

Compressing Resequencing Data with GReEn

Armando J. Pinho, Diogo Pratas, and Sara P. Garcia

Abstract

Genome sequencing centers are flooding the scientific community with data. A single sequencing machine can nowadays generate more data in one day than any existing machine could have produced throughout the entire year of 2005. Therefore, the pressure for efficient sequencing data compression algorithms is very high and is being felt worldwide. Here, we describe GReEn (Genome Resequencing Encoding), a compression tool recently proposed for compressing genome resequencing data using a reference genome sequence.

Key words Data compression, DNA sequences, Probabilistic models, Arithmetic coding, Open source software

1 Introduction

The interest in DNA sequence compression was ignited by the Biocompress algorithm of Grumbach and Tahi [1]. The following two decades have witnessed the proposal of a myriad of algorithms for compressing genomic sequences (e.g., [2–12]). These works explore the non-stationary nature of DNA, which is characterized by an alternation between regions of relatively high and relatively low entropy. Typically, compression algorithms adopt two approaches, one based on Lempel–Ziv-like substitutional procedures that usually perform well on repetitive, low entropy regions, and another based on low-order context-based (Markov) arithmetic coding, which is better suited for regions of high entropy. According to the substitutional paradigm, repeated regions of the genomic sequence are represented by a pointer to a past occurrence of the repetition and by the length of the repeating sequence. Both exact and approximate repetitions, as well as their inverted complements, have been explored.

We have been working in DNA sequence compression for several years [13–18]. The method we proposed in [17] is based

The authors are with the Signal Processing Lab, IEETA, DETI, University of Aveiro, 3810–193 Aveiro, Portugal.

on multiple competing Markov models. We have shown that, somewhat surprisingly, DNA sequences can be accurately explained by these models, which only have local knowledge of the past, in contrast with the generality of other available methods that rely on long range copy operations. In [18], we developed a state-of-the-art method for the efficient compression of resequenced genomes. In this case, a high-quality version of the genome is used as reference, boosting the compression gain. Here, we explain in detail this tool and how to use it for compressing resequenced genomes.

2 Materials

2.1 Availability of Genome Resequencing Encoding

The GReEn (Genome Resequencing Encoding) codec (encoder/decoder) is freely available for noncommercial purposes. It can be downloaded from <ftp://ftp.ieeta.pt/~ap/codecs/GReEn1.tar.gz>.

2.2 Computational Requirements

GReEn is implemented in the C programming language. It was developed under Linux and should compile and execute on every flavor of this operating system. A C compiler is needed (typically, “gcc”), as well as the “tar” application for extracting the required files from the archive, and the “make” application for building the executables. GReEn does not require special hardware: it is able to run on a common laptop computer with Linux installed. However, for handling very large sequences (such as the largest human chromosomes), it is recommended to have at least 4 GBytes of computer memory available.

2.3 Installing GReEn

For installing GReEn, download the package from the website mentioned above and extract the files using, for example,

```
tar zxvf GReEn1.tar.gz
```

A directory named “GReEn1” will be created. From inside this directory, run “make.” Two executables should be created, “GReEnC” and “GReEnD,” respectively, the encoder and decoder. Running each without parameters will list the options available.

The package also includes two sequences, “ecK12DH10B.fa.gz” (*Escherichia coli* K12 substrain DH10B, uid58979) and “ecK12W3110.fa.gz” (*E. coli* K12 substrain W3110, uid58567), that will be used below for illustration.

2.4 Running GReEnC

Running “GReEnC” generates

```
Usage: GReEnC [ -o outFile ]
        [ -v (print more info) ]
        [ -k keySize (def 11) ]
        [ -i (ignore equal size) ]
        [ -f maxFailures (def 0) ]
        refSeq tarSeq
```

where the parameters shown are:

- o **outFile** If this option is present, the encoder writes the encoded sequence in a file named “outFile.”
- v Verbose mode: print more information during operation (*see Note 1*).
- k **keySize** By default, the size of the k -mer used for creating the hash table and for searching for similarities between the reference and target sequences is 11. It can be changed using this encoder option (*see Note 2*).
- i This flag has effect only if the reference and target sequences have the same size. In that case, it forces the standard encoding mode, instead of using the special mode that assumes both sequences are aligned (*see Note 3*).
- f **maxFailures** By default, the maximum number of prediction failures before restarting the copy model is 0. It can be changed using this encoder option (*see Note 4*).
- refSeq** The DNA sequence used as reference (fasta or gzip fasta are accepted) (*see Note 5*).
- tarSeq** The target DNA sequence, i.e., the sequence to be compressed (fasta or gzip fasta are accepted).

To test the encoder, run the command

```
GReEnC -v -o code ecK12DH10B.fa.gz ecK12W3110.fa.gz
```

that will generate a file named “code” with the compressed version of sequence “ecK12W3110.fa.gz,” using “ecK12DH10B.fa.gz” as reference. The following output will be also generated (time and memory information may vary, depending on the computer used to run this example)

```
GReEnC version 1.0
The reference sequence has a total of 4686137 bases
A : 1153640
C : 1190880
G : 1188801
R : 1
T : 1152814
Y : 1
The target sequence has a total of 4646332 bases
A : 1142182
C : 1179079
G : 1181238
T : 1143833
Sequence size difference: -39805
The target sequence has 4 different symbols
```

```

Creating the hash table...
...done
Compressing the sequence...
...done
Total number of bytes: 244661 ( 0.42125 bpb )
Total cpu time used: 1.74 s
Total memory allocated by (m/c/re)alloc: 291.45 MB

```

2.5 Running GReEnD

Running “GReEnD” without parameters displays

```

Usage: GReEnD [ -v (print more info) ]
        [ -o outFile ]
        refSeq codeFile

```

where:

-v Verbose mode: print more information during operation.

-o outFile If this option is present, the decoder writes the decoded sequence in a file named “outFile” (*see Note 6*).

refSeq The DNA sequence used as a reference (fasta or gzip fasta are accepted).

codeFile The compressed file, i.e., the file generated by the encoder using “-o” flag.

To test the decoder and assuming that the “code” file was generated by the previous example, run the command

```
GReEnD -v -o out ecK12DH10B.fa.gz code
```

that generates the file “out” with the sequence of bases of file “ecK12W3110.fa.gz.” This command line will also display information similar to

```

GReEnD version 1.0
The reference sequence has a total of 4686137 bases
The target sequence has a total of 4646332 bases
Sequence size difference: -39805
The target sequence has 4 different symbols
Max failures: 0
Creating the hash table...
...done
Decompressing the sequence...
...done
Got 4646332 bases
Total cpu time used: 1.76 s

```

3 Methods

GReEn [18] is a compression tool based on arithmetic coding. It is able to handle arbitrary alphabets and does not pose any restrictions or requirements on the sequences to compress. Its running time depends linearly on the size of the sequence being compressed.

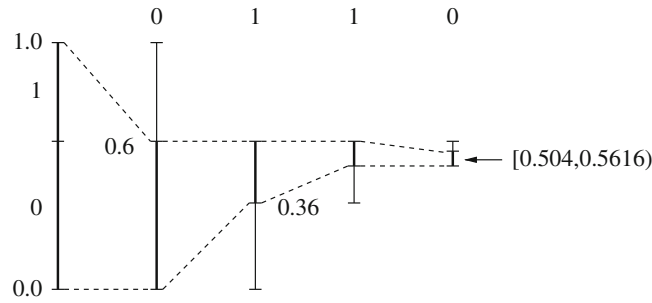


Fig. 1 Example of arithmetic coding: binary alphabet with $P(0) = 0.6$ and message to encode “0110”

3.1 Arithmetic Coding

Arithmetic coding represents the entire message by a single code-word: a number in the $[0, 1)$ interval. It allows for a total separation between the coding process (i.e., generating the bits) and the modeling process (i.e., estimating the probabilities of the symbols of the coding alphabet). For simplicity, consider a binary alphabet and the message “0110.” Also, suppose that the probability of “0” is 0.6 and that this probability is fixed along the message (*see Note 7*).

The encoding of the first symbol of the message (a “0”) leads to a change in the current interval from the initial $[0, 1)$ to $[0, 0.6)$. This is the subinterval that was assigned to symbol “0” and that has a length equal to the probability of the corresponding symbol. Next, the interval is again divided into two parts, the lower 0.6 fraction that is assigned to symbol “0” and the remaining 0.4 fraction to symbol “1.” Since the symbol to encode is now a “1,” the interval is changed to $[0.36, 0.6)$. This procedure is repeated until the end of the message, which is represented by the final interval. In fact, all messages with this “0110” prefix are represented by a subinterval of $[0.504, 0.5616)$, meaning that the “0110” message can be represented by any point inside $[0.504, 0.5616)$, plus the indication of the length of the message. Figure 1 illustrates this coding scheme (*see Note 8*).

3.2 The GReEn Compression Algorithm

GReEn relies on a reference sequence for compressing the target sequence. The reference sequence is generally only slightly different from the target sequence, although this is not mandatory. It is possible to use a sequence from a different species as reference, but, in this case, the compression efficiency will be affected, depending on the degree of dissimilarity between the reference and target sequences. For recovering the target sequence, the decoder needs to have exactly the same reference sequence as that used by the encoder (*see Note 9*).

Denote by $\mathcal{C} = \{c_1, c_2, \dots, c_{|\mathcal{C}|}\}$ the set of all different characters (the alphabet) that are found in the target sequence, where $|\mathcal{C}|$ denotes the number of elements in \mathcal{C} . Also, denote by $\theta(c)$ the

relative frequency of character $c \in \mathcal{C}$ in the target sequence, and by $P_n(c)$ the estimated probability of character $c \in \mathcal{C}$ when encoding the character at position n in the target sequence. The set of probabilities $\{P_n(c), c \in \mathcal{C}\}$ are required by the arithmetic coder (see **Note 10**). For a sequence $x^N = x_1 x_2 \dots x_N$, $x_i \in \mathcal{C}$, with N characters, the arithmetic coder produces a bitstream with

$$-\sum_{n=1}^N \log_2 P_n(x_n) \quad (1)$$

bits, which demonstrates the importance of providing good probability estimates to the arithmetic coder.

The probabilities $P_n(c)$ are calculated using two different models:

1. An adaptive model (the copy model), which assumes that the characters of the target sequence are an exact copy of a part of the reference sequence.
2. A static model that relies on the frequencies of the characters in the target sequence, i.e., $\theta(c)$.

The adaptive model is the main statistical model, as it allows a high compression rate of the target sequence, particularly in areas where the target and reference sequences are highly similar. However, the copy model will at times not be useful and the static model will come into play.

The copy model relies on a pointer to a position in the reference sequence where a character identical to that being encoded is expected. As encoding proceeds, this pointer may be repositioned to different locations on the reference sequence. When repositioning occurs, all parameters of the model are reset. Besides accounting for the number of times, t_n , that the copy model was used after the previous repositioning, two more counters are maintained: h_n^1 , for storing the number of times the model guessed the correct character, including the correct case (uppercase or lowercase), and h_n^2 , to record the number of times the model guessed the character but failed the case (for example, it guessed “A” but the correct character was “a”). Figure 2 illustrates this coding scheme.

Denote by p_n^1 the character predicted by the copy model and by p_n^2 the case converted p_n^1 (for example, if p_n^1 is “A,” then p_n^2 would be “a”). If $p_n^1, p_n^2 \in \mathcal{C}$ (see **Note 11**), the probabilities that are communicated to the arithmetic coder are given by (see **Note 12**)

$$P_n(c) = \begin{cases} \frac{h_n^1 + 1}{t_n + 3}, & \text{for } c = p_n^1, \\ \frac{h_n^2 + 1}{t_n + 3}, & \text{for } c = p_n^2, \\ \frac{1 - P_n(p_n^1) - P_n(p_n^2)}{1 - \theta(p_n^1) - \theta(p_n^2)} \theta(c), & \text{for } c \neq p_n^1, p_n^2. \end{cases} \quad (2)$$

Reference sequence	>SEQ_ID AAAGGATAGGT A CGATATTCCTAG...
Target sequence	>SEQ_ID AGGATAGGT A acgGTATTccta? G ...

Fig. 2 The copy model: when at position 10 of the target sequence (in *green*), the copy model was restarted to position 12 of the reference sequence (in *green*). Apart from the character at position 14 in the target sequence (in *red*), it has since correctly predicted 12 characters, if the case is ignored, or 5 characters (in *upper case*), if case is considered. The copy model is currently predicting that the character in position 23 of the target sequence (in *blue*) should be a G (in *blue* in the reference sequence)

However, if only p_n^1 or p_n^2 belongs to \mathcal{C} , then the probabilities are given by

$$P_n(c) = \begin{cases} \frac{h+1}{t_n+2}, & \text{for } c = p, \\ \frac{1-P_n(p)}{1-\theta(p)} \theta(c), & \text{for } c \neq p, \end{cases} \quad (3)$$

where $h = h_n^1$ if $p = p_n^1$, or $h = h_n^2$ if $p = p_n^2$. As such, we have considered only two events, namely, $\mathcal{E}_1 = \{p\}$ and $\mathcal{E}_2 = \mathcal{C} \setminus \{p\}$, where the distribution of probabilities among the characters of \mathcal{E}_2 is performed as before.

Finally, if both $p_n^1, p_n^2 \notin \mathcal{C}$, the probabilities communicated to the arithmetic coder are the character frequencies of the target sequence, i.e.,

$$P_n(c) = \theta(c). \quad (4)$$

Usually, the codec starts by constructing a hash table with the occurrences and corresponding positions in the reference sequence of all k -mers of a given size (the default size is $k = 11$, but it can be changed using a command line option). Using this hash table, it is easier to find in the reference sequence the characters that come right after all occurrences of a given k -mer.

Before encoding a new character from the target sequence, the performance of the copy model, if in use, is checked. If $t_n - h_n^1 - h_n^2 > m_f$, where m_f is a parameter that indicates the maximum number of prediction failures allowed, the copy model is stopped. The default value for m_f is zero, but this may be changed through a command line option.

Following this performance check, if the copy model is not in use, an attempt is made to restart the copy model before compressing the character. This is accomplished by looking for the positions

in the reference sequence where the k -mer composed of the k -most-recently-encoded characters occurs. If more than one position is found, the one closest to the encoding position is chosen. If none is found, the current character is encoded using the static model and a new attempt for starting a new copy model is performed after advancing one position in the target sequence.

4 Notes

1. If this flag is present, the encoder prints the alphabets of the reference and target sequences. It also indicates how many times it encountered each character.
2. Changing this parameter might affect the compression performance. During normal operation (i.e., when not in “equal size mode”), the encoder often searches for the positionally closest k -mer in the reference sequence that matches the most recently encoded k -mer of the target sequence. The idea is to start a copy model that will predict the following characters with high certainty. If the target sequence is similar to the reference, then values of k greater than 11 might produce higher compression, because in this case it is easier to find the correct location where the repetition occurs in the reference sequence. On the other hand, if the two sequences differ considerably, then it is usually better to leave this parameter unchanged. Note that using greater k values also implies using more memory.
3. When the target and reference sequences are aligned, i.e., when the most probable character in the target sequence is, on average, the one located at the same position in the reference sequence, then there is no need to build the hash table. In fact, the hash table is used for speeding up the process of finding k -mer matches and is overkilling if the sequences are aligned, because it claims unnecessary memory and time. This special mode is selected if the reference and target sequences have the same size. However, there are situations where, despite the sequences having equal size, they are not aligned and, therefore, this mode would produce poor compression results. This flag turns off this special mode, forcing the use of the hash table.
4. This parameter is also directly related to the encoding performance of the method and may be adjusted according to the degree of similarity between the reference and target sequences. Basically, it controls how many failures we allow the copy model predictor to have before it is declared useless and is replaced by another. For target sequences that are reasonably similar to the reference sequence, it is safe to use a value greater than the default (zero), because, despite some prediction failures due, for example, to SNPs, it is expected that a given copy model is useful for long segments. When the target and reference

sequence are considerably dissimilar, then it is better to replace the copy model as soon as it starts failing the predictions.

5. This note applies both to the reference and target sequences. It also applies to the reference sequence used by the decoder. The files can be plain text or compressed by “gzip.” FASTA files are handled by ignoring all lines starting with the “>” character. All new line characters are also ignored. All other characters are considered as belonging to the sequence to be encoded.
6. The output file will contain only the characters that have been considered by the encoder as part of the sequence. Therefore, FASTA formatting is lost. Also, the output file will be made of a single line, i.e., without line breaks. If this option is not present, the decoder will run normally, but without producing an output file.
7. The probabilities of the symbols may change along the message. In fact, this is what makes arithmetic coding powerful, because the statistical model of the source can be continuously updated as encoding proceeds. The only requirement is that the encoder and decoder must be synchronized, i.e., for a given position in the message, the statistical model has to produce exactly the same probability distribution both at the encoder and at the decoder.
8. In practice, arithmetic coding cannot be implemented using directly the procedure that we have described, because the floating point precision would be exhausted quickly. The first practical arithmetic coding algorithm was introduced by Rissanen in 1976 [19]. Further detailing how arithmetic coding is implemented is out of the scope of this chapter. Interested readers can obtain more details in e.g. [20].
9. GReEn does not compress the reference sequence. If needed, it can be compressed using a general purposed compression program, such as “gzip,” or a specialized DNA compression tool [11, 17].
10. Note that, whereas the $\theta(c)$ values are fixed for a given target sequence, the $P_n(c)$ values usually change along the coding process. Moreover, the $\theta(c)$ values are the only ones that are known in advance to both the encoder and the decoder. Therefore, they are communicated to the decoder before decoding starts. All other statistical information related to the target sequence is collected as encoding (decoding) proceeds, i.e., the process is causal. Hence, for a certain position n in the target sequence, the encoder can use only past information for inferring the statistics.
11. Note that characters of the reference sequence that do not appear in the target sequence do not belong to \mathcal{C} .

12. The first two branches of Eq. 2 correspond to Laplace probability estimators of the form

$$P(\mathcal{E}_k) = \frac{N_{\mathcal{E}_k} + 1}{\sum_{k=1}^K N_{\mathcal{E}_k} + K}, \quad \mathcal{E}_k \subset \mathcal{C},$$

where the \mathcal{E}_k s form a set of K collectively exhaustive and mutually exclusive events, and $N_{\mathcal{E}_k}$ denotes the number of times that event \mathcal{E}_k has occurred in the past. In Eq. 3 we considered three events, namely, $\mathcal{E}_1 = \{p_n^1\}$, $\mathcal{E}_2 = \{p_n^2\}$, and $\mathcal{E}_3 = \mathcal{C} \setminus \{p_n^1, p_n^2\}$. The third branch of Eq. 3 defines how the probability assigned to \mathcal{E}_3 , i.e., $1 - P(\mathcal{E}_1) - P(\mathcal{E}_2)$, is distributed among the individual characters of \mathcal{E}_3 . This distribution is proportional to the relative frequencies of the characters, $\theta(c)$, after discounting the effect of treating p_n^1 and p_n^2 differently.

5 Acknowledgments

This work was partially funded by FEDER through the Operational Program Competitiveness Factors—COMPETE and by National Funds through FCT—Foundation for Science and Technology in the context of the projects FCOMP-01-0124-FEDER-010099 (FCT reference PTDC/EIA-EIA/103099/2008) and FCOMP-01-0124-FEDER-022682 (FCT reference PEst-C/EEI/UI0127/2011). Sara P. Garcia acknowledges funding from the European Social Fund and the Portuguese Ministry of Education and Science.

References

1. Grumbach S, Tahi F (1993) Compression of DNA sequences. In: Proceedings of the data compression conference, DCC-93, Snowbird, pp 340–350
2. Rivals E, Delahaye J-P, Dauchet M, Delgrange O (1996) A guaranteed compression scheme for repetitive DNA sequences. In: Proceedings of the data compression conference, DCC-96, Snowbird, p 453
3. Loewenstern D, Yianilos PN (1997) Significantly lower entropy estimates for natural DNA sequences. In: Proceedings of the data compression conference, DCC-97, Snowbird, March 1997, pp 151–160
4. Matsumoto T, Sadakane K, Imai H (2000) Biological sequence compression algorithms. In: Dunker AK, Konagaya A, Miyano S, Takagi T (eds) Genome informatics 2000: proceedings of the 11th workshop, Tokyo, pp 43–52
5. Chen X, Kwong S, Li M (2001) A compression algorithm for DNA sequences. *IEEE Eng Med Biol Mag* 20:61–66
6. Chen X, Li M, Ma B, Tromp J (2002) DNA-Compress: fast and effective DNA sequence compression. *Bioinformatics* 18 (12):1696–1698
7. Manzini G, Rastero M (2004) A simple and fast DNA compressor. *Softw Pract Exp* 34:1397–1411
8. Korodi G, Tabus I (2005) An efficient normalized maximum likelihood algorithm for DNA sequence compression. *ACM Trans Inform Syst* 23(1):3–34
9. Behzadi B, Le Fessant F (2005) DNA compression challenge revisited. In: Combinatorial pattern matching: proceedings of CPM-2005. LNCS, vol 3537. Jeju Island, June 2005. Springer-Verlag, New York, pp 190–200

10. Korodi G, Tabus I (2007) Normalized maximum likelihood model of order-1 for the compression of DNA sequences. In: Proceedings of the data compression conference, DCC-2007, Snowbird, March 2007, pp 33–42
11. Cao MD, Dix TI, Allison L, Mears C (2007) A simple statistical algorithm for biological sequence compression. In: Proceedings of the data compression conference, DCC-2007, Snowbird, March 2007, pp 43–52
12. Giancarlo R, Scaturro D, Utro F (2009) Textual data compression in computational biology: a synopsis. *Bioinformatics* 25(13): 1575–1586
13. Pinho AJ, Neves AJR, Afreixo V, Bastos CAC, Ferreira PJSG (2006) A three-state model for DNA protein-coding regions. *IEEE Trans Biomed Eng* 53(11):2148–2155
14. Pinho AJ, Neves AJR, Ferreira PJSG (2008) Inverted-repeats-aware finite-context models for DNA coding. In: Proceedings of the 16th European signal processing conference, EUSIPCO-2008, Lausanne, August 2008
15. Pinho AJ, Neves AJR, Bastos CAC, Ferreira PJSG (2009) DNA coding using finite-context models and arithmetic coding. In: Proceedings of the IEEE international conference on acoustics, speech, and signal processing, ICASSP-2009, Taipei, April 2009, pp 1693–1696
16. Pinho AJ, Pratas D, Ferreira PJSG (2011) Bacteria DNA sequence compression using a mixture of finite-context models. In: Proceedings of the IEEE workshop on statistical signal processing, Nice, June 2011
17. Pinho AJ, Ferreira PJSG, Neves AJR, Bastos CAC (2011) On the representability of complete genomes by multiple competing finite-context (Markov) models. *PLoS One* 6(6): e21588
18. Pinho AJ, Pratas D, Garcia SP (2012) GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Res* 40(4):e27
19. Rissanen J (1976) Generalized Kraft inequality and arithmetic coding. *IBM J Res Dev* 20 (3):198–203
20. Sayood K (2006) Introduction to data compression, 3rd edn. Morgan Kaufmann, San Francisco

Deep Sequencing Data Analysis

Shomron, N. (Ed.)

2013, X, 234 p. 81 illus., 43 illus. in color., Hardcover

ISBN: 978-1-62703-513-2

A product of Humana Press