 GHz), 64 GB of RAM, and two 3 TB HDDs in RAID 0 (measured average read speed about 350 MB/s).

For the experiments we used two large datasets. *A.thaliana* dataset of total size 94 GB was obtained from the 1001 GP[7] and contains 775 sequences. *H.sapiens* dataset of total size 6670 GB was obtained from the 1000 GP[2] and contains 2184 sequences (from 1092 diploid human genomes). We also used one smaller dataset, *H.sapiens alternate assemblies*, which contains 11 alternatively assembled sequenced (haploid) human genomes. A more detailed description of all datasets (e.g., the chosen reference sequences) is given in the supplementary material.

The comparison of all of the existing genomic data compressors would be very hard due to many reasons. For example, some compressors do not support symbols other than ACGT, some cannot work with so huge data, some are very slow and performing complete experiments would take months. Thus we selected the compressors that proved to be the best (in terms of compression ratio) in the previous studies: 7z (general purpose compressor from the Ziv–Lempel family), RLZ[26], GReEn[27], ABRC[20], GDC normal[19], GDC ultra[19], iDoComp[22], FRESCO[21]. In the preliminary experiments (Table 1), we evaluated them on subsets of our datasets to select the candidates for more complete evaluation. As the results show, the single-reference compressors (RLZ, GReEn, ABRC, GDC-normal, iDoComp) give ratios much smaller than 1000 for *H.sapiens* chromosomes and smaller than 160 for *A.thaliana* chromosomes.

The general purpose 7z can be seen as a multi-reference compressor since it looks for matches between the present sequence and the sequences seen in the past 1 GB. For *H.sapiens* Chromosome 21 it means about 20 recently processed sequences. Nevertheless, for *H.sapiens* Chromosome 1 these would be only 4 sequences. The true multi-reference compressors GDC-ultra and FRESCO give much better ratios for

Dataset	7z	RLZ	GReEn	ABRC	GDC-normal	iDoComp	GDC-ultra	FRESCO
H.sapiens								
Chr. 14	1,068	270	218	472	674	625	2,455	1,946
Chr. 21	1,561	269	211	460	685	642	2,397	2,545
A.thaliana								
Chr. 1	242	86	64	67	154	156	254	186
Chr. 4	234	80	59	61	141	145	230	170

Table 1. Compression ratios for subsets of the datasets for various compressors.

human chromosomes. For FRESCO we set the number of additional reference sequences to 100 as in a preliminary experiment (results not shown) this leaded to better compression ratios than the value 70 used in the original paper[21].

In a consequence, for further experiments we selected two best single-reference compressors, i.e., GDC-normal and iDoComp, and two best multi-reference compressors, i.e., GDC-ultra and FRESCO. The results of evaluation of the chosen compressors and the proposed GDC 2 are presented in Tables 2 and 3. For the *H.sapiens* dataset (Table 2) the compression ratio of GDC 2 is about 9500, which is approximately 4 times better than the best of the existing competitors. As it was mentioned earlier, GDC 2 bears some similarity to FRESCO, so it would be interesting to ask which of the changes between these algorithms have the highest influence on the GDC 2 advantage in the compression ratio. It is hard to answer precisely, as the algorithms differ in many details, and the improvements implemented in GDC 2 are not independent. Nevertheless, from the presented results and other preliminary tests, it seems that the two most important things are: careful estimation of the positions of the second-level matches and allowing more reference sequences (especially, using all the already processed sequences as the references for the current one). Most of the rest of the advantage is probably due to: allowing short matches if they are close to the previous ones and contextual encoding of integers.

In the compression, the fastest is GDC 2, which works with a speed about 200 MB/s. Measuring of the speed of decompression is problematic as some of the compressors work faster than the disk speed (~350 MB/s), which in practice is more than sufficient. Nevertheless, we were interested in what is the true decompression speed of the GDC 2 algorithm, so we measured it with the output redirected to /dev/null (i.e., the sequences were decompressed but not stored) obtaining about 1000 MB/s.

The experiment for the *A.thaliana* dataset (Table 3) shows that the compression ratios are much worse. The best ratio, almost 600 was obtained by GDC 2. This result is approximately 2.4 times better than the second best, GDC-ultra. Also the compression speeds are worse here.

We also experimented with *H.sapiens alternate assemblies* dataset (Table 4). In contrast to previous datasets, here the sequences are much more diversified. Also the collection is much smaller as it contains only 11 individuals. Now the best compression ratios were obtained by GDC ultra algorithm. The advantage over GDC 2 is, however, rather small. Moreover, GDC ultra is more than 30 times slower. It is also important to stress that GDC ultra scales poorly, as the maximal number of reference sequences is 40, so for larger collections GDC 2 should win in the compression ratio. The fastest algorithm here was iDoComp. When running FRESCO, we selected here one-level compression because the collection was so small (or too divergent; it is hard to find the correct answer due to the small number of individuals) that two-level approach gave worse results.

It is also interesting to compare the compression ratios with what is possible, when much more knowledge of the data is given (cf. TGC columns in Tables 2 and 3). Namely, when the input data are given as differences between the sequences and the reference (in VCF format), the best available compressor, TGC, obtained even better ratios. For human dataset they were about 15,500, on average. When we compare this with about 9,500 of GDC 2 we see that we are quite close to what is theoretically possible. Similar results are for *A.thaliana* dataset: ~590 ratio for GDC 2 and ~860 ratio for TGC. What is, however, worth to stress, GDC 2 is able to compress collections of sequences of the same species gathered from various sources (e.g., *de novo* assembled), when no alignment of them is given, while TGC input must be provided as aligned sequences described as variants between them. Therefore, TGC ratios should be seen only as a hint what in theory is possible for the examined collections of genomes, even if it would be extremely hard to obtain such results, and should not be directly compared to the ratios of the rest of examined compressors.

In the next experiment, we measured the influence of the number of sequences in the input collection on the compression ratio, compression and decompression speeds, and memory usage. The results for two chromosomes are shown in Fig. 2. As one can see for the human chromosome the compression ratio is about 8000 for 300 input sequences and increases moderately for growing number of input sequences. The same phenomenon can be observed for *A.thaliana* dataset, but the ratio is about an order of magnitude lower.

Data	Raw size [GB]	GDC normal ratio	iDoComp ratio	GDC ultra ratio	FRESCO ratio	GDC 2 ratio	TGC ratio
Chr. 1	551.7	680	659	2,508	2,279	**10,556**	*17,090*
Chr. 2	538.5	628	608	2,318	2,113	**9,828**	*15,744*
Chr. 3	438.4	602	552	2,263	2,044	**9,564**	*15,513*
Chr. 4	422.8	547	503	2,202	1,911	**8,979**	*14,800*
Chr. 5	400.6	624	576	2,260	1,997	**9,578**	*15,449*
Chr. 6	378.7	566	522	2,184	1,950	**8,832**	*14,628*
Chr. 7	352.3	592	545	2,138	1,918	**8,752**	*14,344*
Chr. 8	323.9	584	543	2,137	1,916	**8,817**	*14,558*
Chr. 9	312.5	718	666	2,450	2,359	**10,400**	*17,000*
Chr. 10	300.1	578	564	2,123	1,973	**9,335**	*14,736*
Chr. 11	298.8	560	521	2,171	1,967	**9,043**	*14,825*
Chr. 12	296.2	595	547	2,167	1,958	**9,127**	*14,897*
Chr. 13	255.0	611	564	2,452	1,842	**10,669**	*17,336*
Chr. 14	237.6	674	625	2,458	1,946	**10,654**	*17,371*
Chr. 15	227.0	716	664	2,458	2,020	**10,815**	*17,556*
Chr. 16	200.1	647	604	2,068	2,076	**8,980**	*14,374*
Chr. 17	179.7	646	594	2,059	2,090	**8,651**	*14,137*
Chr. 18	172.9	568	525	2,051	2,066	**9,033**	*14,368*
Chr. 19	130.8	569	519	1,773	1,828	**7,137**	*11,931*
Chr. 20	139.5	633	619	2,014	2,240	**9,150**	*14,419*
Chr. 21	106.5	686	642	2,405	2,545	**10,414**	*16,975*
Chr. 22	113.5	823	772	2,455	2,718	**10,547**	*17,537*
Chr. X-fem	178.5	911	826	2,551	2,628	**11,060**	*17,364*
Chr. X-mal	81.0	945	896	2,740	2,469	**11,546**	*15,255*
Chr. Y-mal	30.0	38,233	59,062	42,870	39,228	**132,123**	*188,704*
Chr. X-mal1	2.8	312	310	587	713	**2,423**	*3,967*
Chr. X-mal2	0.35	280	456	741	943	**5,914**	*9,199*
Complete	6,669.8	627	586	2,262	2,065	**9,557**	*15,490*
Compression speed [MB/s]		73	51	12	111	**202**	—
Compression time [h:m]		25:23	36:32	154:24	16:38	**9:11**	—

Table 2. Compression ratios for *H.sapiens* dataset. The ratios are calculated as raw size divided by compressed size rounded to the integer. Compression speeds (in MB/s) and times (in hours:minutes) are given in the bottom part of the table. Raw sizes are in GBs. The rightmost column contains the ratios for TGC program[18]. The values are in italic to stress that they cannot be directly compared with ratios for the remaining algorithms, since TGC assumes that the input data are in VCF format, which explicitly stores information of all differences between aligned genomes from the collection. Nevertheless, the TGC ratios show what is possible, when much more knowledge of the data is given.

The memory usage of GDC 2 depends mainly on the number of sequences serving as the second-level references as they must be stored (and indexed) in memory during compression. In this experiment all sequences were used as additional references, so the memory consumption grew constantly up to about 6 GB. (The most memory consuming was compression of *H.sapiens* Chromosome 2 for which about 24 GB of RAM was necessary.) The visible stepwise increment of the memory usage is a consequence of the assumed possible hash table size (being always a power of 2).

The compression and decompression speeds for the human dataset initially grow with the increasing number of sequences and are the highest for the collection of size about 300–500. This is correlated with the growing compression ratio. Roughly speaking, the more second-level references, the better the second-level factoring (i.e., longer matches can be found) and so, there are significantly less data to process by the arithmetic coder. However, for larger collections, much more data must be analyzed during the second-level factoring, so the speed of compression falls down. A similar thing happens in the decompression. The better second-level factoring means less data to be arithmetically decoded, which

| Data	Raw size [GB]	GDC normal ratio	iDoComp ratio	GDC ultra ratio	FRESCO ratio	GDC 2 ratio	TGC ratio
Chr. 1	23.9	154	156	254	186	**621**	*916*
Chr. 2	15.5	143	148	239	175	**559**	*807*
Chr. 3	18.4	147	152	238	169	**551**	*804*
Chr. 4	14.6	141	145	230	170	**553**	*803*
Chr. 5	21.2	148	151	254	187	**624**	*916*
Chr. C	0.12	652	1,830	652	1,750	**25,061**	*8,983*
Chr. M	0.29	558	807	600	374	**1,401**	*1,592*
Complete	94.0	148	151	245	179	**587**	*857*
Compression speed [MB/s]		**120**	47	13	7	94	—
Compression time [h:m]		**0:13**	0:33	1:57	3:31	0:17	—

Table 3. Compression ratios for *A.thaliana dataset*. The ratios are calculated as raw size divided by compressed size rounded to the integer. Compression speeds (in MB/s) and times (in hours:minutes) are given in the bottom part of the table. Raw sizes are in GBs. Please confer Table 2 caption for the discussion on TGC ratios.

Data	Raw size [MB]	GDC normal ratio	iDoComp ratio	GDC ultra ratio	FRESCO ratio	GDC 2 ratio
Chr. 1	2,647	146	134	**170**	118	159
Chr. 2	2,700	198	180	**224**	150	213
Chr. 3	2,098	174	159	**192**	137	179
Chr. 4	2,125	145	129	**158**	113	147
Chr. 5	2,018	170	153	**196**	135	184
Chr. 6	1,907	204	184	**252**	149	238
Chr. 7	1,755	214	191	**282**	155	267
Chr. 8	1,622	217	194	**248**	156	236
Chr. 9	1,478	280	255	**363**	202	344
Chr. 10	1,496	249	225	**319**	182	313
Chr. 11	1,499	185	168	**214**	145	205
Chr. 12	1,486	166	150	**186**	130	175
Chr. 13	1,206	185	168	**219**	146	208
Chr. 14	1,115	269	243	**313**	202	305
Chr. 15	1,045	180	164	**218**	150	200
Chr. 16	954	69	72	**86**	68	81
Chr. 17	892	128	113	**153**	107	139
Chr. 18	859	157	146	**174**	141	173
Chr. 19	681	175	155	**202**	133	188
Chr. 20	695	222	200	263	181	**264**
Chr. 21	475	219	197	**258**	179	248
Chr. 22	497	153	137	**168**	132	163
Chr. X	1,690	90	83	**97**	83	91
Chr. Y	404	269	241	**302**	225	275
Complete	33,346	169	153	**194**	134	183
Compression speed [MB/s]		1.5	**27.5**	0.3	4.2	11.0
Compression time [h:m]		6:02	**0:20**	26:36	2:11	0:50

Table 4. Compression ratios for *H.sapiens alternate assemblies* dataset. The ratios are calculated as raw size divided by compressed size rounded to the integer. Compression speeds (in MB/s) and times (in hours:minutes) are given in the bottom part of the table. Raw sizes are in MBs.
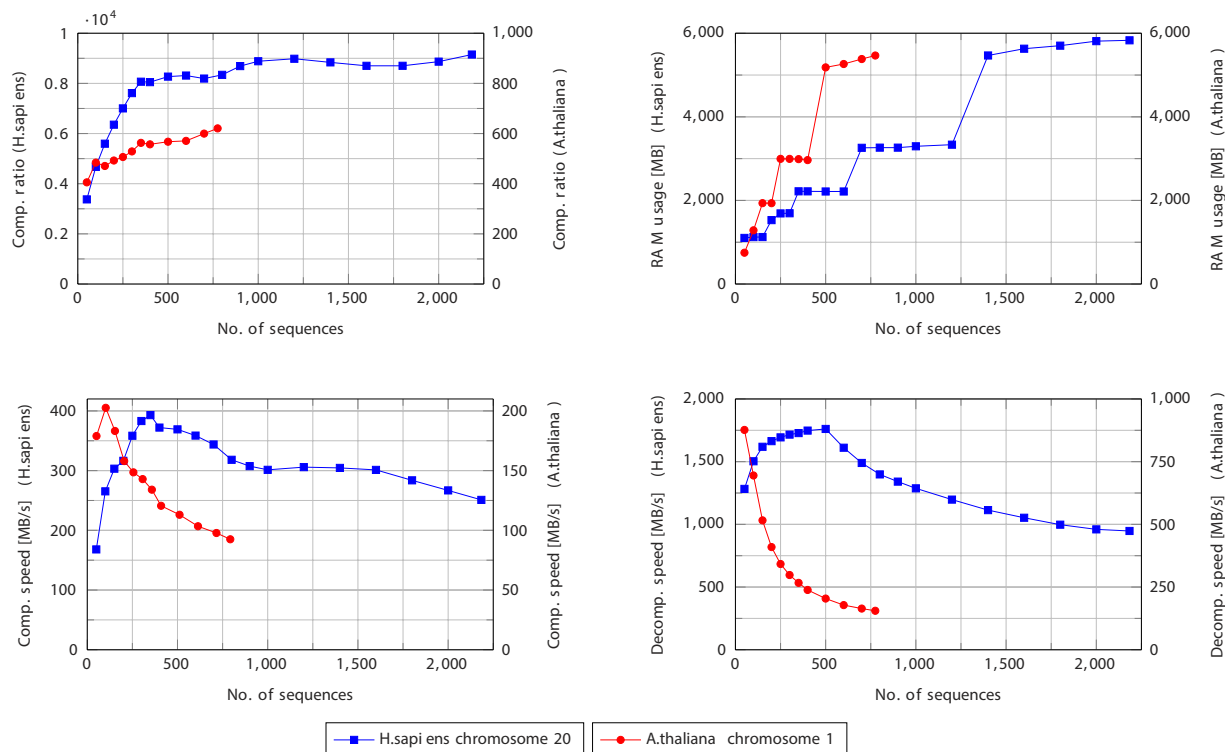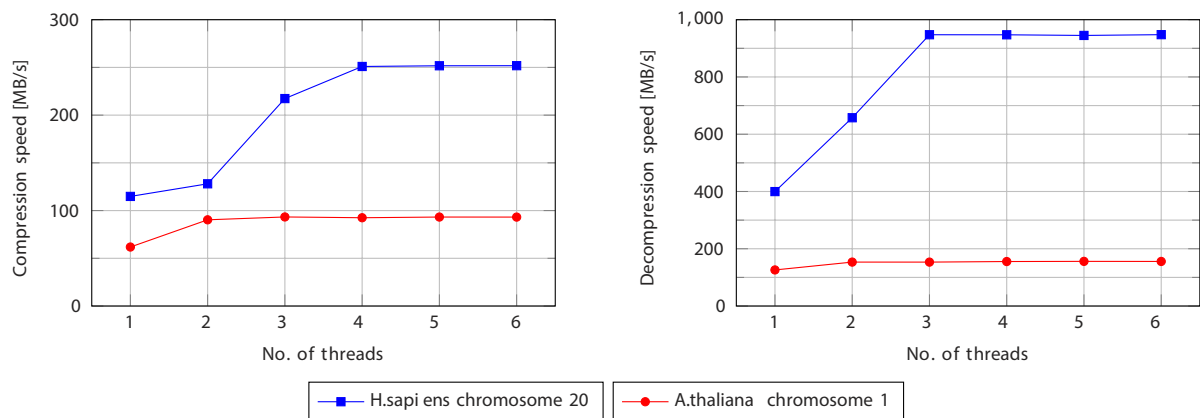
Figure 2. Influence of the number of sequences in the input collection on GDC 2: compression ratio (left top), memory usage (right top), compression speed (left bottom), decompression speed (right bottom). The decompression speed was measured when the output was redirected to /dev/null, i.e., the sequences were decompressed but not stored.
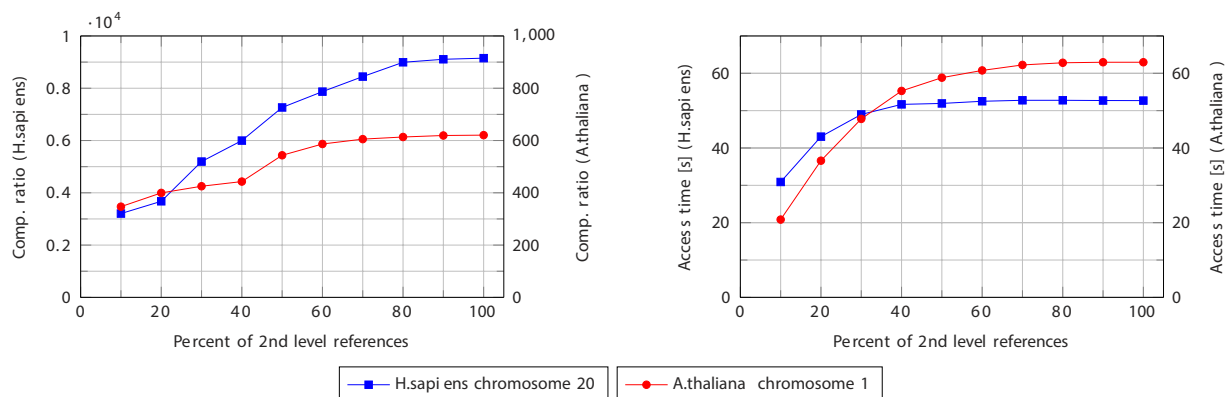
Figure 3. Influence of the percent of 2nd level references on GDC 2 compression ratio (left), decompression (access) time of a single sequence (right). In both cases the input dataset contained all available sequences.

increases the speed. Unfortunately, more second-level references means much more computations for the estimation of the positions of matches, which significantly influences the decompression time for large collections.

In the next experiment, we measured the influence of the number of reference sequences in the second level of GDC 2 on the compression ratio, (de)compression speeds and the extraction time of a single sequence of a collection. The most important results are presented in Fig. 3 (the complete results are in Supplementary Figure S1). Decreasing the number of second-level references by half results in a reduced RAM usage (about half less RAM is used) and a noticeable speed up of compression (24% for *H.sapiens* dataset and 17% for *A.thaliana* dataset) at a cost of some decrease of compression ratio (26% and 14%, respectively). Using even less sequences in the second level of GDC 2 leads also to significant gains in

Figure 4. Influence of the number of threads used by GDC 2 algorithm on compression and decompression speeds.

speed of decompression of complete collection or a single sequence, obviously at a cost of decreased compression ratio. For 10% of the sequences used, average single sequence access times decreased from 53 to 31 seconds for *H.sapiens* dataset (at a cost of 2.85 worse compression ratio) and from 63 to 21 seconds for *A.thaliana* dataset (at a cost of 1.79 worse compression ratio).

GDC 2 is implemented in a multithreaded fashion, so it is natural to ask how its speed scales when the number of threads is increased. By default, GDC 2 uses 4 threads: 3 for the first level Ziv–Lempel factoring and 1 for the second-level factoring and arithmetic coding. The results presented in Fig. 4 show that the value 3 or 4 seems to be the best choice for the used datasets and the test machine. The speed is limited by disk speed or (for fast disks) by the single second-level compressing thread. This suggest that splitting this thread into two, e.g., one performing Ziv–Lempel factoring and other performing arithmetic compression would increase the total performance of GDC 2. Nevertheless, since the absolute values of compression speeds are high, we resigned from that in the present version of the software.

Discussion

We proposed a new algorithm for compression of collections of complete genome sequences. The evaluation shows that its compression ratios are roughly 4 times better than the best existing competitors. Moreover, it is very fast, as the compression speed for the *H.sapiens* dataset is about 200 MB/s. The decompression speed is limited by the speed of the disk used in the experiments. When we measured this speed without storing the files onto disks, it was about 1000 MB/s. The algorithm is designed primarily to compress and decompress efficiently a large collection of genomes all at once. However, it also performs well for relatively small, divergent genome sets. Moreover, extraction of a single sequence is also possible. The access time, although not impressive (counted in tens of seconds), can be significantly improved at a cost of some decrease in the overall compression ratio.

References

1. Illumina Inc. TruGenome Clinical Sequencing Services. (2015) Available at: http://www.illumina.com/clinical/illumina_clinical_laboratory/trugenome-clinical-sequencing-services.html (Accessed: 17th February 2015)
2. The 1000 Genome Project Consortium. An integrated map of genetic variation from 1092 human genomes. *Nature*. **491,** 56–65 (2012).
3. The UK10K Consortium. Rare Genetic Variants in Health and Disease. (2013) Available at: http://www.uk10k.org/ (Accessed: 16th February 2015)
4. Ball, M. P. *et al.* A public resource facilitating clinical use of genomes. *PNAS*. **109(30),** 11920–11927 (2012).
5. U.S. Department of Veteran Affairs. The Million Veteran Program. (2013) Available at: http://www.research.va.gov/mvp/veterans.cfm (Accessed: 16th February 2015)
6. Weigel, D. & Mott, R. The 1001 Genomes Project for Arabidopsis thaliana. *Genome Biology*. **10,** Article no. 107 (2009).
7. Max Planck Institute for Developmental Biology. 1001 Genomes: A Catalog of Arabidopsis thaliana Genetic Variation (2012) Available at: http://1001genomes.org/ (Accessed: 16th February 2015)
8. Kahn, S. D. On the future of genomic data. *Science* **331,** 728–729 (2011).
9. Deorowicz, S. & Grabowski, S. Data compression for sequencing data. *Algorithms for Molecular Biology*. **8,** Article no. 25 (2013).
10. Jones, D. C., Ruzzo, W. L., Peng, X. & Katze, M. G. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*. **40,** e171 (2012).
11. Bonfield, J. K. & Mahoney, M. V. Compression of FASTQ and SAM format sequencing data. *PloS ONE*. **8,** e59190 (2013).
12. Roguski, L. & Deorowicz, S. DSRC 2—Industry-oriented compression of FASTQ files. *Bioinformatics*. **30,** 2213–2215 (2014).
13. Grabowski, S. Deorowicz & Roguski, R. Disk-based compression of data from genome sequencing. *Bioinformatics*. **31,** 1389–1395 (2015).
14. Hach, F., Numanagic, I. & Sahinalp, S. C. DeeZ: reference-based compression by local assembly, *Nature Methods*. **11,** 1082–1084 (2014).
15. Fritz, M. H.-Y., Leinonen, R., Cochrane, G. & Birney, E. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*. **21,** 734–740 (2011).
16. Christley, S., Lu, Y., Li, C. & Xie, X. Human genomes as email attachments. *Bioinformatics*. **25,** 274–275 (2009).

17. Pavlichin, D., Weissman, T. & Yona, G. The human genome contracts again. *Bioinformatics*. **29,** 2199–2202 (2013).
18. Deorowicz, S., Danek, A. & Grabowski S. Genome compression: a novel approach for large collections. *Bioinformatics*. **29,** 2572–2578 (2013).
19. Deorowicz, S. & Grabowski, S. Robust relative compression of genomes with random access. *Bioinformatics*. **27,** 2979–2986 (2011).
20. Wandelt, S. & Leser, U. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology*. **7,** Article no. 30 (2012).
21. Wandelt, S. & Leser, U. FRESCO: Referential Compression of Highly Similar Sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*. **10(5),** 1275–1288 (2013).
22. Ochoa, I., Hernaez, M. & Weissman, T. iDoComp: a compression scheme for assembled genomes. *Bioinformatics*. doi: 10.1093/bioinformatics/btu698 (2014).
23. Giancarlo, R., Rombo, S. E. & Utro, F. Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Briefings in Bioinformatics*. **15,** 390–406 (2014).
24. Zhu, Z., Zhang, Y., Ji, Z., He, S., Yang, X. & High-throughput D. N. A. sequence data. *Briefings in Bioinformatics*. **16,** 1–15 (2015).
25. Levy, S. *et al*. The diploid genome sequence of an individual human. *PLoS Biology*. **5,** e254 (2007).
26. Kuruppu, S., Puglisi, A. J. & Zobel, J. Optimized relative Lempel-Ziv compression of genomes. In: *Proceedings of the ACSC Australasian Computer Science Conference* (ed. Reynolds, M.). Australian Computer Society, Inc., Sydney, Australia, 91–98 (2011).
27. Pinho, A. J., Pratas, D. & Garcia, S. P. GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*. **40,** e27 (2012).
28. Ziv, J. & Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions of Information Theory*. **23,** 337–343 (1977).
29. Danecek, P. *et al*. The variant call format and VCFtools. *Bioinformatics*. **27,** 2156–2158 (2011).
30. Knuth, D. E. The art of computer programming: Volume 3: Sorting and Searching. 2nd Edition. Addison-Wesley Professional, Boston (1998).
31. Salomon, D. & Motta, G. Handbook of data compression. Springer, London (2010).
32. Storer, J. A. & Szymanski, T. G. Data compression via text substitution. *Journal of the ACM*. **29,** 928–951 (1982).

Acknowledgments

Author Contributions

S.D. and A.D. designed the algorithm. S.D., A.D. and M.N. prepared the implementation. S.D. and A.D. performed the experiments. S.D. and A.D. drafted the manuscript and the supplementary material. All authors read and approved the final manuscript.

Additional Information

Supplementary information accompanies this paper at http://www.nature.com/srep

Competing financial interests: The authors declare no competing financial interests.

How to cite this article: Deorowicz, S. *et al*. GDC 2: Compression of large collections of genomes. *Sci. Rep.* **5**, 11565; doi: 10.1038/srep11565 (2015).