FQSqueezer: k-mer-based compression of sequencing data

Abstract

Modern sequencing instruments create a massive amount of data that must be processed. the advanced data compression of FASTQ files has gotten a lot of attention. Current algorithms, on the other side, are still flawed, and the best tools create very large archives. FQSqueezer is a novel data compression algorithm capable of processing single- and paired-end reads of varying lengths. It is based on the well-known prediction by partial matching and dynamicMarkov coder algorithms from the world of general-purpose compressors. The compression ratios are often tens of percent higher than what state-of-the-art software can provide. Large memory and time requirements are disadvantages of the proposed system.

Introduction

Genome sequencing has progressed to the point that it is now a mature technology with various medical applications. Illumina's instruments generate the vast majority of available data at a low cost. The sequenced reads are short but of high quality. The PacBio or Oxford Nanopore 3rd generation instruments can produce much longer readings, but at a much lower throughput and with much lower efficiency.

In this article, we propose a novel compression algorithm for FASTQ files. The main novelty is in the compression of DNA bases, for quality scores and read identifiers we follow similar strategies. Our algorithm, FQSqueezer, make use of the ideas from the prediction by partial matching (PPM) and dynamic Markov coder (DMC) general-purpose methods. the PPM-like strategy to sequencing reads would be, very hard and likely unsuccessful. There are four main reasons for that. First, PPM algorithm should construct a dictionary of all already seen strings of length up to some threshold. Second, the PPM algorithms often need many accesses to the main memory. Third, sequenced data contain errors that should be corrected. Fourth, the PPM algorithms usually learn slowly.

To address these issues, we created a collection of fixed-k dictionaries for k-mers (k-symbol long substrings) We use a much more aggressive approach to estimating the likelihood of symbols appearing. As a result, compression is greatly improved. Finally, we perform some kind of error correction when storing k-mers in dictionaries.

Proposed ideas are that longer (assembled) fragments are of much better quality than reads, so small differences between very similar fragments are usually due to variations between organisms. In sequencing datasets, we usually work with reads originating from a single genome and differences are in majority due to sequencing errors. A compressor of sequencing data should be designed in a different way than a compressor of assembled genomes or its parts. We greatly developed the previously listed ideas and suggested new approaches in this

article. The most significant are the organizing of large dictionaries and the design of PPM-like probabilistic estimation. We designed the entire FASTQ compressor system.

The main asset of FQSqueezer is its compression ratio. It is a few times slower than the mentioned competitors in compression and much slower in decompression. Our tool has, however, also some

Related Work

The storage and transfer of sequenced drawbacks in terms of speed and memory usage.
reads will consume a lot of money and could be a dominant factor in total costs related to sequencing. Therefore, there is a lot of research was made to overcome this problem.

The first step was application of gzip, a general-purpose compressor. about 3-fold reduction of files was remarkable. The main problem of gzip is that it was designed mainly for textual data, or more precisely, for data with textual-like redundancy types.

The next step was an invention of specialized algorithms taking into account types of redundancies specific for FASTQ files. The details of the proposed algorithms were different, but in general the authors tried (with some exceptions) to compress the reads locally, i.e., not looking for the large-scale relations between the reads.

After that, the authors introduced a variant of the Burrows–Wheeler transform to find overlaps between reads. For human reads with 40-fold coverage they were able to spend about 0.5 bits per base, which was a significant improvement.

In the following years, other researchers explored the concept of using minimizers, short lexicographically smallest, substrings of sequences, to find reads from close regions. The key observation was that if two reads originate from the close regions of a genome their minimizers are usually the same.

The recent experiments suggest that reduction of quality score resolution has very little impact on the quality of variant calling.it was shown that even more aggressive reduction to just two quality values can be justified, at least in some situations. Moreover, there are several algorithms like QVZ and Crumble that perform advanced analysis of quality scores to preserve only the most important information.

Result

Tools and datasets

We employed state-of-the-art competitors, such as FaStore, Spring, and Minicom, for the evaluation. We decided not to test certain other good compressors, such

as BEETL, Orcom, AssembleTrie, and HARC, because earlier research had shown that they performed worse than the tools we chose. As established in previous articles, older utilities are not competitive in terms of compression ratio.

The datasets for experiments were taken from the previous studies. They are characterized in Supplementary Table 1. In the main part of the article, we used 9 datasets but more results can be found in the Supplementary Worksheet. Unfortunately, some compressors do not support all the examined modes, which is a reason of lack of their results in some tables. All experiments were run at workstation equipped with two Intel Xeon E5-2670 v3 CPUs (2 × 12 double-threaded 2.3 GHz cores), 256 GB of RAM, and six 1 TB HDDs in RAID-5. If not stated explicitly the programs were run with 12 threads.

Compression of the bases

The most important part of the present work is the compression of bases. Therefore, in the first experiment we evaluated the tools in this scenario. The ratios are in output bits per input base. As it is easy to observe, in the majority of cases FQSqueezer outperforms the competitors. We investigated this situation a bit closer by checking whether the compression ratio will depend on the initial ordering of the reads. When we shuffled the reads before compression, the compression ratios for SRR1265495 1 and SRR1265496 1 varied dramatically for all of the compressors tested, namely, 0.632 0.531 and 0.646 0.527 (Spring), 0.448 0.539 and 0.484 0.568 (Minicom), 0.506 0.472 and 0.517 0.487 (Minicom) (FQSqueezer). This indicates that the reads' initial ordering in these datasets is far from random, which Minicom may take benefit of. FQSqueezer always wins in the mode that allows reordering of the readings (REO). Also, for all of the approaches tested, the compression ratios are nearly twice as good as in OO mode.

However, the most significant disadvantages of FQSqueezer are its time and space requirements. It is a few times slower than the competition in compression, but the difference is larger in decompression. The reason is obvious. Time is needed for FaStore, Spring, and Minicom to detect overlapping reads that are likely generated from close genomic sites. Despite this, decoding the matches between overlapping reads is a simple. The FQSqueezer algorithm is a PPM algorithm. The algorithms in this family essentially repeat the work done in compression during decompression, therefore the variations in compression and decompression times are insignificant.

One of the parameters of FQSqueezer is the assumed genome size, which should be comparable to the true genome size in the sequencing experiment. The lengths of the k-mers kept in the dictionaries are determined by the genome size. We looked at the effects of this parameter in the last experiment in this section. The findings demonstrate that stating an incorrect genome size lowers the compression ratio, but the differences aren't significant.

Full FASTQ compressors

FQSqueezer is a FASTQ compressor, so we ran it for a few datasets to verify its performance in such scenario. There were only two competitors: FaStore and Spring as Minicom was designed just for bases. The results are for three modes. In the *lossless* mode, all data were preserved. In the *reduced* mode, the IDs were truncated to just the instrument name and the quality values were down-sampled to 8 levels (i.e., Illumina 8-level binning). In the *bases only* mode, only the bases are stored.

The experiment confirmed the advantage of FQSqueezer in terms of compression ratio. Nevertheless, our compressor is slower and needs more memory than the competitors. It is interesting to note that in the lossless modes both Spring and FQSqueezer give smaller archives when they do not permute the reads. This is caused by the large cost of compression of IDs that are not so similar for subsequent reordered reads. In the remaining modes, the cost of ID storage is negligible, so the reordering modes win.

Methods

Basic description of the algorithm

FQSqueezer is a multi-threaded algorithm. It accepts both single-end and paired-end reads. The reads can be stored in the original ordering (OO) or can be reordered (REO). In the reordering mode, the reads are initially sorted according to the DNA sequence (first read of a pair in the PE mode).

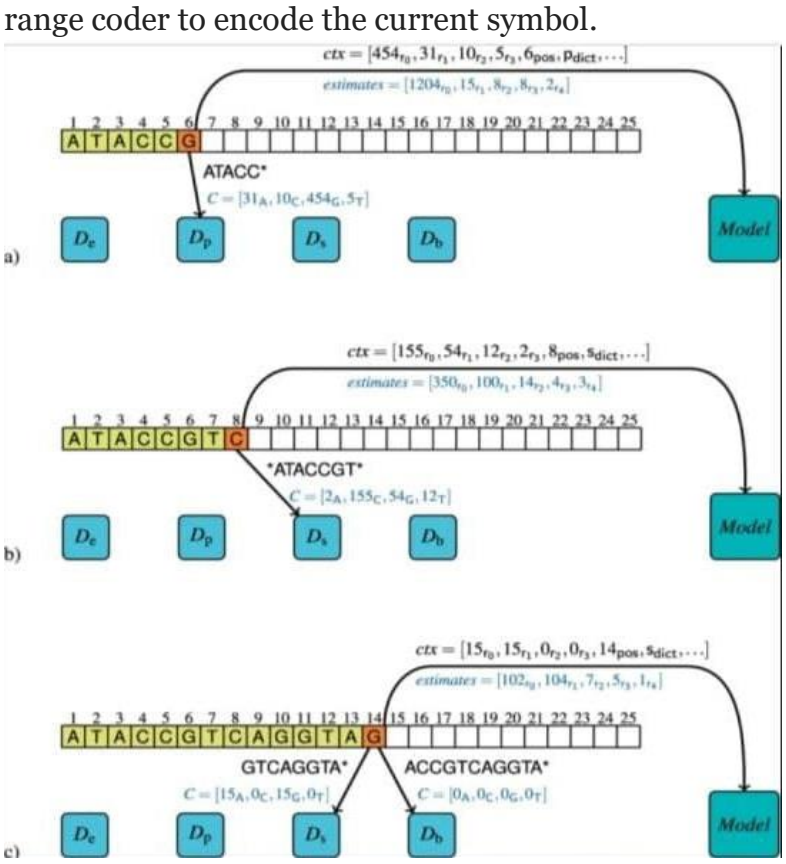
The input FASTQ files are loaded in blocks of size 16 MB. The reads from a single block are compressed one by one (or pair by pair). The read ID and quality values are compressed using rather standard means.

Compression of bases

In the compression of a SE read (or first read of a pair), we process the bases from the beginning of a read position by position. For each base we determine the statistics of occurrences of k-mers ending at this base in the already processed part of the input data. To this end, we maintain a few dictionaries: D_e, D_p, D_s, and D_b that store numbers of occurrences of: e-mers, p-mers, s-mers, and b-mers, respectively, where e<p<s<be<p<s<b.

Let us follow the example given in Fig.1 assuming the sizes of k-mers: $e = 4$, $p = 6$, $s = 9$, $b = 12$. The subfigures (a)–(c) show how the probabilities of appearance of each symbol from the alphabet {A,C,G,T,N} are estimated for some symbols of a read. In Fig.1a, the 6th symbol (orange cell in the figure) is encoded. We consult the D_p dictionary (as there are too few already-processed-read symbols to use D_s or D_b) for statistics of appearance of all p-mers ATACC*, where "*" represents any symbol. The answer (blue font in the figure) is that: ATACC**A** appeared 31 times, ATACC**C**—10 times, ATACC**G**—454 times, ATACC**T**—5 times. Then, we reorder the alphabet symbols according to this statistics (for simplicity let us assume that for equal counters the symbols are ordered lexicographically): G (rank 0), A (rank 1), C (rank 2), T

(rank 3), N (rank 4). In the next step, we check in the *Model* dictionary how many times for such ordered counters, at the 6th position, when the statistics are from the D_p dictionary the current symbol was the one with rank 0, 1,.... The answer is that 1204 times the symbol of rank 0 was encoded, 15 times the symbol of rank 1, and so on. The frequencies 1204, 15, 8, 8, 2 are then used by the range coder to encode the current symbol.



The situation presented in Fig. 1b is a bit different. Here we are at the 8th position in a read so we are able to look for statistics in D_s dictionary. Nevertheless, this dictionary stores statistics for 9-mers and we can construct only 8-mers from the already processed symbols. Therefore, the dictionary is asked for *ATACCGT* 9-mers, where "*" represents any symbol. The answer is that: *ATACCGT**A*** appeared 2 times, *ATACCGT**C***—155 times, *ATACCGT**G***—54 times, *ATACCGT**T***—12 times. Then, we ask the *Model* dictionary and obtain estimates: 350 for rank-0 symbol (C in this situation), 100 for rank-1 symbol (G), 14 for rank-2 symbol (T), 4 for rank-3 symbol (A), 3 for rank-4 symbol (N). These estimates are used for encoding the current symbol using the range coder.

In Fig. 1c, we show the processing of the 14th symbol. Now, the number of processed symbols is sufficient to use D_b dictionary for statistics of occurrence of 12-mers ACCGTCAGGTA*. Unfortunately, the answer is that no such 12-mer has been seen so far. Therefore, we use the D_s dictionary for GTCAGGTA* and obtain statistics: 15 for A, 0 for C, 15 for G, 0 for T. Then we use the *Model* dictionary and see that the estimates for the current symbol are: 102 for rank-0 symbol (A), 104 for rank-1 symbol (G), 7 for rank-2 symbol (C), 5 for rank-3 symbol (T), 1 for rank-4 symbol (N). As previously, these values can be used to encode the current symbol by the range coder.

When a symbol is encoded, it is checked whether it looks like a sequencing error. Let us assume that the statistics from the D_b dictionary are C=[0A,5C,0G,0T]C=[0A,5C,0G,0T] and the current symbol is A. The dominance of C symbols suggests that A is a sequencing error. Therefore, we encode A at this position but replace A by C in the read, which has impact on the processing of the next symbols in the read. To assume that we have a sequencing error the most frequent symbol should have counter at least 3 and the current symbol should have counter 0.

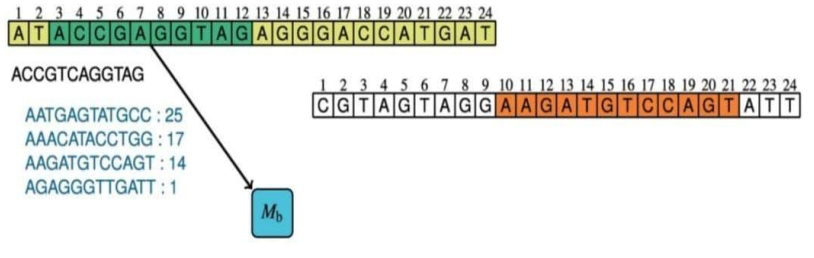
In the REO mode, the prefix of size p of a SE read (or first read of a pair in the PE mode) is encoded in a different way. Roughly speaking, we treat it as a $2p$-bit unsigned integer and encode the difference between it and the integer representing the prefix of the previous read. Using the same read as in the above example, we pick the 6-mer ATACCG and convert it to 12-bit integer using mapping: 00_2 for A, 01_2 for C, 10_2 for G, 11_2 for T. Therefore the read prefix is represented as $0011000101102=790100011000101102=79010$. Let us also assume the previous read prefix was ATACAT, which translates to $0011000100112=787100011000100112=78710$. Therefore. the current read prefix is encoded as a difference $790-787=3790-787=3$. The differences are usually small numbers which can be encoded efficiently using a range coder. The suffix is compressed in the same way as in the OO mode.

In the PE modes, the first read of a pair is compressed exactly in the same way as in the SE mode. The processing of the second read is a bit more complex, but it is the same in the OO and REO modes. Dictionary Mb stores pairs of minimizers of read pairs seen so far. We try to predict some b-mer of the second read. Then, we store the substrings of the read following and preceding this b-mer.

Figure 2 shows an example how the pair of minimizers are used to predict some of the second read b-mer. At the beginning a minimizer, m1=ACCGAGGTAG, in the first read is found (green cells). Then, we look in the Mb dictionary which minimizers in the second read appeared together with m1 in the past. We obtain four candidates ordered by the number of appearances. Then for each of the candidates we check whether it appears in the second read. In our example, we found AAGATGTCCAGT (orange cells

Figure 2

From: FQSqueezer: *k*-mer-based compression of sequencing data



Compression of IDs

In lossless mode, IDs are compressed similarly as in the state-of-the-art compressors, like Spring or FaStore. The ID of each read is tokenized and compared with the tokens of the previous read. If the tokens at corresponding positions contain numerical values, the difference between the integers is calculated and stored.