Supplementary Material

February 27, 2018

In this supplementary material we provide a detailed description of the method employed by METHCOMP, some implementation details, and more experimental results.

Methods

We seek to compress DNA methylation data stored in the bedMethyl format. The bedMethyl format is a "modified" BED format consisting of 11 columns, with the first nine following the same format as BED files, and the last two columns being methylation specific. In particular, each column (c) contains the following information:

- c1: **chrom**, nreference chromosome or scaffold
- c2: **chromStart**, the start position in chromosome
- c3: **chromEnd**, the end position in chromosome
- c4: **name**, the name of item
- c5: **score**, the score of item
- c6: **strand**, the strand of item
- c7: **thickStart**, the start position where the feature is drawn thickly
- c8: **thickEnd**, the end of position of thick displayed feature
- c9: **itemRgb**, the color mapping value of item
- c10: **coverage**, the number of reads
- c11: **percentage**, the percentage of methylated reads

The columns are a combination of strings and integers, and each of them encodes different information. Thus, we use different compression strategies for different columns. The columns containing string values include *chrom* (c1), *name* (c4), and *strand* (c6). The columns in the bedMethyl format are sorted by reference chromosome (c1), and within a given chromosome, by the starting position (c2). In addition, a given reference chromosome may contain millions of items. With these assumptions, it is reasonable to use run-lengh coding for column 1. The value of column 4 (*name*) is the same for consecutive records, and thus for the same reasons as for column 1, we use run-length encoding. The *strand* column indicates if methylation occurs on the sense or antisense strand (it can take values in $\{+, -, .\}$, where "." corresponds to "unknown origin"). The strand value is most likely independent of the value of any other columns, and thus we use an adaptive arithmetic coding with a dictionary of size 3 to compress this column. Some of the remaining columns, which contain integers, are highly correlated. Hence, we can discard the values in column 3 *chromEnd* as these values are always equal to the values of column 2 *chromStart* plus one. In addition, in the WGBS pipeline, which is the one we use, columns 7 (*thickStart*) and 8 (*thickEnd*) are identical to *chromStart* and *chromEnd*, respectively, and thus they can also be discarded. Columns 5 (*score*) and 10 (*coverage*) are also highly correlated, with the score being equal to the min value between the coverage and 1000. Thus we encode only the *coverage* column. Finally, column 9 *itemRgb* is a deterministic color mapping that depends on the *percentage* value (c11), and hence can be easily retrieved from the column *percentage*. In summary, from the integer columns, we only need to encode columns 2 (*chromStart*), 10 (*coverage*), and 11 (*percentage*).

The values of column 2 (*chromStart*) are sorted within each reference chromosome (c1), and we first apply differential coding as a preprocessing step to potentially reduce the alphabet size and make the distribution skewed towards smaller values. The resulting values are then encoded by means of adaptive arithmetic encoding. The values of column 10 (*coverage*) are also encoded by means of adaptive arithmetic encoding. Finally, column 11 (*percentage*) takes values between 0 and 100, and thus we encode it with an adaptive arithmetic encoder of alphabet size 101.

The strategies employed by METHCOMP to compress the different columns that constitute the bedMethyl file are summarized below (all other columns are discarded during compression):

- c1: **chrom**, run-length
- c2: **chromStart**, differential encoding followed by adaptive arithmetic
- c4: **name**, run-length
- c6: **strand**, adaptive arithmetic
- c10: **coverage**, adaptive arithmetic
- c11: **percentage**, adaptive arithmetic

Implementation Details

We implemented the METHCOMP encoder and decoder in C++, and the software may be retrieved from https://github.com/jianhao2016/METHCOMP. Instructions on how to run the code are available at the same site. Notice that in order to compile the source, the standard version C++11 [2] of C++ is required. This version has a number of versatile features amenable for use in large scale data compression.

The Encoder

The encoder reads the bedMethyl file line by line, and then parses each line into a Row object that keeps the raw data retrieved from each line in their default type. For example, *chrom* (c1), *name* (c4) and *strand* (c6) are stored as strings, while all the remaining seven columns except for *itemRgb* (c9) are stored as integers; *itemRgb* (c9) is maintained as a string since it is discarded during subsequent processing and given that there are only 11 different colors in the color map, a saved string is easier to retrieve by an enumerate class than individual numbers. Each Row is sliced into an InputData class, which performs data preprocessing such as mapping and differentiation. The InputData of each row only contains the necessary columns described earlier in this section. The two run-length codes operating on *chrom* and *name* are updated inside the InputData class while lines are being fed to the encoder.

Each InputData is inherited from an ArithmeticInt class, so every column in the integer part of InputData is sent to a separate adaptive arithmetic encoder. The values in *strand* (c6) and *percentage* (c11) are fixed to 3 and 101, respectively. As a result, the arithmetic encoder can use fixed dictionaries of respective size 3 and 101 on them. The values in *chromStart* (c2) and *coverage* (c10) have an unknown range which depends on the results of the experiments themselves. Keeping a dictionary of size 2^{32} to cover all the possible integer values is too expensive and impractical. One solution is to update the dictionary while encoding so as to keep the size of the dictionary at a minimum, but this approach requires keeping additional information that indicates the update and frequent extensions of the dictionary, which consumes significant computational resources. Given that the file size is already very large, a simpler method is preferred. This is why we opted for "slicing" the 32-bit integer into 4 different 8-bit integers and using a fixed dictionary of size 256 for each of them. Since the higher register bits do not change as often as the lower register ones, the frequency counts used in the corresponding arithmetic encoders will concentrate around several small values. This enables adding only a few bits to each integer while keeping the model as simple as possible.

The encoding procedure is depicted in Fig. 1. The input is a bedMethyl file, and after compression, we obtain three files. Two of them are obtained from run-length coding, namely "outfileChrom" and "outfileName", and are saved as plain text since they are both of size less than 100 KB (for the files tested). The third file is the output bit stream from the arithmetic encoders. The encoder consecutively goes over the Row of these encoders and outputs them into the same bit stream.

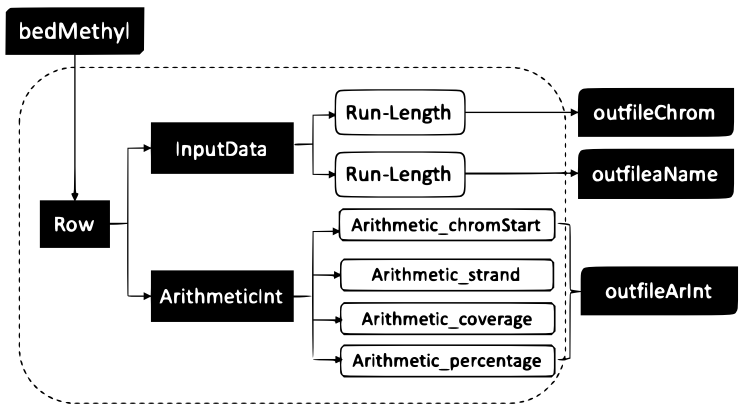

Figure 1: Schematic of the METHCOMP encoder.

The Decoder

From the data flow perspective, the decoder performs mirror image processing with respect to the encoder: the decoder takes three input files and feeds them to different decoder subunits to generate the components needed for reassembling the final output. The first step is arithmetic decoding of the files bit-by-bit and recovering ArithmeticInt instances. Afterwards, lines from the run-length coded files are used to form the InputData instance. At the third stage, the decoder combines these two instance into a Row instance and writes a row into the final output file.

The decoder terminates once the following three conditions are fulfilled: an EOF symbol is decoded from the arithmetic decoder; the last line in outfileChrom is reached; the last line in outfileName reached.

In the ideal case, those three conditions will be achieved simultaneously, since the encoders also terminate at the same time. If any of those conditions are met before the others, then the decoder will produce an “Error message” informing the user that the input files do not match each other.

Experimental results

To test the performance of our compression method, we used the ENCODE assays specified in Table 3. In the main text, we used the file suffix to denote each individual file in assay *ENCSR835OJU*, since they all have identical prefix. For example, 167OJH and 327MVH correspond to ENCFF167OJH and ENCFF327MVH, respectively. METHCOMP is tested in terms of the compression ratio, defined in Table 1, and with respect to execution time (Table 2). Table 1 and 2 can be found in the main document. For comparison, in the same tables we listed the same performance results for gzip (Apple version 272). All experiments were performed on an Intel i7 laptop with 16GB of RAM and 500GB of disk memory. METHCOMP only uses one thread and occupies less than 2GB of memory, and it can run on any other 64 bit machine.

assay name	description
ENCSR835OJU	Mus musculus C57BL/6 heart embryo Biosample
ENCSR888JFA	Mus musculus C57BL/6 forebrain embryo
ENCSR351IPU	Homo sapiens hepatocyte originated from H9
ENCSR656TQD	Homo sapiens mammary epithelial cell female and female adult

Table 3: Description of assays used in main text.

We used an integer implementation of arithmetic coding [12], which instead of using a floating point representation for the interval $[0, 1)$ employs a mapping onto n bits, and hence reduces the precision. Therefore, one needs to iteratively output bits and rescale the interval so as to make the calculated values fit into the allocated n-bit register. In order to avoid multiplication overflow problems, the number of bits should not be too large, and we settled on a 32-bit representation which is compatible with a 64-bit machine. For a 32-bit machine, this value may have to be decreased in case that arithmetic errors occur.

From the results in Tables 1 and 2, we see that METHCOMP provides storage savings exceeding **97%** on average. With respect to compression ratio improvements, METHCOMP offers roughly **7.5**× better results than gzip, with the corresponding numerical values for file 327MVH (which shows the most improvement) being 49.55 and 5.65, respectively. With respect to compression/decompression speed, METHCOMP introduces larger decompression delays compared to gzip due to parsing and reassembling of the file line by line. From the results in Table 4, the compression time of METHCOMP is roughly twice that of gzip, while the decompression time is more than three times higher than that of gzip. The average compression and decompression speeds of METHCOMP and gzip are 11.61 and 98.31, and 22.91 and 356.18 MB/s, respectively. It is worth pointing out that compression requires significantly more time than decompression, so that the overall time complexities of the schemes are only a factor of two apart. Furthermore, given that the decompression time is higher for METHCOMP than gzip, amounting to roughly 8 min for a file of size 50 GB, METHCOMP may be best suited for archival storage as it also offers immense file size reductions. The decompression speed of METHCOMP is still much faster than the download speed.

file name	original size(GB)	compression time(s)		compression speed(MB/s)		decompression time(s)		decompression speed(MB/s)	
		gzip	METHCOMP	gzip	METHCOMP	gzip	METHCOMP	gzip	METHCOMP
167OJH	13	567	1201	**23.48**	11.08	36	135	**369.78**	98.61
327MVH	48	2058	4277	**23.88**	11.49	126	460	**390.10**	106.85
428AXW	2.6	122	243	**21.82**	10.96	8	30	**332.80**	88.75
677YTO	13	572	1200	**23.27**	11.09	38	137	**350.32**	97.17
751DLO	2.6	128	241	**20.80**	11.05	7	30	**380.34**	88.75
945JPE	48	2060	4405	**23.86**	11.16	127	473	**387.02**	103.92
ENCSR1	128.2	5752	10921	**22.11**	12.02	379	1305	**331.71**	100.60
ENCSR2	139	6070	12407	**23.67**	11.47	381	1513	**368.10**	94.08
ENCSR3	138.9	6257	12119	**23.02**	11.74	388	1472	**356.51**	96.63
average				**22.91**	11.61			**356.18**	98.31

Table 4: Comparison of compression speeds of gzip and METHCOMP. The speed is computed according to $\left(\frac{\text{original size}}{\text{time taken by teh task}}\right)$

To ensure a completely fair comparison between METHCOMP and gzip, we also ran gzip on what we refer to as the “extracted” files. Extracted files contain only those columns effectively used by METHCOMP, as outlined at the introduction of Section Methods. The results for a selected subset of the tested files are shown in Table 5 and Table 6. Specifically, the files 365XZL and 506SUF were obtained from *ENCSR888JFA*, while the files 170PBE and 487XOB were obtained from *ENCSR351IPU*. Even when operating on the preprocessed files with extracted columns, METHCOMP still offers an approximately **3**× better compression ratio than gzip (improvement may be observed in Table 6). At the same time, the compression/decompression times of the two methods are comparable in this mode, as the largest fraction of the processing time is spent on extracting repetitive columns and reinserting them during decompression. METHCOMP offers further improvements in the decompression speed due to the use of data blocking.

file name	original size(GB)	extracted size(GB)	compressed size(GB)	compression ratio	compression time(s)	compression speed(MB/s)	decompression time(s)	decompression speed(MB/s)
365XZL	2.6	1.0	0.24	12.33	219	12.16	149	17.87
506SUF	13	5.1	0.88	14.69	1126	11.82	784	16.98
170PBE	15	5.5	1.00	15.00	1259	12.20	843	18.22
487XOB	3.5	1.3	0.27	12.89	292	12.27	199	18.01
average				13.73		12.11		17.77

Table 5: Performance of gzip when applied to the subset of columns effectively used by METHCOMP.

file name	original size(GB)	compressed size(GB)	compression ratio	improvement	compression time(s)	compression speed(MB/s)	decompression time(s)	decompression speed(MB/s)
365XZL	2.6	0.086	30.25	2.45	230	11.58	31	85.88
506SUF	13	0.316	41.09	2.80	1145	11.63	143	93.09
170PBE	15	0.350	42.91	2.87	1260	12.19	170	90.35
487XOB	3.5	0.110	31.72	2.46	304	11.79	44	81.45
average				2.77		11.88		89.99

Table 6: Performance of METHCOMP when applied to same files in Table 5.

Discussion

The prevalent trend in compression practice in the field of genomics is to apply universal methods that can compress arbitrary file formats to an acceptable level. Unfortunately, with the surge of Big Data platforms in biological and medical research, it has become imperative to devise significantly more space efficient, specialized algorithms for the underlying data. In this direction, new software suites for FASTQ and VCF files have been developed for genomic data storage [3, 4, 5, 10, 11], along with a number of specialized methods for compressing metagenomic data, RNA-seq and ChIP-seq measurements in lossless and lossy modes [6, 7, 8]. METHCOMP is one more addition to the growing library of high-performance compression suites for *-omics* data that is expected to play a significant role in future cancer genomics and personal medicine research. Given that the inherent characteristics of methylation data are unchangeable (e.g., chromosome name, start and end point of the identified methylation cite, methylation statistics etc), shifts in methylation data acquisition technologies will not impact the utility of the software. The column-by-column compression approach also allows to flexibly incorporate new information columns, or discard unnecessary ones. Furthermore, since the bedMethyl format is a special form of a BED format, it may be easily extended to operate on other BED formats.

References

[1] Song, Q., Decato, B., Hong, E. E., Zhou, M., Fang, F., Qu, J., Smith, A. D. (2013). A reference methylome database and analysis pipeline to facilitate integrative and comparative epigenomics. *PloS one*, **8(12)**, e81148.

[2] C++ Standards Committee., (2011). ISO/IEC 14882:2011, Standard for Programming Language C++. *Technical report, 2011.* **http://www.open-std.org/jtc1/sc22/wg21**.

[3] Malysa, G., Hernaez, M., Ochoa, I., Rao, M., Ganesan, K., Weissman, T. (2015). QVZ: lossy compression of quality values. *Bioinformatics*, **31(19)**, 3122-3129.

[4] Long, R., Hernaez, M., Ochoa, I., Weissman, T. (2017, April). GeneComp, a new reference-based compressor for SAM files. *Data Compression Conference (DCC)*, **(pp. 330-339)**. IEEE.

[5] Tatwawadi, K., Hernaez, M., Ochoa, I., Weissman, T. (2016). GTRAC: fast retrieval from compressed collections of genomic variants. *Bioinformatics*, **32(17)**, i479-i486.

[6] Ravanmehr, V., Kim, M., Wang, Z., Milenkovic, O. (2017). ChIPWig: A Random Access-Enabling Lossless And Lossy Compression Method For ChIP-Seq Data. *bioRxiv*, 127464.

[7] Kim, M., Zhang, X., Ligo, J. G., Farnoud, F., Veeravalli, V. V., Milenkovic, O. (2016). MetaCRAM: an integrated pipeline for metagenomic taxonomy identification and compression. *BMC bioinformatics*, **17(1)**, 94.

[8] Wang, Z., Weissman, T., Milenkovic, O. (2015). smallWig: parallel compression of RNA-seq WIG files. *Bioinformatics*, **32(2)**, 173-180.

[9] Yang, A. S., Estécio, M. R., Doshi, K., Kondo, Y., Tajara, E. H., Issa, J. P. J. (2004). A simple method for estimating global DNA methylation using bisulfite PCR of repetitive DNA elements. *Nucleic acids research*, **32(3)**, e38-e38.

[10] Deorowicz, S., Danek, A., Grabowski, S. (2013). Genome compression: a novel approach for large collections. *Bioinformatics*, **29(20)**, 2572-2578.

[11] Roguski, Ł., Deorowicz, S. (2014). DSRC 2-Industry-oriented compression of FASTQ files. *Bioinformatics*, **30(15)**, 2213-2215.

[12] Witten, I. H., Neal, R. M., Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, **30(6)**, 520-540.