1 INTRODUCTION

In this article, we present strategies for the lossless compression of the large number of short DNA sequences that comprise the raw data of a typical sequencing experiment.

Much of the early work on the compression of DNA sequences was motivated by the notion that the compressibility of a DNA sequence could serve as a measure of its information content and hence as a tool for sequence analysis. This concept was applied to topics such as feature detection in genomes (Grimbacher and Tahi, 1994; Milosavljevic and Jurka, 1993; Rivals et al., 1998) and alignment-free methods of sequence comparison (Chen et al., 2009)—a comprehensive review of the field up to 2009 is given by Gramaglia et al. (2009). However, Grimbacher and Tahi in 1994 have been echoed by many subsequent authors in citing the exponential growth in the size of nucleotide sequence databases as a reason to be interested in compression for its own sake. The recent and rapid evolution of DNA sequencing technology has given the topic more practical relevance than ever.

The outcome of a sequencing experiment typically comprises a large number of short sequences—often called ‘reads’—plus metadata associated with each read and a ‘quality score’ that estimates the confidence of each base. Tembe et al. (2011) and Deorowicz and Grabowski (2011) both describe methods for compressing the FASTQ file format in which such data are often stored. The metadata is usually highly redundant, whereas the quality scores can be hard to compress, and these two factors combine to make it hard to estimate the degree of compression achieved for the sequences themselves. However, both schemes employ judicious combinations of standard text compression methods such as Huffman and Lempel-Ziv, with which it is hard to improve substantially upon the naive method of using a different 2-bit code for each of the four nucleotide bases. For example, GenCompress (Chen et al., 2009) obtains 1.92 bits per base (henceforth bpb) compression on the Escherichia coli genome.

An experimenter wishing to sequence a diploid genome such as a human might aim for 20-fold average coverage or more, with the intention of ensuring a high probability of capturing both alleles of any heterozygous variation. This oversampling creates an opportunity for compression that is additional to any redundancy inherent in the sample being sequenced. However, in a whole-genome shotgun experiment, the multiple copies of each locus are randomly dispersed among the many millions of reads in the dataset, making this redundancy inaccessible to any compression method that relies on comparison with a small buffer of recently seen data. This can be addressed by ‘reference-based compression’ (Fritz et al., 2011; Geranios et al., 2011), which saves space by sorting aligned reads by the position they align to on a reference sequence and expressing their sequences as compact encodings of the differences between the reads and the reference. However, this is fundamentally a lossy strategy that achieves best compression by retaining only reads that closely match the reference, limiting the scope for future reanalyses such as realignment to a refined reference
practical to compute the BWT of large collections of DNA sequences (containing, perhaps, ethnicity-specific haplotypes [Dewey et al. 2011]) or any sort of de novo discovery on reads that did not align well initially. Moreover, as Yano[vs]ky 2011 points out, a reference-based approach is inappropriate for experiments for which a reference sequence is not clearly defined (metagenomics) or entirely absent (de novo). Yano[vs]ky 2011 describes a lossless compression method ReCoil for sets of reads that works in external memory (i.e. via sequential access to files held on disk) and is therefore not constrained in scale by available RAM. A graph of similarities between the reads is first constructed and then each read is expressed as a traversal on that graph, encodings of these traversals then being passed to a general-purpose compressor [ReCoil uses 7-Zip (http://www.7-zip.org, Igor Pavlov)].

The two-stage nature of this procedure is shared by the family of compression algorithms based on the Burrows–Wheeler transform (BWT). The BWT is simply a permutation of the letters of the text and so is not a compression method per se. Its usefulness for compression stems from the facts that the BWT tends to be more compressible than its originating text (since it tends to group symbols into ‘runs’ of like letters, which are easy to compress) and, remarkably, that the originating text can be reconstructed from it (thus allowing the BWT to represent the originating text without loss of information). Once generated, the BWT is compressed by standard techniques: a typical scheme would follow an initial move-to-front encoding with run length encoding and then Huffman encoding.

The widely used BWT-based compressor bzip2 (http://www.bzip.org, Julian Seward) divides a text into blocks of (at most, and by default) 900 kB and compresses each separately, hence is only able to take advantage of local similarities in the data. Mantaci et al. 2005 gave the first extension of the BWT to a collection of sequences and used it as a preprocessing step for the simultaneous compression of the sequences of the collection. They show that this method is more effective than the technique used by a classic BWT-based compressor, because one could potentially access redundancy arising from these long-range correlations in the data. Until recently, computing the BWT of a large collection of sequences was prevented from being feasible on very large scales by the need either to store the suffix array of the set of reads in RAM (requiring 400 GB of RAM for a dataset of 1 billion 100mer reads) or to resort to ‘divide-and-conquer then merge’ strategies at considerable cost in CPU time. However, in Bauer et al. 2011, three of the present authors described fast and RAM-efficient methods capable of computing the BWT of sequence collections of the size encountered in human whole genome sequencing experiments, computing the BWT of collections as large as 1 billion 100mers.

Unlike the transformation in Mantaci et al. 2005, the algorithms in Bauer et al. 2011 required ordered and distinct ‘end-marker’ characters to be appended to the sequences in the collection, making the collection of sequences an ordered multiset, i.e. the order of the sequences in the collection is determined by the lexicographical order of these end-markers. It is easy to see that the ordering of sequences in the collection can affect the compression, since the same or similar sequences might be distant in the collection. We will outline different ordering strategies for the input sequences and will show their effect on simulated and real data.

We have created an open-source C++ library BEETL that makes it practical to compute the BWT of large collections of DNA sequences and thus enables the redundancy present in large-scale genomic sequence datasets to be fully exploited by generic second-stage compressors such as bzip2 and 7-Zip. We use simulated read collections of up to 60× redundancy to investigate the effect of genome coverage, sequencing error and read length on the level of compression. Furthermore, we show the effect of read trimming and choices of second-stage compressors on the level of compression. Finally, we describe an extension of our method that implicitly records sequences in the collection so as to drastically improve compression.

2 METHODS

Consider a string \( s \) comprising \( k \) symbols from some finite ordered alphabet \( \Sigma = \{e_1, e_2, \ldots, e_\sigma\} \) of size \( \sigma \). We mark the end of \( s \) by appending an additional symbol \( S \) that we consider to be lexicographically smaller than any symbol in \( \Sigma \). We can build \( k + 1 \) distinct suffixes from \( s \) \( \text{by starting at different symbols of the string and continuing rightwards until we reach } S \). If we imagine placing these suffixes in alphabetical order, then the BWT of \( s \) (Burrows and Wheeler 1994) can be defined such that the \( i \)-th element of the BWT is the symbol \( e_i \) that precedes the first symbol of the \( i \)-th member of this ordered list of suffixes. Each symbol in the BWT therefore has an associated suffix in the string. We recommend [Berend et al. 2008] for further reading.

One way to generalize the notion of the BWT to a collection of \( m \) strings \( S = \{s_1, s_2, \ldots, s_m\} \) is to imagine that each member \( s_i \) of the collection is terminated by a distinct end-marker \( S_i \), such that \( S_1 < \cdots < S_m \). Moreover, we assume that the end-markers are lexicographically smaller than any symbol in \( \Sigma \). In Bauer et al. 2011, we give two related methods for computing the BWT of large collections of DNA sequences by making use of sequential reading and writing of files from disk. The first variant \( \text{BCR} \) requires 14 bytes of RAM for each sequence in the collection to be processed (and is hence capable of processing over a billion reads in 16 GB of RAM), whereas the second variant \( \text{ReCoil} \) uses negligible RAM at the expense of a larger amount of disk I/O.

To understand how a BWT string might be compressed, we can think of it as the concatenation of a set of ‘runs’ of like letters, each of which can be described by its constituent symbol \( e_i \) plus an integer \( l_i \) denoting the number of times \( e_i \) is repeated. We assume all runs are maximal, i.e. they do not abut another run of the same character. Intuitively, for two strings of the same length, we expect the string that consists of fewer (and hence, on average, longer) runs to compress to a smaller size.

Given \( S \), our strategy is to search for permutations \( S \rightarrow S' \) of the sequences in the collection such that the BWT of the permuted collection \( S' \) can be more readily compressed than the BWT of \( S \). For the BWT of \( S \), we define a bit array called the SAP-array (‘same-as-previous’) whose elements are set to 1 if and only if the suffixes associated with their corresponding characters in the BWT are identical (their end-markers excluded) to those associated with the characters that precede them. Thus each \( 0 \) value in the SAP-array denotes the start of a new SAP-interval in the BWT, within which all characters share an identical associated suffix.

The BWTs of \( S \) and \( S' \) have identical SAP-arrays and can only differ within SAP-intervals that contain more than one distinct symbol. Within such an SAP-interval, the ordering of the characters is entirely determined by the ordering of the reads they are contained in, so best compression is achieved if we permute the characters so they are grouped into as few runs as possible. In doing this we are implicitly permuting the sequences in the collection. As illustrated in Figure 8 if the collection is sorted such that the reverses of the sequences are in lexicographic order (we call this reverse lexicographic order, or RLO for short), then the symbols in an SAP-interval of length \( l \) must group into at most \( \phi(l) \) runs, where \( \phi(l) = \sigma \) is the number of distinct characters encountered within the interval. This compares with an upper bound up to \( l \) runs if no sorting is applied. We would therefore expect the RLO-sorting of a collection to compress better.
that the original and our experiments in the next section show this to be the case in practice.

However, we wish our approach to scale to many millions of reads and so we would prefer to avoid sorting the collection as a preprocessing step. If we know the SAP-array of a BWT, it is a simple single-pass procedure to read both the BWT and its SAP-array from disk in tandem, identify SAP-intervals, sort the characters within them and output a modified BWT. We note there is scope to experiment with different heuristics—e.g. in Figure 1 placing the T prior to the two Cs in the ACC-count would eliminate another run by extending the run of Ts begun in the preceding ACC-count interval.

It remains to compute the SAP-array. To do this, we show that the method of BWT construction introduced by three of the present authors [Bauer et al. 2012] can be modified to compute the SAP-array alongside the BWT with minimal additional overhead. We do not attempt to describe this previous work in full detail, but the BWT construction algorithm proceeds in $k$ stages, $k$ being the length of the reads in the collection (our implementation and, for simplicity, our description here assume all reads in $R$ have the same length, but this restriction is not in fact intrinsic to the algorithm). At stage $j$, the $j$-suffixes ($0 \leq j \leq k$) of the reads (that is, the suffixes of length $j$), the 0-suffix being defined as the suffix that contains only the end-marker $\sigma$) are processed in lexicographic order and the characters that precede them are merged into a ‘partial BWT’.

The partial BWT at step $j$ can be thought of as the concatenation of $j + 1$ segments $B(0), B(1), \ldots, B(\sigma)$, where each segment $B(i)$ corresponds to the symbols in the partial BWT that precede all suffixes of $S$ that are of length smaller or equal to $i$ (in our implementation, we assume that all end-markers are equal, and $e_0 \triangleq e_k$, if $r = 1$, for $h = 0$, and with $c_h \in \Sigma$, for $h = 1, \ldots, \sigma$).

At each step $j$, with $0 \leq j \leq k$, the main idea is to compute the positions of all characters that precede the $j$-suffixes in the old partial BWT to obtain an updated partial BWT that also contains the symbols associated with the suffixes of length $j$. The two variants BCR and BCRec of the Bauer et al. [2012] algorithm converge in slightly different ways (with different trade-offs between RAM use and I/O) to arrange matters so that the symbols we have to insert into the partial BWT are processed by lexicographic order of their associated $j$-suffixes, which allows each segment of the partial BWT to be held on disk and accessed sequentially. Crucially, this also means that the characters in an SAP-interval will be processed consecutively. When we write a BWT character, we can also write its SAP status at the same time, since it is not going to change in subsequent iterations.

We consider two suffixes $u$ and $v$ of length $j$ of two different reads $s_i$ and $s_t$, satisfying $1 \leq i, t \leq m$ and $r \neq v$. Both $s_i$ and $s_t$ belong to $S$ and assume that $u$ is lexicographically less than $v$. Then, they are identical (up to the end-markers) and so the SAP status of $v$ must be set to 1 of and only if both the following conditions hold: the symbols associated with suffixes $u'$ and $v'$ of length $(j - 1)$ of $s_i$ and $s_t$ are equal (which guarantees that $u$ and $v$ begin with the same symbol) and they are stored in the same BWT segment (implying that $u'$ and $v'$ begin with the same symbol). The SAP status of all suffixes between $u'$ and $v'$ must be set to 1 which ensures that all suffixes between $u'$ and $v'$ in iteration $j - 1$ coincide and have the length $j - 1$. Note that the symbols in the BWT segment associated with suffixes between $u'$ and $v'$ could be different from the symbol preceding $u'$ (and $v'$).

Actually, we compute the SAP status of the suffixes $u$ and $v$ at step $j - 1$ when we insert the corresponding suffixes $u'$ and $v'$. In particular, when we copy the values from the old to the new partial BWT and insert the new $m$-values, we can read the SAP values and verify the conditions stated above at the same time. It means that at the end of the iteration $j - 1$, we obtain the partial BWT and the SAP values of the next suffixes to insert and, at iteration $j$, we can directly set the SAP status of the $j$-suffixes and compute the SAP status for the suffixes of the next iteration.

We can compute the SAP-interval by induction, i.e. we can compute the SAP-status of the $j$-suffixes at iteration $j - 1$ by using the symbols associated with the $(j - 1)$-suffixes and their SAP values. At iteration $j - 1$, we assume that we know the SAP values of the $(j - 1)$-suffixes and we have to compute the SAP values of the $j$-suffixes for the next iteration. For simplicity, we focus on the computation of the SAP values of a fixed BWT segment, i.e. we only consider the insertion of the $(j - 1)$-suffixes starting with the same symbol. In our implementation, we use a counter $A$ for each symbol of the alphabet and a generic counter $Z$.

The element $A[h]$, for each $1 \leq h \leq \sigma$ and $c_h \in \Sigma$ (we can ignore the end-marker because the reads have the same length and hence it does not appear in the partial BWT), contains the number of SAP-intervals between the first position and the position of the last inserted symbol associated with a read $e_h$ (for some $1 \leq h \leq m$) equal to $c_h$ in the considered BWT segment. The counter $Z$ contains the number of the SAP-intervals between the first position and the position where we have to insert $e_h$. The symbol $c_h$ is associated with the new suffix of length $j - 1$ of read $s_h$ with $1 \leq h \leq m$, that we want to insert. If the value $A[h]$ is equal to $Z$, then the SAP status of the $j$-suffix of $s_h$ (obtained by concatenating $e_j(0)$-suffix of $s_h$) must be set to 1, otherwise it is set to 0. We observe that if $A[h] = Z$ holds true, then this implies that $j$-suffixes of $s_i$ and $s_t$ are in the same SAP-interval.

3 RESULTS

Reads simulated from the E.coli genome (K12 strain) allowed us to assess separately the effects of coverage, read length and sequencing...
error on the level of compression achieved. First, a 60× coverage of error-free 100 base read datasets was subsampled into datasets as small as 10×. We compared different compression schemes, both on the raw input sequences and the BWT. RLO denotes a reverse lexicographical ordering of the reads. SAP is the dataset where all the reads are ordered according to the same-as-previous array. The x-axis gives the coverage level whereas the y-axis shows the number of bits used per input symbol. Gzip, Bzip2, PPMd (default) and PPMd (large) show compression achieved on the raw sequence data. BWT, BWT-SAP and BWT-RLO give compression results on the BWT using PPMd (default) as second-stage compressor.

Table 1. Different combinations of first-stage (BWT, SAP-permuted BWT) and second-stage (bzip2 with default parameters, PPMd mode of 7-Zip with default parameters, PPMd mode of 7-Zip with -m 16 -mmem 2048m, deflate mode of 7-Zip with -m 9) compression compared on 192 million human reads previously analyzed by Yanovsky 2011. Time is in CPU seconds, as measured on a single core of an Intel Xeon X5450 (Quad-core) 3 GHz processor. Compression is in bits per base.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stage 1</td>
<td>Stage 2</td>
</tr>
<tr>
<td>** Reads **</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>905</td>
<td>2.25</td>
</tr>
<tr>
<td>bzip2</td>
<td>324</td>
<td>1.98</td>
</tr>
<tr>
<td>PPMd (default)</td>
<td>5155</td>
<td>2.00</td>
</tr>
<tr>
<td>PPMd (large)</td>
<td>17974</td>
<td>2.00</td>
</tr>
<tr>
<td>-mx9</td>
<td>17974</td>
<td>2.00</td>
</tr>
<tr>
<td>** BWT **</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>818</td>
<td>2.09</td>
</tr>
<tr>
<td>bzip2</td>
<td>353</td>
<td>1.93</td>
</tr>
<tr>
<td>PPMd (default)</td>
<td>4953</td>
<td>2.05</td>
</tr>
<tr>
<td>PPMd (large)</td>
<td>16709</td>
<td>2.09</td>
</tr>
<tr>
<td>-mx9</td>
<td>16709</td>
<td>2.09</td>
</tr>
<tr>
<td>** BWT-SAP **</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>601</td>
<td>1.40</td>
</tr>
<tr>
<td>bzip2</td>
<td>347</td>
<td>1.21</td>
</tr>
<tr>
<td>PPMd (default)</td>
<td>3116</td>
<td>1.28</td>
</tr>
<tr>
<td>PPMd (large)</td>
<td>11204</td>
<td>1.34</td>
</tr>
<tr>
<td>-mx9</td>
<td>11204</td>
<td>1.34</td>
</tr>
</tbody>
</table>

Fig. 2. We simulated 60× coverage of error-free 100 base reads from the E.coli genome and subsampled this into datasets as small as 10×. We compared different compression schemes, both on the raw input sequences and the BWT. RLO denotes a reverse lexicographical ordering of the reads. SAP is the dataset where all the reads are ordered according to the same-as-previous array. The x-axis gives the coverage level whereas the y-axis shows the number of bits used per input symbol. Gzip, Bzip2, PPMd (default) and PPMd (large) show compression achieved on the raw sequence data. BWT, BWT-SAP and BWT-RLO give compression results on the BWT using PPMd (default) as second-stage compressor.

4 DISCUSSION

Analyzing the SRX001540 dataset allowed us to compare our results with ReCoil, but the reads are noisier and shorter than more recent datasets and, at under 3×, the oversampling of the genome is too
Table 2. Different combinations of first-stage (BWT, SAP- and RLO-permuted BWT) and PPMd as the second-stage compression method compared on a 45 x human dataset. Values state the compression achieved in bits per input base. We trimmed the reads by following the strategy described for Isoa which removed 1.3% of the bases. This leads to an improvement in terms of compression ratio of 4.5%

<table>
<thead>
<tr>
<th>Input size</th>
<th>BWT</th>
<th>BWT-RLO</th>
<th>BWT-SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>untrimmed</td>
<td>1.35 Gb</td>
<td>0.746</td>
<td>0.526</td>
</tr>
<tr>
<td>trimmed</td>
<td>1.33 Gb</td>
<td>0.721</td>
<td>0.506</td>
</tr>
</tbody>
</table>

In both cases, reordering the strings in the collection—implicitly or explicitly—is necessary for best compression. One situation where this might be problematic would be if the reads are paired—typically, sequenced from different ends of a library of DNA templates. An obvious way to capture this relationship would be to store the two reads in a pair at adjacent positions in \( S \)—i.e. the first two reads in the collection are reads 1 and 2 of the first read-pair, and so forth. If we wanted to retain this information after a change in ordering, we would incur the overhead of storing an additional pointer to associate each read with its mate.

In [4], we give efficient algorithms for inverting the BWT and recovering the original reads. However, when augmented with relatively small additional data structures, the compression ratio remains the same as for inverting the FM-index [10].

Most sequencing technologies associate a quality score with each sequenced base and [12] found that significant lossless compression of these scores can be hard to achieve, but also showed that a lossy reduction in resolution of the scoring scheme could obtain better compression while having limited impact on the accuracy of variant calls derived from the sequences. Future work we are pursuing would complement such approaches by using queries to a BWT-based index for de novo identification of bases that are not likely to be important in downstream variant calls, whose quality scores can then be discarded entirely.

More broadly, many computationally intensive tasks in sequence analysis might be facilitated by having a set of reads in indexed form. For example, [20] show how the overlap of a set of reads can be computed from its FM-index and apply this insight to build a practical tool for de novo assembly. By making it feasible to index sets of reads on the scale necessary for whole human genome sequencing, our work raises the intriguing possibility of a new generation of algorithms that operate directly on sets of reads held in compressed index form.

ACKNOWLEDGEMENT

The authors thank Dick Evers for his support throughout this project.

REFERENCES


