GTC: an attempt to maintenance of huge genome collections compressed

Agnieszka Danek[1] & Sebastian Deorowicz[1]

April 28, 2017

We present GTC, a new compressed data structure for representation of huge collections of genetic variation data. GTC significantly outperforms existing solutions in terms of compression ratio and time of answering various types of queries. We show that the largest of publicly available database of about 60 thousand haplotypes at about 40 million SNPs can be stored in less than 4 Gbytes, while the queries related to variants are answered in a fraction of a second. GTC can be downloaded from `http://sun.aei.polsl.pl/REFRESH/gtc`.

In the last two decades the throughput of genome sequencers increased by a few orders of magnitude. At the same time the sequencing cost of a single human individual decreased from over 1 billion to about 1 thousand dollars. Stephens *et al.* predicts[1] that in 2025 the genomic data will be acquired at 1 zetta-bases/year, while about 2–40 EB/year of them should be deposited for a long term. From the other side, the prices of storage and transfer decrease moderately[2], which means that keeping the data management costs under control becomes a real challenge.

Recently Numanagic *et al.*[3] benchmarked the tools for compression of sequenced reads. The ability of the examined utilities to shrink the data about 10 times is remarkable. Nevertheless, much more is necessary. The obvious option is to resign from storage of raw data (in most experiments) and focus just on the results of variant calling, deposited usually in the Variant Call Format (VCF) files.[4] The famous initiatives, like the 1000 Genomes Project[5] or the 10K UK Project, deliver VCF files for thousands of samples. Moreover, the scale of compilation works, like of the Haplotype Reference Consortium (HRC)[6] or the Exome Aggregation Consortium (ExAC)[7], is by an order of magnitude larger. For example, the VCF files of the HRC consist of 64,976 haplotypes at about 39.2 million SNPs and occupy 4.3 TB. It is also clear that much larger databases will be formed in the near future. VCF files contain a list of variants in a collection of genomes as well as evidence of presence of a reference/non-reference allele at each specific variant position in each genome. As they are intensively searched, the applied compression scheme should support fast queries of various types. The indexed and gzipped VCF files can be effectively asked using VCFtools[4] or BCFtools when the query is about a single variant or a range of them. Unfortunately, retrieving a sample data means time-consuming decompression and processing of a whole file.

Recently Layer *et al.*[8] introduced Genotype Query Tools (GQT) that made use of some specialized compression algorithm for VCF files. GQT was designed to compare sample genotypes among many variant loci, but did not allow to retrieve the specified variant as well as sample data. Shortly after that, Li proposed BGT[9] based on the positional Burrows–Wheeler transform[10]. It offered more than 10-fold better compression than GQT and supported queries about genotypes as well as variants. Moreover, it allowed to restrict the range of samples according to some metadata conditions. The SeqArray library[11] for the R programming language is yet another solution to effectively compress and browse VCF files. The applied compression is based on the LZMA algorithm.[12]

We introduce GTC (GenoType Compressor), a new tool for compressed representation of genotypes supporting fast queries of various types. It is designed to offer much better compression and much faster queries than existing solutions.

At the beginning, GTC divides the variants into blocks of 3584 consecutive entries and processes each block separately (Fig. 1). The bit vectors representing presence/absence of reference alleles in all haplotypes are formed. In fact two bit vectors are necessary to describe each variant, as four possibilities must be covered: a reference allele, non-reference allele, the other non-reference allele, or unknown allele.

The haplotypes in each block are independently permuted to minimize the number of differences (i.e., a Hamming distance) between successive entries. As determination of the best permutation of haplotypes is equivalent to solving the Travelling Salesperson Problem (TSP)[13] it is practically impossible to find the optimal solution in a reasonable time. Thus, the Nearest Neighbor heuristic[13] is picked to quickly calculate a reasonably good solution. There are better algorithms for this task (in terms of minimizing the total number of differences between neighbor haplotypes). Unfortunately, they are too slow to be applied here. The description of the found permutation must be stored for each block to allow retrieval of the original data.

The permuted haplotypes are compressed (still within blocks) using a hybrid of specialized techniques inspired by Ziv-Lempel compression algorithm, run length encoding, and Huffman coding.[12] The bit vectors are processed one by one. In the current bit vector we look for the longest runs of 0s or 1s, as well as longest matches (same sequences of bits) in the already

[1]Institute of Informatics, Silesia University of Technology, Gliwice, Poland. Correspondence should be addressed to {agnieszka.danek, sebastian.deorowicz}@polsl.pl

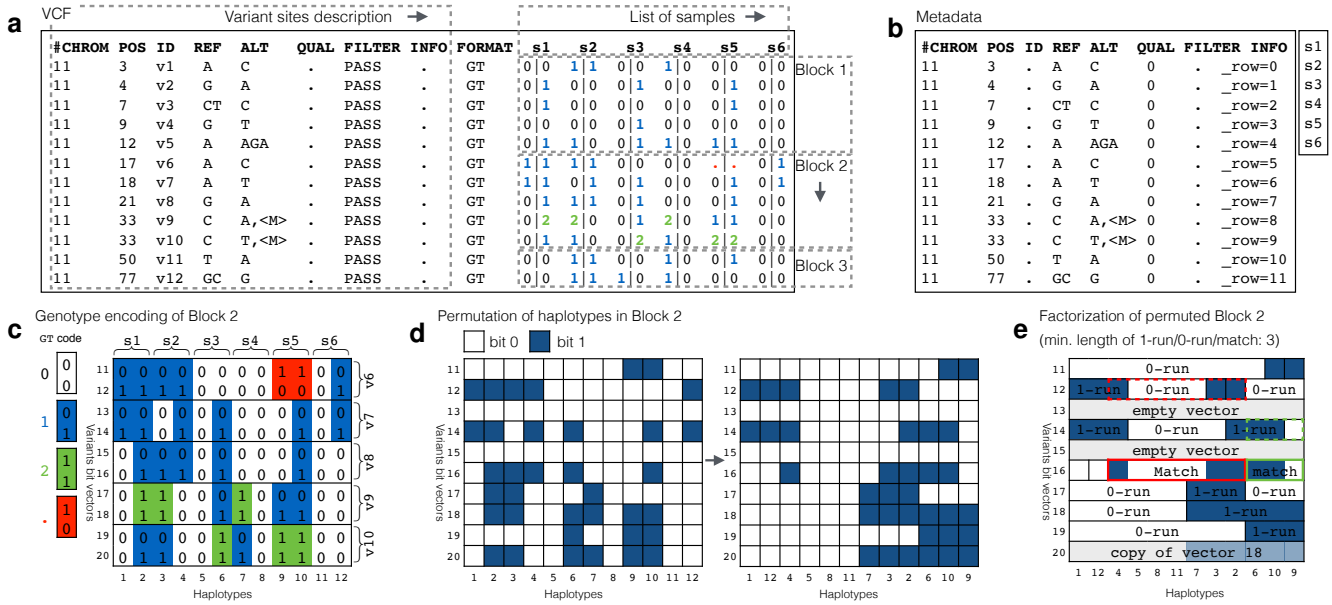

Figure 1: Construction of GTC archive. (**a**) The input VCF file with genotypes of 6 diploid samples at 12 variant sites. It is decomposed into metadata and blocks of genotypes. Each block (here: max. 5 variant sites) is processed separately. (**b**) Metadata: variant sites description (stored as a site-only BCF) and list of samples. (**c**) Bit vector representation of genotypes in Block 2. (**d**) Permutation of haplotypes in Block 2. Resulting order of haplotypes is stored. (**e**) Factorization of permuted Block 2. Empty and copied vectors are marked. All unique bit vectors are described as a sequence of longest possible 0-runs, 1-runs, matches to previous bit vectors in the block and literals.

processed bit vectors. As the result we obtain a description of the current variant using the previous variants, which is much shorter than the original bit vector. The description is finally Huffman encoded to save even more space. More details of the algorithm are given in Online Methods.

To evaluate GTC and the competitors we picked two large collections of *H. sapiens* genomes: the 1000 GP Phase 3 (2,504 genotypes), and the HRC (27,165 genotypes). HRC collection used in our experiments was picked from the European Genomphenom Archive and is slightly smaller than the full data set (containing 32,488 samples; unfortunately unavailable for public use). The characteristics of the data sets and the compression capabilities of gzip, BGT, GQT, SeqArray, and GTC are given in Fig. 2a. GTC archive appeared to be about two times more compact than BGT archive and tens times than gzipped VCF. The SeqArray archive size is between GTC and BGT for the smaller collection, but is much larger than BGT for the large one.

For a detailed examination of the compression ratios we reduced the HRC collection to 1000, 2000, etc. samples containing only Chromosome 11 variants. Fig. 2b shows that the relative sizes of the BGT and GTC archives are similar in the examined range. A more careful examination shows that the compressed sizes of BGT, GTC grows slightly sub-linearly for growing number of genotypes. Moreover, the margin between BGT and GTC is almost constant for more than 5000 samples, which suggest that it would remain the same for even larger collections. SeqArray seems to scale poorer than BGT and GTC when the collection size grows.

The great compression would be, however, of little value without the ability of answering queries of various types. The VCFtools and BCFtools offer relatively quick access to VCF, gzipped VCF, or BCF files when we ask about a specified variant or variant range but the situation changes when we focus on samples.

Figures 2c–g show the times of answering various types of queries for Chromosome 11 data containing various number of samples. The GTC decompression of the whole collection of variants is from 7 times (1000-sample subset) to 13 times (all samples) faster than of BGT and about 3 times faster than of SeqArray (Fig. 2c). The GTC extraction of single variant or range of genome of size 23,329 bases (median size of human gene) is up to 6 times faster than of BGT (Fig. 2de). It is also worth to note that BCFtools are almost as fast as GTC for a single variant query. Fig. 2g shows the extraction time of a range of genome (from 1 base to 1 million bases) for 27,165 samples. Once again GTC is about ten times faster than BGT. Moreover, the advantage of GTC grows slightly when the range extends. It can be noted how the relative performance of BCTtools degrades when the range width exceeds a few thousands bases. A bit different situation can be observed in Fig. 2f, when the decompression times of single sample are presented. For collections not larger than 5,000 samples GTC is the fastest, but then, BGT takes the lead becoming 1.5 times faster for the complete collection. Nevertheless, both GTC and BGT answer the query in less than 1 second. As expected BCFtools are clearly the slowest.

a

Dataset	No. genot.	No. var. [M]	Size [GB] VCF	VCF-gzip	GQT	BGT	SA	GTC
1000GP–3	2,504	85.20	853	17.41	14.34	3.87	2.93	**1.81**
HRC	27,165	40.36	4,300	75.01	84.42	7.01	12.68	**3.98**
HRC Chr11	27,165	1.94	210	3.42	4.11	0.33	0.62	**0.18**

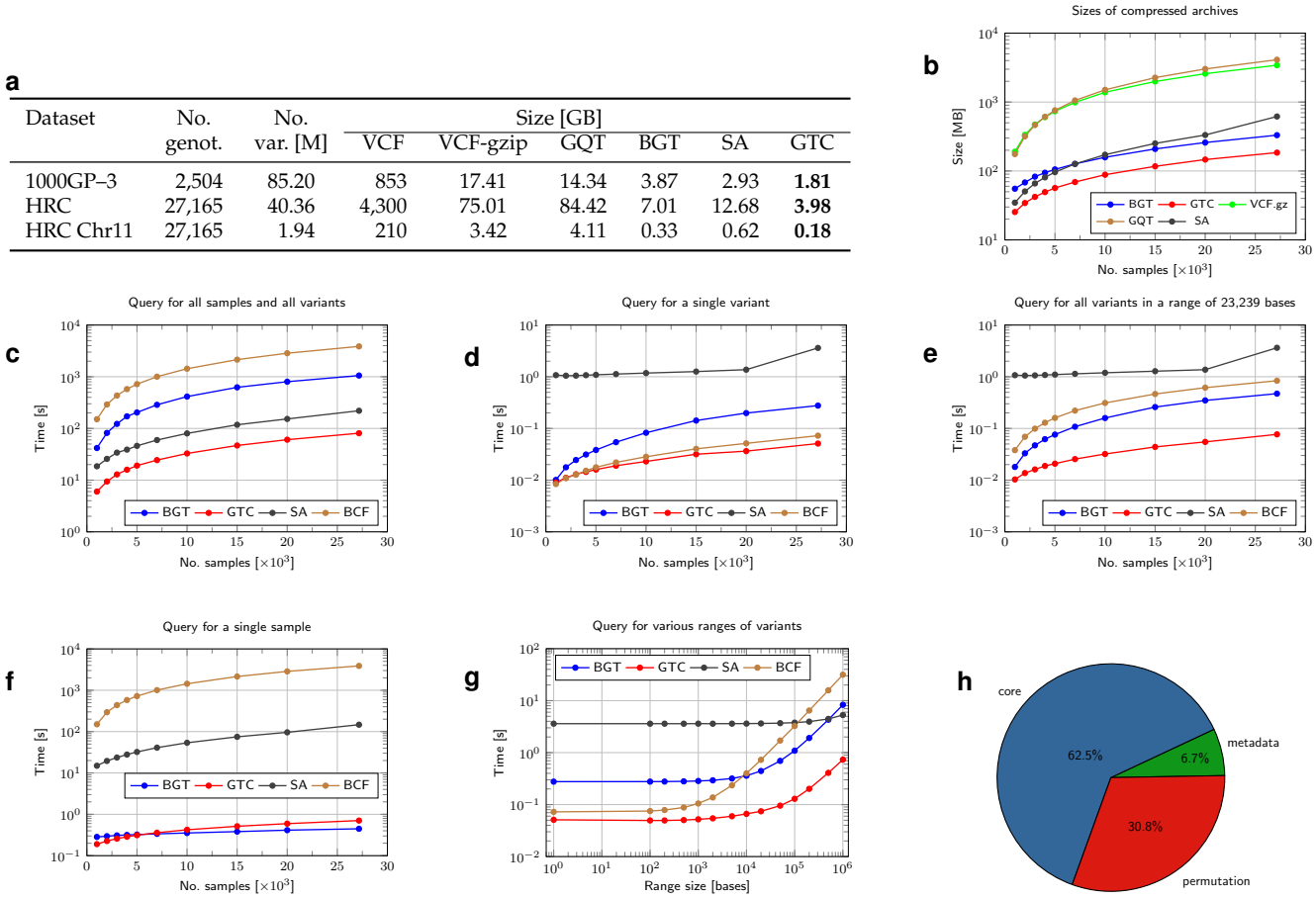

Figure 2: Comparison of compressed data structures. (**a**) Sizes of compressed archives for the examined collections. (**b**) Sizes of compressed sampled archives of HRC data for VCF (gzipped), GQT, BGT, SeqArray (SA), and GTC. (**c**) Decompression times for HRC Chromosome 11 data. (**d**) Variant query times for HRC Chromosome 11 data. (**e**) Variant range query times for HRC Chromosome 11 data. (**f**) Sample query times for HRC Chromosome 11 data. (**g**) Variant range query times (for various range sizes) for HRC Chromosome 11 data. (**h**) Components of GTC archive for HRC data.

The compression times of BGT, SeqArray, and GTC are similar (slightly more than a day for the complete HRC collection at Intel Xeon-based workstation clocked at 2.3 GHz) as they are dominated by reading and parsing of the huge input gzipped VCF file.

It is also interesting to see the sizes of components of GTC archive for the complete HRC data. As can be observed in Fig. 2h the majority of the archive (62.5%) is for the description of matches, 0-runs, etc. 30.8% is for the description of permutations in blocks. Finally, 6.5% is for the description of variants (position, reference and non-reference alleles, etc.) and 0.2% for list of sample names. For smaller block sizes the haplotypes can be better compressed, but the description of permutation consumes more space (see Supplementary Fig. 7). In the Supplementary Figures 1–6 we also show the influence of various parameters of GTC (like block size, minimal match length, etc.) on the compression ratio.

From the pure compression-ratio perspective it is worth to note that the average size of genotype information in GTC archive for a single sample is just about 146 KB. Even better result would be possible when we resign from fast queries support. To date the best pure compressor of VCF files is TGC[14]. Recently Tatwawadi *et al.* proposed GTRAC[15], a modified version of TGC, supporting some types of queries, e.g., for single (or range of) variants or single samples. Unfortunately, the output of GTRAC queries is just a binary file so no direct comparison with VCF/BCF-producing compressors (BGT, GTC, SeqArray) is possible. Moreover, we were unable to run GTRAC for the examined data sets. Nevertheless, we modified our compressor to produce similar output and support similar query types, and run both GTC and GTRAC for the 1000GP Phase 1 data containing 1092 genotypes and 39.7M variants. The compressed archives were of size 622 MB (GTC) and 1002 MB (GTRAC). The queries for single variants were solved in 7 ms (GTC) and 47 ms (GTRAC). The queries for samples were answered in similar times, about 2 s for Chromosome 11 data.

The most significant advantage of GTC over other approaches is a new compression algorithm designed with a care of fast

queries support. The data structure is so small that it is possible to store it in the main memory of even commodity workstations giving the impressive query times. The scalability experiments suggest that much larger collections can be also maintained effectively.

Methods

Methods and any associated references are available in the online version of the paper.

Acknowledgments

This work was supported by National Science Centre, Poland under project DEC-2015/17/B/ST6/01890.

Author contributions

A.D. and S.D. conceived the compressed data structure and query algorithms. A.D. developed majority of the software. A.D. and S.D. designed and performed the experiments. S.D. and A.D. wrote the manuscript and supplementary material. All authors have read and approved the final manuscript.

Competing financial interests

The authors declare no competing financial interests.

References

1. Stephens, Z.D. *et al. PLOS Biol.* **13**, e1002195 (2015).
2. Deorowicz, S. & Grabowski S. Algorithms Mol. Biol. **8**, 25 (2013).
3. Numanagic, I. *et al. Nat. Methods* **13**, 1005–1008 (2016).
4. Danecek, P. Bioinformatics **27**, 2156–2158 (2011).
5. Sudmant, P.H. *et al. Nature* **526**, 75–81 (2015).
6. McCarthy, S. *et al. Nat. Genetics* **48**, 1279–1283 (2016).
7. Lek, M. *et al. Nature* **536**, 285–291 (2016).
8. Layer, R.M., Kindlon, N., Karczewski, K.J., Exome Aggregation Consortium, Quinlan, A.R. *Nat. Methods* **13**, 63–65 (2016).
9. Li, H. *Bioinformatics* **32**, 590–592 (2015).
10. Durbin, R. *Bioinformatics* **30**, 1266–1272 (2014).
11. Zheng, X. *et al. Bioinformatics* doi:10.1093/bioinformatics/btx145 (2017).
12. Salomon, D., Motta, G. *Handbook of data compression* (2010).
13. Johnson, D.S., McGeoch, L.A. In *Local Search in Combinatorial Optimisation.* 215–310 (1997).
14. Deorowicz, S., Danek, A., Grabowski, S. *Bioinformatics* **29**, 2572–2578 (2013).
15. Tatwawadi, K., Hernaez, M., Ochoa, I., Weissman, T. *Bioinformatics* **32**, i479–i486 (2016).

Online methods

General idea

GTC compresses a collection of genotypes in a VCF/BCF format and allows for queries about genotypes:

- at specified range of variant sites positions (e.g., a single variant site), referenced to as variant query,
- of specified samples (e.g., a single sample), referenced to as sample query,
- combination of the above.

The ploidy of individuals determines the number of haplotypes that make up a single genotype. For diploid organisms, a genotype of an individual is defined by two separate haplotypes.

Definitions

For precise description of the proposed algorithm let us denote some terms. As an input we have a VCF/BCF file that describes H haplotypes at V single allele variants (sites). For any bit vector X, $X[i]$ it the i th bit of this vector, $|X|$ denotes its length, while $\mathrm{rank}(X[i])$ is a number of set bits in vector X from position 0 to position $i-1$. For simplicity of notation, we assume that $\mathrm{rank}(X) = \mathrm{rank}(X[|X|])$, so it is a number of set bits in the whole bit vector.

Compression algorithm

There are three main phases of the compression algorithm described in more details below: preprocessing the input VCF file (Fig. 3), processing single blocks of genotype data (Fig. 4) and merging blocks.

Preprocessing the input VCF file

Managing the input VCF file. Unphased genotypes in the input BCF/VCF are arbitrary phased, while each of multi allele variant sites (described in a single line of VCF) is break into multiple single allele sites, each described in a separate line (as in BGT[9] VCF preprocessing). Thus, there are four possible allele values for each haploid genotype: '0' for the reference allele, '1' for the non-reference allele, '2' for the other non-reference allele (stored in a different line of the VCF file), '.' for unknown.

Extraction of metadata. The altered description of V variant sites is stored in a site-only BCF file (the HTSlib[16] library is used for this task), with *_row* variable indicating site id added in the INFO field. List of names of samples is stored in a separate text file.

Initial encoding of genotypes. Each haploid genotype at each site is represented as a dibit (00 for the reference allele, 01 for the non-reference allele, 11 for the other non-reference allele, 10 for unknown). Each of V site annotations is represented by two bit vectors of size H: one for lower and one for higher bits of the dibits representing subsequent haploid genotypes. Together there are $2V$ bit vectors (at this point information about genotypes takes $2HV$ bits in total). The vectors are identified by ids ranging from 0 to $2V-1$. The vectors with even ids correspond to the lower bits of the dibits representing haploid genotype sites, while vectors with odd ids correspond to the higher bits. In the next stages, described in details below, the bit vectors are compressed and indexed, making it possible to randomly access an arbitrary vector.

Forming blocks of genotype data. The bit vectors representing genotype data are divided into blocks. A single block contains genotype data of 3584 consecutive variant sites (value chosen experimentally) that is 7168 consecutive bit vectors. Thus, there are $\lceil V/3584 \rceil$ blocks (last may contain data about less than 3584 variants). The blocks are processed independently to each other, in parallel (if possible).

Processing a single block of genotype data

Permutation of haplotypes. The haplotypes in a block are permuted to minimize the number of differences (i.e., a Hamming distance) between neighboring haplotypes. The Nearest Neighbor heuristic[13] is used to calculate a reasonably good solution in an acceptable time. The permutation of the haplotypes (their order after permutation) in the i-th block is stored in P^i array.

Categorizing variant bit vectors. The variant bit vectors (representing genotype annotations) are processed one by one, in a sequential manner. In the initial phase, a byte vector is categorized either as an empty vector (all bytes unset), a copy of a previously processed vector, or a unique vector, not equal to any of the previously processed vectors. The classification is done with help of a dictionary structure HT^{vec} (namely, hash table with linear probing[17]). The hash value is computed for each processed vector and HT^{vec} stores ids of all previously processed unique vectors in the block (notice: first non empty bit vector is a unique vector). Four bit vectors of size 3584 (number of variants in the block) are formed for the i-th block: E^i_{even} and E^i_{odd} used to distinguish, if the kth even or odd (respectively) vector is empty, and C^i_{even} and C^i_{odd} used to distinguish if the kth even or odd (respectively) vector is a copy of one of previous vectors. The C^i_{origin} array maps subsequent copied vectors with their equivalents in the set of unique vectors (vid).

Processing unique variant bit vectors. Every unique variant bit vector, represented by byte vector (of size $\lceil H/8 \rceil$ bytes, padded with zeros), is transformed into a sequence of tuples, which represents literals, runs of zeros or ones, or matches to a previously encoded vector. The encoding is done byte by byte, from

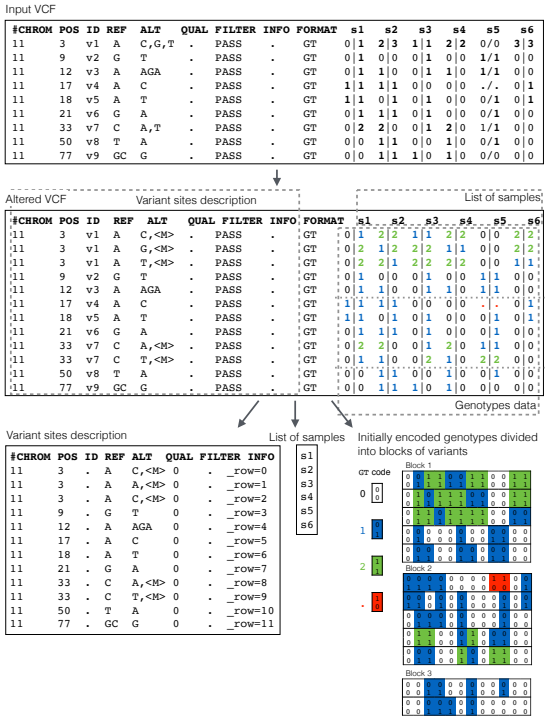

Figure 3: Compression algorithm: preprocessing of the input VCF file. Non-phased genotypes are arbitrary phased. Multiple allele sites are broken into multiple single allele sites. The altered VCF is split into BCF with variant sites description, list of samples and blocks of genotype data. The block size is 5 variants in the example (3584 variants in the real implementation).

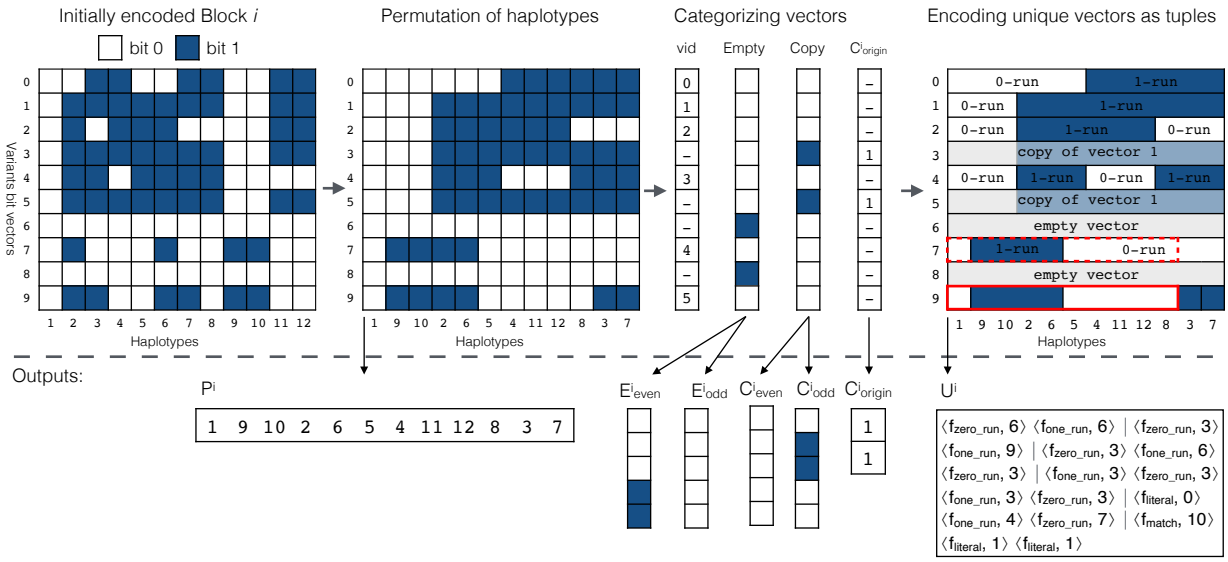

Figure 4: Compression algorithm: processing a single block of genotype data. First, the haplotypes are permuted. Next, each variant bit vectors is categorized as either an empty, a copy or a unique vector. Finally, all unique variant bit vectors are encoded as tuples representing literals, zero runs, one runs, and matches to previous vecotrs. The outputs are: permutation of haplotypes P^i, bit vectors E^i_{even}, E^i_{odd}, C^i_{even}, and C^i_{even} categorizing variant bit vectors, a C^i_{origin} array with ids of original variant bit vectors for all repeated ones, and a sequence of tuples U^i. Note that here all transformations are performed on bit vectors, while in the real implementation byte vectors are considered.

left to right, starting at position $j = 0$. At each analyzed position we first look for the length of the longest substring of zeros (bytes equal to zero) or ones (bytes equal to 255). If it is of length $r_{\min}$ or more ($r_{\min} = 2$ by default), we encode the sequence as a run of zeros (or ones). Otherwise, we look for the longest possible match to one of the previously encoded unique vector (identical byte substring). To increase the chance of a long match, we search for common haplotypes, that is for the longest match starting at jth position in any previously processed byte vector. The matches to arbitrary parts of other vectors are accidental and thus unlikely to be long. Moreover, with the position restriction, the matches need fewer bits to encode (no need to store their positions). To make the search faster, the already processed data from the vectors are indexed using a hash tables HT. Each HT's key consists of a sequence extracted from a vector and its position. The value is the vector id (vid). HT stores keys with sequences of length $h = 5$. The minimum match length is equal to h (and it is not possible to find shorter matches using HT). The matching byte substring (or its parts) in a previously encoded vector can be already encoded as a match. However, we restrict a *match depth* that is a number of allowed vectors describing each byte in a match. By default, the maximum allowed match depth ($d_{\max}$) is 100. A subsequent match to the same vector is encoded with fewer bits, as there is no need to store the id of the previous vector. If, with the match depth restriction, no sufficiently long match can be found, the current byte is encoded as a literal. The runs of literals (between 20 and 252 literals by default) are merged and encoded as a separate tuple.

The type of a tuple is indicated by its first field, a flag f_{type}. Overall, there are six possible tuple types at the current position j:

- $\langle f_{\text{zero\_run}}, num_0 \rangle$—a run of num_0 zero bytes, the position j is updated to $j + num_0$,
- $\langle f_{\text{one\_run}}, num_1 \rangle$—a run of num_1 one bytes (all bits set), the position j is updated to $j + num_1$,
- $\langle f_{\text{match}}, vid, len \rangle$—a match of length len to a vector with id vid, the position j is updated to $j + len$,
- $\langle f_{\text{match\_same}}, len \rangle$—a match of length len to a vector with the same id as the previous match, the position j is updated to $j + len$,
- $\langle f_{\text{literal}}, bv \rangle$—a literal, where bv is the value of the byte, the position j is increased by 1.
- $\langle f_{\text{literal\_run}}, n, bv_1, bv_2, ..., bv_n \rangle$—a run of n literals, where $bv_1, bv_2, ..., bv_n$ are the values of the consecutive bytes, the position j is increased by n.

Merging blocks

The $\lceil V/3584 \rceil$ blocks of genotype data are gathered and merged in the order of their appearance in the input VCF file (each ith block is added, for $i = 1$ to $i = \lceil V/3584 \rceil$).

The bit vectors E^i_{even}, E^i_{odd}, C^i_{even} and C^i_{odd} are merged into four global bit vectors of size H: E_{even}, E_{odd}, C_{even} and C_{odd}. The vectors are kept in a succinct form.

The C_{origin} array gathers all C^i_{origin} arrays adjusting stored vids (adding to each vid from the current block number of unique vectors in all previous blocks). For subsequent copied vectors it refers to vids of the original unique vectors out of all unique vectors. Every vid in C_{origin} is then delta coded (difference between vid of the next unique vector and original vid is calculated) with the minimum necessary number of bits.

The encoded unique vectors are stored in a single byte array U. The flags, literals (bv), lengths of matches (len) and lengths of runs of zeros (num_0) and ones (num_1) are compressed (separately) with an entropy coder (we use Huffman coder) in the context of total number of set bits in the current bit vector (8 separate groups by default). For each literal run, the number of bits it occupies is kept. The ids of vectors (vid) are stored using minimum possible number of bits necessary to encode any vector ($\log_2(number\_of\_unique\_vectors)$).

The starting positions of all unique vectors are stored in a byte array U_{pos}, where every 1025th unique vector is stored using 4 bytes, while for the 1024 subsequent vectors only the difference (between the current vector position and the nearest previous position encoded with 4 bytes) is stored, using d bits, where d is the minimum number of bits necessary to store any encoded position difference.

Finally, permutations of all blocks, P^i, are merged into single array of permutations, P. It is stored with minimum necessary number of bits. The id of the variant bit vector to decompress is enough to find the right permutation in P.

Design

The main components of the GTC data structure are as follows:

- E_{even} and E_{odd}: two bit vectors, each of size H, indicating if subsequent variant vectors (out of all vectors for lower or higher bits, respectively) are zero-only vectors,
- C_{even} and C_{odd}: two bit vectors, each of size H, indicating if subsequent variant vectors (out of all vectors for lower or higher bits, respectively) are copies of other vectors,
- C_{origin}: ids of the original vectors (out of all unique vectors) for subsequent vectors being a copy; minimum necessary number of bits is used to store each id (exactly: $\lceil \log_2(no\_unique\_vectors) \rceil$ bits, where $no\_unique\_vectors = 2V - (\text{rank}(E_{\text{even}}) + \text{rank}(E_{\text{odd}}) + \text{rank}(C_{\text{even}}) + \text{rank}(C_{\text{odd}}))$,
- U: byte array storing unique bit vectors, encoded into tuples and compressed (as described above),
- U_{pos}: byte array with positions of the subsequent unique vectors in the U structure; full position is stored for every 1025th vector, the position of the remaining vectors are delta coded.

- P: byte array storing permutations of subsequent blocks; a single permutation is a sequence of ids of haplotypes reflecting the order in which they appear in the block. Each id is stored with minimum necessary number of bits (exactly: $\lceil \log_2 H \rceil$ bits).

The E_{even}, E_{odd}, C_{even}, and C_{odd} bit vectors are represented by the compressed structure[18,19] implemented in the SDSL[20] library.

Query

Query parameters

By default, the entire set is decompressed into VCF / BCF file. It is possible to restrict the query by applying additional conditions. Only variants and samples meeting all specified conditions are decompressed. The following restrictions are possible:

- range condition—it specifies the chromosome and a range of positions within the chromosome,
- sample condition—it specifies sample or samples,
- alternate allele frequency / count condition—it specifies minimum / maximum count / frequency of alternate allele among selected samples for each variant site,
- variant count condition—it specifies the maximum number of variant sites to decompress.

Decompression algorithm

Two decompression algorithms are possible depending on the chosen query parameters. For most queries a variant-oriented approach is used. A sample-oriented approach is applied in queries about small, i.e., up to 10, number of samples without a range condition. In case of both queries the genotypes are decoded based on the appropriate bytes of variant bit vectors decompressed.

Variant-oriented In this algorithm two vectors representing a variant site are fully decompressed. Their ids are known thanks to *_row* variable in the BCF with variant sites description. Initially, the E_{even}, E_{odd}, C_{even}, and C_{odd} bit vectors are used to define a category of the variant bit vector. Decompression of an empty vector is straightforward. For a copied vector, the rank operations on E_{even}, E_{odd}, C_{even}, and C_{odd} bit vectors are used to determine which copy is it, while the id of the original, unique vector is found using the C_{origin} array. The U_{pos} array is used to find position of the unique variant bit vector in the U array. The consecutive bytes of the variant bit vector are decompressed by decompressing and decoding all flags, literals, lengths of 0s and 1s runs, and matches. If a match is encountered, the variant bit vector containing the match is not fully decompressed. Instead, the complete decoding of all irrelevant tuples from the beginning of the vector up to the match position is skipped as far as possible. For example, stored bit length of a run of literal allows to skip the run without time-consuming literal decoding. The P array helps to find the original order of haplotypes.

If a range of variant sites is queried, the decompression is speed up by keeping the track of an adequate number of previous, already decompressed unique variant bit vectors. Moreover, the permutation of haplotypes only needs to be read from the P array at the beginning of a block, not for each variant site separately.

Sample-oriented In this approach all haplotypes representing the queried sample are fully decompressed. For example, if sample is diploid, two haplotypes are decompressed. In case of more samples, more haplotypes are decompressed.

The decompression starts at the first variant site. The P array is used at the beginning of each block to find the positions of the haplotypes in the permuted bit variant vectors. Each variant bit vector is partly decoded once. The bytes containing information about the decompressed haplotypes are fully decoded and stored, the complete decoding of other bytes is skipped, if possible. As the previous decoded bytes are kept, if a match covering the decompressed haplotype is encountered (only a match within the same block is possible), the byte value can be read immediately.

References

16. Li H. *et al. Bioinformatics* **25** 2078–2079.
17. Knuth, D.E., *The art of computer programming* 426–458 (1999).
18. Raman, R., Raman, V., Rao, S. *Proc. 13th SODA* 233–242 (2002).
19. Navarro G, Providel E. *Proc. SEA* 295–306 (2012).
20. Gog, S., Beller, T., Moffat, A., Petri, M. *Proc. SEA* 326–337 (2014).