BIOINFORMATICS APPLICATIONS NOTE

Vol. 27 no. 6 2011, pages 860–862
doi:10.1093/bioinformatics/btr014

Sequence analysis

Advance Access publication January 19, 2011

Compression of DNA sequence reads in FASTQ format

Sebastian Deorowicz[1,*] and Szymon Grabowski[2]

[1]Institute of Informatics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice and [2]Computer Engineering Department, Technical University of Łódź, Al. Politechniki 11, 90-924 Łódź, Poland

Associate Editor: Dmitrij Frishman

ABSTRACT

Motivation: Modern sequencing instruments are able to generate at least hundreds of millions short reads of genomic data. Those huge volumes of data require effective means to store them, provide quick access to any record and enable fast decompression.

Results: We present a specialized compression algorithm for genomic data in FASTQ format which dominates its competitor, G-SQZ, as is shown on a number of datasets from the 1000 Genomes Project (www.1000genomes.org).

Availability: DSRC is freely available at http://sun.aei.polsl.pl/dsrc.

Contact: sebastian.deorowicz@polsl.pl

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

Received on November 16, 2010; revised on December 21, 2010; accepted on January 3, 2011

1 INTRODUCTION

One of the main tasks of bioinformatics is to collect and analyze huge genomic data. Sequencing instruments are able to generate at least hundreds of millions of short reads, accompanied with annotations, like quality scores denoting uncertainties in sequence identification processes. Read identifications and other descriptions, e.g. unique instrument names, are also kept in the same file.

We propose an algorithm for effective compression of genomic reads of the described kind in the widely accepted FASTQ input format, and show experimental results suggesting its supremacy over existing solutions. Our scheme provides fast random access to individual records of the dataset and is capable of finding repeating sequences in reads strings.

We are not aware of any direct competitors to our algorithms, except for G-SQZ (Tembe *et al.*, 2010), presented in the experimental section. There is a number of DNA-compressing algorithms but they deal with genomic (and usually not annotated) sequences rather than DNA reads. The used compression techniques include detecting exact and inexact repeats (Chen *et al.*, 2002), complementary palindromes (Grumbach and Tahi, 1994), higher-order coding, and more. Unfortunately, most of the known algorithms are computationally intensive and have been tested only on small datasets (e.g. a few MB's). A recent trend is to exploit similarities between sequences of individuals of the same species (Kuruppu *et al.*, 2010), but this is also a problem different to the one we consider here. For a survey of compression techniques in computational biology, see (Giancarlo *et al.*, 2009).

*To whom correspondence should be addressed.

2 METHODS

Data compression methods can be divided into dictionary and statistical methods. The former encode repeating subsequences of the input text as references to subsequences seen earlier. The latter encode each input character on the basis of gathered statistics, possibly involving its context.

The FASTQ format so far lacks a clear definition and several incompatible variants of it are in use (Solexa, Sanger, Illumina 1.3+ and more). The need for format standardization was advocated in (Cock *et al.*, 2009) and a format proposal presented therein. Another, similar, proposal was given by Institute for Integrative Genome Biology of University of California (http://illumina.ucr.edu/ht/documentation/standardized-fastq-format-aka-fastq2, accessed on December 15, 2010), where e.g. line wrapping after 80 characters and a few other restrictions are obligatory. Our solution is compatible with those two recent specifications. FASTQ data can naturally be perceived as ordered collections of records, each consisting of three streams: title information, DNA sequence (read), quality scores. In our solution the three FASTQ streams within a record are processed (almost) independently. Also, we impose a hierarchical structure of the compressed data, to provide fast random access to records. To this end, we divide the dataset into blocks of b records ($b=32$ by default) and also group blocks into superblocks of size s blocks each ($s=512$ by default). Our format (but not the current implementation) allows for incremental growth of the archive, where new records are added at the end. Each superblock is compressed independently while blocks within a superblock share statistics but apart from that are also independent. In other words, in order to decompress a random block, the superblock it belongs to is first located, the block offset within the superblock is found, the appropriate statistical data common to all blocks in the superblock are read, and finally the block's data are extracted. Extracting a random record requires five disk seek operations (details are given in Section 2 of the Supplementary Material). Our solution also supports extraction of the whole superblock (note that arbitrary ranges of contiguous records can be composed of one or several successive superblocks, where possibly a prefix of the first superblock and a suffix of the last superblock are truncated). This strategy is much faster than naïve invocation of the extraction procedure for individual records in the superblock.

In the following subsections we briefly present the techniques used for compressing the three FASTQ data streams.

2.1 Title lines

The Title lines start with the @ character followed with a sequence identifier and (optional) description, which can store information like unique instrument name, flowcell lane, read length etc. There is no specification, or even one commonly accepted convention, of the description format, hence we choose a heuristic approach. We treat the Title lines as a concatenation of several fields with separators (blank space, colon, period and a few more) between them. The first Title line in a superblock specifies the number of fields and the separators between them for the rest of the superblock (the rare cases of varying number of fields or heterogeneous layout of separators are also handled). Then, each field is inspected separately, with statistics gathered usually on the superblock level. The techniques used here include: compact

Downloaded from https://academic.oup.com/bioinformatics/article/27/6/860/235075 by guest on 20 August 2022

© The Author 2011. Published by Oxford University Press. All rights reserved. For Permissions, please email: journals.permissions@oup.com

encoding of constant (fixed) fields, recognition of numeric and non-numeric fields, efficient encoding of columns with fixed characters in non-numeric fields, detection of the numeric fields amenable to differential coding, entropy coding.

We explain our processing of the Title lines on an excerpt from the beginning of the SRR003177 dataset. The input lines are as follows.

```
@SRR003177.1 FC60WVV01ASZMI length=42
@SRR003177.2 FC60WVV01AEUTT length=213
@SRR003177.3 FC60WVV01AIMMC length=121
@SRR003177.4 FC60WVV01AHKQQ length=497
@SRR003177.5 FC60WVV01AVYZ0 length=174
```

The first Title line is split on the separators, which are a period, a blank space, a blank space and an '=' character. Clearly, the number of obtained fields is five. Two constant fields are detected, and their values, SRR003177 and length, respectively, stored only once (of course, this check is performed for the whole superblock). The other fields are divided into numeric (the second and the fifth field) and non-numeric (the third field) ones. Numeric fields consist of digits only, with non-zero at the beginning if the digit sequence length is greater than 1. For each numeric field it is then decided if it should be differentially (delta) encoded. We assume that a field benefits from delta coding if the difference of the maximum and the minimum delta between the field values in two successive records is smaller than the difference between the maximum and the minimum field values without delta encoding; the extremes are taken over the whole superblock. In our small example, the second field produces all delta values equal to 1, while the extreme 'raw' values are 5 and 1, hence delta coding would be applied. For the last field, delta coding seems harmful (deltas vary from -323 to 376, while the range for raw values is narrower). The resulting (i.e. delta-transformed or unchanged) sequence of numbers is then compactly encoded, together with the minimum and the maximum field value. The chosen encoding variant depends on the resulting range of values. For ranges smaller than 512, order-0 Huffman encoding is applied. Otherwise, the values are encoded using the minimum required number of bits, same for each. For example, if the field is not delta-encoded and the minimum and the maximum values are 200 and 1190, respectively, then 10 bits per number are used, since all numbers from the range $[200, 1190]$ can be encoded using 10 bits. In our example, the second field is even more compactly represented, because all delta values are equal and thus the delta is stored only once.

The processing of non-numeric fields is different. Commonly for such fields the characters at some positions ('columns') are fixed. In our example, for the third field the fixed positions comprise the prefix of length 10 (FC60WVV01A). As in general any positions can be fixed, we store a bit vector at the superblock level, denoting the fixed positions, and the fixed characters are stored only once. This bit vector, which we also call a Hamming mask, is of length equal to the length of the respective field in the first record. The characters in non-fixed positions are statistically encoded, using a separate order-0 Huffman model for each field column.

We notice that fields may be of variable length, which is not a problem (characters at the trailing positions cannot simply be considered fixed). Still, if a field is of variable length, its minimum and maximum length must be recorded, to compactly encode the field length for each record, in the manner of compact encoding of numerals, presented above. If the field length is constant, it is recorded only once.

Finally, we note that if the Title field format is inconsistent across different records (which may happen if the data come from different experiments) then our encoding of those data is relatively inefficient.

2.2 DNA

The number of distinct DNA bases is usually four, but the (rare) occurrences of N and other symbols (of the IUPAC list) for inconclusive identification make the simple idea of packing bases into bytes at least cumbersome. Our solution of this issue is to move the extra symbols from the DNA stream into the quality stream; the range for qualities is (at most) 94, hence over 160 byte values can be utilized for this purpose. Having got a symbol from the augmented alphabet, we mix it with its quality score, assuming the score is one of its 8 lowest values ($[33, 40]$; ambiguous symbols should have low-quality scores), which produces 120 values, easily fitting the upper half of the byte range. If the superblock contains at least one symbol not in the allowed set (of 4 standard bases and 15 extra symbols) or at least one of its extra symbols is accompanied with quality score above 40, the whole DNA stream in this superblock is Huffman-encoded. Such a situation is rare though. Still, our algorithm does more than mere packing bases in fours in bytes (and handling extra symbols). From the sequencing process it is obvious that some some sequence reads should overlap, so LZ77-style encoding of them is applied. Below we present an outline of our LZ77-style compression scheme while more details are in the Supplementary Material.

The LZ77-based compression algorithms basically need two data structures: the buffer with the text already read, in which matched phrases are found, and an indexing structure (a dictionary) for fast match search, often implemented as a hash array, which is also used in our solution.

One of the assumptions we made is that encoded matches are read prefixes. If for the current read an LZ-match is found, then this read is not indexed, i.e. cannot be a reference to a future read. This prevents from possibly very long reference chains, detrimental for fast record access. Another assumption is that it pays not to hash the current read if it is not hopeful enough to yield a significant gain as a reference to a future read. This estimation is based on the corresponding quality scores; if they are low, then also chances for a match are low and the read is not indexed and searching for its match is canceled.

The hashes are calculated over read subsequences of length 36, which obviously is also the minimum match length threshold. Clearly, for much shorter matches it is cheaper to encode the literals one by one. Even for 'prospective' reads, not all of its subsequences are added to the hash array. We take some care to minimize space usage; for example, we prevent adding new entries if the number of collisions for a given hash is large enough.

2.3 Quality scores

We have noticed that there are basically three sorts of quality streams: (i) quasi-random with mild dependence of quality score position onto the score symbol distribution; (ii) similar to the previous one but with many quality strings ended with several # characters; and (iii) the last one, with strong local correlations within individual records, which can be exploited with simple means, like run-length encoding, which precedes a statistical coding phase. The compressor detects the actual case and chooses the appropriate processing variant. In the case (ii), a bit flag per record is spent to tell if its quality string ends with hashes. If so, the trailing hashes are removed and their length stored. With this difference, the cases (i) or (ii) are handled in the same way: the quality data of a given block are split into as many streams as the length of the longest quality sequence in the superblock. The streams are order-0 Huffman-encoded using frequencies stored on the superblock level. In case (iii), the concatenated quality sequences of the block are split into run heads and run lengths, and encoded statistically. For example, the input string BCCBCBC@BBBCB would be split into BCBCBC@BCB (run heads) and $0, 1, 0, 0, 0, 0, 0, 2, 0, 0$ (run lengths, decreased by 1 as lengths must be at least one). The run lengths are limited to 256 to fit a single byte; even longer runs (which are rare) are split into several shorter ones. The resulting head and length streams are then order-1 Huffman-encoded, using their respective statistics stored on the superblock level.

3 IMPLEMENTATION AND RESULTS

The C++ implementation of our algorithm is called DSRC (DNA Sequence Reads Compressor). In order to evaluate its practical assets: compression ratio, compression speed, decompression speed and random access (record extraction) time, we compared it against two popular general-purpose compressors, gzip 1.3.5 and bzip2 1.0.3 (both run with switch -9), and the specialized DNA reads compressor G-SQZ 0.6 (Tembe *et al.*, 2010).

Downloaded from https://academic.oup.com/bioinformatics/article/27/6/860/235075 by guest on 20 August 2022

Table 1. Compression results.

Dataset	Type	Size [GB]	gzip			bzip2			G-SQZ				DSRC				DSRC-LZ			
			Ratio	c-sp [MB/s]	d-sp [MB/s]	Ratio	c-sp [MB/s]	d-sp [MB/s]	Ratio	c-sp [MB/s]	d-sp [MB/s]	e-t [ms]	Ratio	c-sp [MB/s]	d-sp [MB/s]	e-t [ms]	Ratio	c-sp [MB/s]	d-sp [MB/s]	e-t [ms]
SRR001471	LS454	0.22	3.24	2.2	90.7	3.94	7.8	17.3	–	–	–	–	4.44	36.4	41.3	7	4.63	23.2	38.7	7
SRR003177	LS454	1.20	3.17	2.0	89.9	3.81	7.7	16.9	–	–	–	–	4.24	40.5	42.0	8	4.38	20.4	39.0	8
SRR003186	LS454	0.89	2.98	1.9	86.4	3.59	7.4	16.0	–	–	–	–	4.02	36.4	37.6	7	4.08	18.6	35.6	7
SRR007215_1	SOLiD	0.70	4.19	9.1	92.7	5.21	4.2	23.9	4.71	25.2	4.8	178	6.51	21.0	35.8	8	6.51	21.0	35.8	8
SRR010637	SOLiD	2.09	3.48	7.6	96.1	4.25	4.8	21.0	3.98	22.1	4.0	117	5.13	21.7	31.2	8	5.13	21.7	31.2	8
SRR013951_2	SOLEXA	3.19	2.41	3.9	81.0	2.81	6.4	15.3	2.77	15.2	2.7	193	3.23	26.6	29.1	7	3.29	17.7	28.3	7
SRR014961_2	SOLiD	40.90	3.55	6.8	84.3	4.37	5.0	21.1	3.98	21.2	3.9	113	5.49	19.9	31.4	9	5.49	19.9	31.4	9
SRR027520_1	SOLEXA	4.81	2.88	3.7	81.5	3.42	6.5	17.0	3.03	16.4	2.9	178	3.98	27.5	34.5	7	4.05	15.4	33.3	7
SRR027520_2	SOLEXA	4.81	2.81	3.7	78.4	3.33	6.3	16.7	3.03	16.5	3.0	168	3.90	27.3	33.0	7	3.97	15.8	31.9	7
Average for SOLiD and SOLEXA			3.22	5.8	85.7	3.90	5.5	19.2	3.58	19.4	3.5	158	4.71	24.0	32.5	6	4.74	18.6	32.0	6
Average for all			3.19	4.6	86.8	3.86	6.2	18.4	–	–	–	–	4.55	28.6	35.1	7	4.61	19.3	33.9	7

The columns 'ratio' denote how many times the compressed output is smaller than the original file. 'c-sp' and 'd-sp' stand for compression speed and decompression speed, respectively. 'e-t' stands for average extract time for a single record.

Basically, G-SQZ applies order-0 Huffman coding on combined bases and respective qualities. Many details of its internal work cannot be inferred from the paper and we were unable to obtain program sources. G-SQZ cannot handle datasets with variable-length reads. DSRC is run twice: with and without LZ-matches on the reads stream. The test data comprise nine files from 1000 Genomes Project; more results are shown in the Supplementary Material. The test machine was an AMD Opteron™8378 2.4 GHz, using a single core, 16 GB of RAM, under Red Hat Enterprise Linux Server 5.4. Time measurements were performed with the Unix time command.

For all tested compressors we present the ratio between the original and the compressed file size (so, higher is better), and compression and decompression speed (see Table 1). Also, for the specialized compressors, G-SQZ and DSRC, we show the random record extract time. First, we notice that in almost all cases DSRC is able to reduce the input datasets from four to over six times. Second, adding LZ-matching generally helps, although sometimes the reads are below the minimum match length and then this idea becomes effectively turned off. On other files, the overall gain from this idea is usually about 2–4%. Even without LZ-matching our algorithm achieves significantly higher compression than the competitors, on each test file.

As expected, LZ-matches harm the speed, but in decompression the loss rarely exceeds 5%. On the other hand, the compression time grows by 60–105% (the milder slow-down in the reported averages comes from the fact that on several files LZ-matching was actually not used, as explained above). In real numbers, the decompression speed on the test machine is about 30–40 MB/s while the compression speed varies from 15 MB/s to 23 MB/s with LZ-matching and from 20 MB/s to 40 MB/s without. We note that even our slower variant is several times faster in compression than gzip and bzip2. This is not the case with G-SQZ which is comparable in compression speed with DSRC with LZ-matching, but usually slower than DSRC in the faster mode. As for the decompression speed, gzip is unsurpassable, running at 80–100 MB/s. The other competitors are clearly slower than DSRC here. Over an order of magnitude difference between DSRC and G-SQZ can be noticed in the random access test, where DSRC makes it possible to extract any individual record in less than 10 ms. Finally, the time to extract a whole random superblock depends on the read lengths, but typically varies from 2 s to 5 s with LZ-matching and from 0.1 s to 0.3 s without.

4 CONCLUSION

We presented a specialized compressor, DSRC, for genomic sequences with quality scores, clearly superior both in compression efficiency and performance to its only known competitor for this kind of data, G-SQZ. Comparison with two well-known general-purpose compressors also puts our solution in a favorable position. A carefully designed two-level structure of compressed records made it possible to extract any individual record in less than 10 ms.

Funding: Polish Ministry of Science and Higher Education under the project (N N516 441938) in part.

Conflict of Interest: none declared.

REFERENCES

Chen,X. *et al*. (2002) DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, **18**, 1696–1698.

Cock,P.J.A. *et al*. (2009) The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res.*, **38**, 1767–1771.

Giancarlo,R. *et al*. (2009) Textual data compression in computational biology: a synopsis. *Bioinformatics*, **25**, 1575–1586.

Grumbach,S. and Tahi,F. (1994) A new challenge for compression algorithms: genetic sequences. *Inf. Process. Manage.*, **30**, 875–886.

Kuruppu,S. *et al*. (2010) Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. *Lect. Notes Comput. Sci.*, **6393**, 201–206.

Tembe,W. *et al*. (2010) G-SQZ: compact encoding of genomic sequence and quality data. *Bioinformatics*, **26**, 2192–2194.

Downloaded from https://academic.oup.com/bioinformatics/article/27/6/860/235075 by guest on 20 August 2022