Practical Compression for Multi-Alignment Genomic Files

Rodrigo Cánovas Alistair Moffat

NICTA Victoria Research Laboratory
Department of Computing and Information Systems
The University of Melbourne
Victoria 3010

Abstract

Genomic sequence data is being generated in massive quantities, and must be stored in compressed form. Here we examine the combined challenge of storing such data compactly, yet providing bioinformatics researchers with the ability to extract particular regions of interest without needing to fully decompress multi-gigabyte data collections. We focus on data produced in SAM format, which is particularly voluminous in nature, and describe storage techniques that have the desired blend of attributes.

Keywords: Genomic data, lossless compression, lossy compression, SAM format.

1 Introduction

Next generation sequencing machines produce vast amounts of genomic data (Ansorge, 2009). This data is valuable for the insights it allows now into the health of individuals and whole populations, and will continue to be of benefit into the future as medical knowledge grows. But for genomic data to be long-term useful, it must be stored. And with output files in the gigabyte range now being generated within an hour or less of technician time, and at a cost of just a few hundred dollars, the mechanics of storing them – and retrieving information out of them when it is required – is a challenge. Bioinformatics researchers are increasingly regarding big data storage facilities as being fundamentally necessary to their operations.

In this paper we consider data stored in SAM (*S*equence *A*lignment *M*ap) format files (Li et al., 2009). These files can contain millions of *reads*, each produced as a continuous fragment of data extracted from the processing of a single genome, represented as a string of *bases*, letters that indicate the fundamental molecules of DNA. A number of meta-data fields are associated with each read to form an *alignment read*, and some of these fields are as expensive to store as the sequence of bases. Because of the multiplicity of alignment reads extracted from each genome, the repetition in the meta-data elements, and the fact that they are stored as printable ASCII text, there is considerable redundancy in SAM files. Our purpose in this project is to identify and exploit that redundancy, and develop a new compressed representation for SAM-style genomic data that is both economical of space and readily queryable, so that data about particular alignments can be extracted in isolation, without requiring whole files to be decompressed. The latter option is of considerable benefit to researchers working with SAM data, who rarely wish to fully decompress archived data – and indeed, may not have the time or space resources required to do so.

The next section provides a general overview of compression methodologies, including a mode we refer to as being *information preserving* that sits between the conventional lossless and lossy approaches. Section 3 then introduces the SAM format that is used to store multi-alignment genomic data. Compression mechanisms suitable for the various SAM fields are examined in Section 4, including measurement of their effectiveness on several typical files. The issue of querying SAM files is then examined in Section 5. Section 6 presents related work, and then Section 7 concludes our presentations.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

2 Compression technologies

This section summarizes several issues relevant to the design of compression techniques. For detailed coverage of these topics, see, for example, Bell et al. (1990), Moffat and Turpin (2002) and Navarro and Mäkinen (2007).

Lossless and lossy compression

Compression techniques can be categorized as belonging to one of two distinct classes: *lossless*, or exact compression; and *lossy* compression. Lossy compression methods are typically applied to data originally sampled from continuous domains, and are based on the recognition that the process of turning that data into digital form can, within limits, be further approximated to save space. For example, digital cameras take images that can be stored in either `.jpg` form (lossy compression) or as `.raw` files (larger uncompressed files). But even the `.raw` file is a quantized approximation of the original scene, and its attractiveness to photography purists is not that it contains *no* loss of fidelity, but only that it contains no *additional* loss of fidelity. In most applications and environments a viewer will not perceive any difference between the two formats.

On the other hand, data which is fundamentally discrete and non-continuous, such as ASCII text, is almost always represented using lossless methods (although it is also worth noting that from time to time the observation has been made that human readers can still make sense of some lossy representations of text).

Information-preserving approaches

There is a second way in which lossy compression concepts might be useful for some data sources, which we

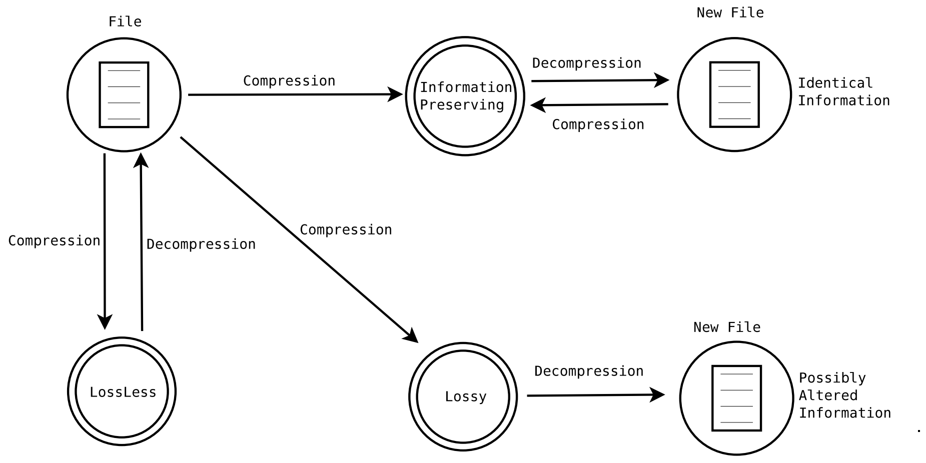


Figure 1: Three alternative compression modalities.

introduce by way of an example. Suppose a list of n numbers each in the range $0 \ldots (m-1)$ is generated by some process, in no particular order. These numbers might be stored in ASCII in a file of at most $n(\lceil\log_{10} m)\rceil+1)$ bytes (allowing $+1$ for a newline character after each value), or, if a minimal binary representation is used, in a file of $n\lceil\log_2 m\rceil$ bits. If the purpose to which the data will be put is unknown, and no further indication is provided in regard to the distribution of the numbers within the specified range, then the binary form is an efficient one. But suppose the downstream application that uses the set of numbers places no importance on the order in which the numbers are received. If so, then the set of numbers can be sorted, differences between those numbers taken, and the differences coded using $n(2+\lceil\log_2((n+m)/n)\rceil)$ bits (Moffat and Turpin, 2002), which might be a non-trivial saving. That is, if we regard the *order* in which the data is presented as being of no importance, then shifting the input into a particular arrangement might allow better compression. In such cases, the decompressed output will also be in that order, and so the original input file cannot be exactly regenerated, in the sense that the Unix `diff` and `cmp` commands will certainly report a discrepancy. Nevertheless, there might also be a sense in which all of the actual information embedded in the source data has been preserved, even if the physical representation has not. We call such representations *information preserving*, and regard them as being a third possible compression modality, neither lossless nor lossy.

Figure 1 illustrates these notions. A lossless compression mechanism must exactly recreate the input file, in all syntactic detail, as indicated by the double arrow. An information preserving regime will not be able to reproduce the original file (denoted by a single arrow), but once it has been decompressed a first time into its new form, it can be recompressed and decompressed a second time without further change taking place (the double arrow at the top of the diagram). That is, a representational fixed-point is reached after one compress/decompress cycle. A lossy scheme cannot reproduce the input file; nor is there even any guarantee that a second iteration of compression and decompression will achieve the same file.

Information preserving techniques have also been proposed in other application areas. For example, consider the area of program source code compression. In a syntax-aware compressor for a language such as C or Pascal, white-space tokens can be normalized and other changes made that do not alter the compilability or correctness of the program, and only affect how it looks in an editor.

It may also possible to augment an information preserving compression regime with additional information so as to adjust its output to recreate the exact input. That is, losslessness might be an optional enhancement, at the cost of storing further data. In the case of program source code compression, the auxiliary information would specify the exact whitespace token to be inserted at each location from which it had been stripped. In the case of the numeric example used at the beginning of this discussion, a permutation index costing $n\lceil\log_2 n\rceil$ bits would be required, which is a relatively high cost. In the SAM-format compression proposed in the next section, the items being permuted are long lines of text, and the overhead cost of storing the required permutation vector is small.

Modeling and coding

It is also recognized that compression should be thought of as consisting of two complementary activities. *Modeling* is the process of inferring structure from the data that is presented, and, for each type of symbol or type of context, estimating (either explicitly or implicitly) a probability distribution that covers the set of options that might occur next. Those probabilities are then – again, either explicitly or implicitly – used to drive a *coding* step, in which the actual symbol that appears is represented into the output bitstream, taking into account the probability estimates generated by the model. One early example of such a structure is given the blend of model and coder sometimes referred to as *Huffman coding*, in which symbol probabilities are estimated using a zero-order Markov model counting their occurrence frequencies in the text in question; and then the symbols in the sequence are coded via a bit-aligned minimum-redundancy code.

Static, semi-static, and adaptive

A third characterization of compression techniques is whether they make use of *static*, *semi-static*, or *adaptive* probability estimations (and hence coders). In a static regime, probability estimates are independent of any particular input file, and are constant. In the semi-static approach, the probability estimates are based on the data file being represented, and are established in a preliminary pass through the data and then sent as the first component of the compressed message. In an adaptive system the probability estimates are constructed on-the-fly, based on the part of the message that has already been processed, and if necessary, allowing for the possibility of previously unencountered symbols to be added as they arise.

Static and semi-static codes tend to allow faster decoding, because there is no need for the decoder to track the

Field	Type	Description
QNAME	string	Query template name
FLAG	int	Bitwise flags (values between 1 and 1160) that give properties of the alignment, including if the sequence is a reverse complement
RNAME	string	Reference sequence name
POS	int	Leftmost mapping position of the first matching base
MAPQ	int	Mapping quality: $-10\log_{10} Pr(\text{mapping position is wrong})$
CIGAR	string	The CIGAR string
RNEXT	string	Reference sequence name of the next segment in the template
PNEXT	int	Position of the next segment in the template
TLEN	int	Signed observed template length
SEQ	string	Sequence of nucleotides bases of the read used in the alignment
QUAL	string	Estimated error probability of each base: $-10\log_{10} Pr(\text{base is wrong}) + 33$
OTHER	string	Optional fields of the form TAG:TYPE:VALUE

Table 1: The twelve fields recorded for each read in a SAM file. The first eleven are required, but may be replaced with * for strings and 0 for numeric fields if data is not known or is not being stored. Further details are provided by Li et al. (2009).

Letter	Value	Probability
(	40	20%
7	55	0.6%
F	70	0.02%
U	85	0.0006%
d	100	0.00002%

Table 2: Examples of values stored in the QUAL field. The ASCII letters represent probabilities of error in the corresponding base according the relationship *value* $= -10\log_{10} Pr(\text{error}) + 33$. The error probabilities are computed by the sequencing hardware.

probability changes. Static and semi-static methods also make it easier to provide random-access into the compressed file. In particular, if a bit pointer is provided into the compressed package, coding can be resumed from that location provided the context is understood.

3 Genomic data formats

Genomes are typically described by (usually long) sequences of identifying letters, one per base-pair of the original. In simplest form, the letters are the four acronyms of the fundamental bases, A, C, G, and T. Some formats (including SAM) add other letters, such as N, for unknown bases; and some formats further extend the alphabet to include specific identifying letters for other proteins that might be present. The common thread in all formats is the small alphabet that is employed (between four and around twenty symbols), and the dominance of the four key symbols.

SAM format

In the SAM format, each sequence of bases is accompanied by eleven other fields that add considerably to the total stored size (Li et al., 2009). These fields are shown in Table 1. All of them are required, in the sense that they cannot be omitted; but it is also common for them to be stored as place-holder 0 and/or * values.

```
                                 1
              1 2 3 4 5 6 7 8    9 0 1 2 3 4 5 6
Sequence 1:   T T A G A T A A * * G A T A G C T G

Sequence 2:   T T A G A T A A A G G A T A * C T G

     CIGAR:   8M 2I 4M 1D 3M
```

Figure 3: Example of CIGAR analysis. The positions marked with * are indicative only, and not present in either of the two sequences.

The field labeled SEQ is the sequence of bases corresponding to this read; the other critical field from a data storage point of view is QUAL, which is the same length as the SEQ field, and also contains an ASCII letter for each sequence position. The value stored in each QUAL field is an estimate of the correctness of the corresponding SEQ field. The mathematical relationship between estimated probability of error and value stored in QUAL is shown in Table 1; and Table 2 provides some examples. The QUAL sequence can be thought of as a quantization of a underlying phenomena that is numeric and continuous, and hence a candidate for possible use of lossy compression. Figure 2 shows a sample read containing 25 bases in the SEQ string, with the accompanying QUAL string indicating that each base after the first has an error probability of well under 0.05%.

Each of the reads may be referenced against an external resource described by the RNAME field, which can be thought of as a reference identifier indicating the external location of a related resource. If an alignment has been computed for this read relative to that resource, then the POS field indicates the offset within the resource at which the alignment commences.

Overall, a SAM file consists of a header block describing attributes of the sample as a whole, such as meta-data describing the experimental environment and regime; followed by thousands or millions of relatively short reads – each perhaps 30 to 120 bases long – derived from a single experimental run. Hence, it is not unusual for the same RNAME to turn up many times in the SAM file, and nor is it in any way unusual for the reads to overlap, in the sense of the identified alignment for one of them being within the range of the identified alignment of another.

In some cases, the read represented by the SEQ string has not only been aligned against the RNAME string, it is also represented relative to it as a sequence of edit instructions. If so, the corresponding CIGAR field (*C*ompact *I*diosyncratic *G*apped *A*lignment *R*eport) is non-empty. If it is present, the CIGAR string consists of a sequence of instructions: M atch the next ℓ characters; D elete the next ℓ characters; or I nsert a group of ℓ character. Figure 3 gives an example showing two similar reads, and a CIGAR string that describes their (relative to each other) structure.

Fields that are absent are represented by * and/or 0 characters. The file itself is tab-delimited between fields, and newline delimited between read alignments.

BAM format

The SAMtools software suite[1] provides other storage options. A second standard representation is known as BAM format, in which blocks of text from a SAM file are stored compressed using a modified zlib library. Compared to the original SAM file, a BAM compressed version can be expected to occupy around 30% of the original size, with an auxiliary index that allows limited random access to the reads in order to support queries. The BAM representation uses the BGZF (Blocked GNU Zip Format) compression

[1]See http://samtools.sourceforge.net/.

```
SEQ   :    C  T  G  A  A  C  T  T  A  G  G  C  T  C  A  G  C  C  T  C  A  G  T  A  A

QUAL  :    <  F  E  F  G  E  I  G  G  H  H  H  J  H  I  I  I  I  J  H  J  I  G  H  H

Value :   60 70 69 70 71 69 73 71 71 72 72 72 74 72 73 73 73 73 74 72 74 73 71 72 72
```

Figure 2: Example of the SEQ and QUAL components for one read within a SAM file.

	HG00113	HG00559	Local
Alignments	5,630,211	2,515,117	8,095,516
Av. read (bases)	90	108	100
Size (MB)	2,220.5	1,121.6	1,965.7
BAM (MB)	457.8	285.1	762.9
Gzip (MB)	429.2	260.7	748.2

Table 3: Statistics for three SAM files. File HG00113 refers to HG00113.chrom11.ILLUMINA.bwa.GBR.exome.20111114; File HG00559 is HG00559.chrom20.ILLUMINA.bwa.CHS.lowcoverage.20111114; and file Local was supplied by a local researcher based on their own bioinformatics work.

Field	Raw		Gzip	
	MB	%	MB	%
QUAL	488.6	22.0%	239.6	66.5%
SEQ	488.6	22.0%	38.2	10.6%
OTHER	933.5	42.0%	27.0	7.5%
QNAME	100.9	4.5%	20.1	5.6%
PNEXT	49.0	2.2%	12.2	3.4%
TLEN	24.0	1.1%	8.7	2.4%
POS	49.0	2.2%	8.2	2.3%
Total	2220.5	100.0%	360.4	100.0%

Table 4: Percentage required by various SAM fields of HG00113, before and after compression using gzip in a striped manner, ordered by decreasing compressed contribution to the total, and with six smaller fields omitted. The striped and compressed representation totals 360.4 MB.

format, which adds an access structure on top of the standard gzip file format.

As is shown in Table 3, BAM conversion results in a stored file that is a little larger than can be attained by gzip alone. The difference is largely caused by the additional BAM index, which links positions in the reference sequence with reads in the blocks of the BAM file.

4 Compressing SAM components

To evaluate alternative compression methodologies, three SAM files are used. Some attributes of the three sample files are summarized in Table 3. In the remainder of the text they are referred to by their abbreviated names HG00113, HG00559, and Local.

Striping

One well-known mechanism for compressing data stored in structured formats like SAM is to *stripe* the data into separate streams, and then use a general-purpose compressor – such as gzip – on the concatenated contents of each stream. Decompression regenerates the various streams, and they can be re-interleaved to recreate the original file. Striping is effective if the fields are distinctive in nature, and/or contain vertical repetitions. Table 4 shows the raw cost of some of the striped components of the test file HG00113, as a percentage of the file size, before and after the components are compressed by gzip.

Notable in the table is that the QUAL and SEQ fields are equal in size prior to application of a standard compression regime, but that the SEQ field is far more compressible than the QUAL field, and that the latter dominates the compressed representation. The high relative compressibility of the SEQ field is a consequence of the fact that it is over a very small alphabet; and that overlapping reads are likely to contain common subsequences that can be identified by the string match-based gzip compression mechanism. The OTHER field is also highly compressible, and moves from being the dominant cost in the uncompressed version of the file, to being less than 8% of the striped compressed file. Note that the percentages in Table 4 are relative to the sizes of the two original files. Overall compression rates for particular components can be estimated from the figures supplied. For example, on file HG0013 the SEQ field drops from being 22.0% of 2,200 MB to being 10.6% of 360.4 MB, an overall saving of more than 445 MB, and a reduction to around just 8% of the original SEQ requirement.

On the other hand, the QUAL field has both a larger alphabet and less repetition, because the estimated error is a function of many factors, and is only loosely correlated with its position in the underlying genome. It requires fully two-thirds of the striped compressed representation.

The SEQ field

We now focus on the SEQ field, and consider if there are additional storage savings possible.

One of the reasons that gzip obtains such good compression on SEQ components is the large number of repeated subsequences, which arise because of the process used to generate SAM files. The biological source material contains many copies of the underlying genetic sequences. When cut randomly into segments, each particular nucleotide in the original appears in any number of the reads that are reproduced into the SAM file. Within the SAM file each of these reads might partially or fully overlap with other reads in the file, or might be unique. Moreover, when reads do overlap, they are likely to be highly similar. If the process used to generate them were infallible, and if there were no mutations in any of the cells in the source material, the reads at any sequence location should be in perfect agreement. But there are discrepancies introduced by the inexactness of the process and machinery used; by the possibility that some of the reads arise from mutated cells; and by computation errors made when the alignments are identified.

Even so, there can be a high degree of repetition across the multiple reads that span any particular location in the genome, and even though it is a general-purpose text compression program rather than one tailored to DNA sequences, gzip does a good job of identifying and exploiting the common subsequences.

An obvious question is whether a tailored compression regime might do better. In particular, there is additional information associated with each read that could be used to explicitly identify the set of reads that are believed to have overlapping SEQ fields. It is not mandatory for SAM records to include a meaningful RNAME – like many of the fields listed in Table 1, it can be stored as a * to indicate "not present". But when it is present, it indicates which

	HG00113	HG00559	Local
Different RNAMEs	1	1	68
Overlapping reads (%)	97.1	99.9	80.8
Overlapping bases (%)	94.4	98.6	75.2
Median multiplicity	102	6	16

Table 5: Statistics for the three SAM files in terms of overlaps relative to the given reference sequences. The final row shows the median, taken over the set of all bases present in the file, of the number of bases that share the same offset in regard to the same RNAME sequence, counting one for bases that appear in read alignments with no RNAME specified.

reference sequence the SEQ field is like, and the numeric POS field indicates an offset relative to that reference sequence. When these two fields are available it is thus possible for an encoder to permute the records in the SAM so that all of the alignments that relate to a given reference sequence are placed in a cluster of consecutive records, and also for them to be ordered by POS within that cluster.

If the encoder permutes the records within the SAM file, there are then two options for decompression. The first is for a permutation vector to be added to the compressed representation, so that the decoder is able to invert the permutation and restore the original ordering. As noted in Section 2, if there are n records, then a total of $n\lceil\log_2 n\rceil$ bit suffices for this purpose. If the downstream applications do not require that the original SAM file ordering be retained – indeed, if there is no importance of any sort associated with the original ordering of the records, and their arrangement was an arbitrary artifact of the process that generated them – then there is no need to store the permutation vector, and the compression regime can be *information preserving* rather than lossless.

Table 5 shows the extent of the read overlaps in the three example files. Only the file Local has less than around 95% or more overlaps in terms of both reads and individual bases. It is lower is because no RNAME field is supplied for 18% of the reads, and hence it is not possible to identify overlaps for those SEQ components.

If it could be assumed that the set of reference sequences used in each SAM file was available to the compressor and decompresser as a static external resource, then each of the reads in the SAM file could be compressed relative to it. But this would be a risky assumption, and would mean that the compressed SAM file could not be regarded as being self-contained.

Presumed Reference Sequence

Instead, we construct what we call a *presumed reference sequence*, or PRS, that is specific to the SAM file in question, and does not require linkage to any external resources. Figure 4 shows how this is done, using as an example four reads with slightly different POS fields and the same RNAME field. First, the set of reads in the SAM file are ordered by the RNAME field, and then by the POS field specified for each one. Where there is overlap, the reads are aggregated by a simple majority vote to form a presumed reference sequence.

In Figure 4(a) it is supposed that four reads each of $20+$ bases are slightly offset from each other, and are the only four reads that span a section of the REF sequence. The presumed reference sequence is shown at the top, and is in complete agreement with the four reads in all but 11 (out of a total of 97) of the base positions, as shown in Figure 4(b). (For reasons that are explained shortly, the N in read four is not permitted to install an N into the PRS.) To encode these four reads, the PRS string spanning 38 bases is stored, then four sets of "offset, length, exceptions" information, one per read, detailing: the commencement within the PRS of that read (which can be inferred from the POS field); its length (which is usually constant throughout the whole SAM file); and a list of locations in the read where bases other than is stored in the PRS are to be inserted.

Representing exceptions

Figure 5 gives more details of the process used to encode the reads via copies from the PRS and a list of exceptions.

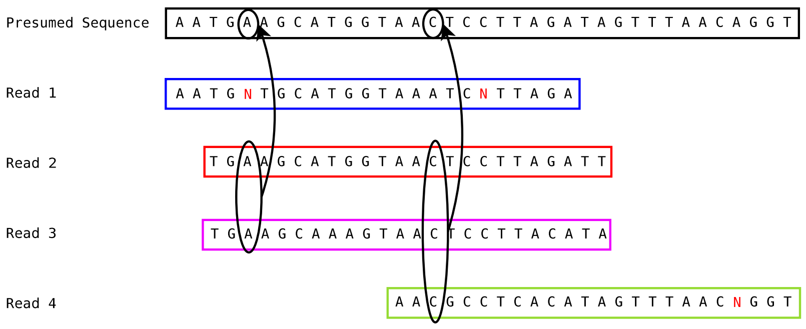
If the compressed SAM file is to be self-contained, the first component to be stored must be the PRS. It is a string of bases, typically over an alphabet of size $\sigma = 4$, covering symbols A, C, G, and T with their usual meanings, and requiring two bits per base to economically encode them. Note that N, the symbol used to indicate "unknown" symbols, is not permitted in the PRS. If it is the majority symbol – as is the case with the N in Read 4, it is replaced in the PRS by any other symbol.

Each of the reads relative to the PRS is stored as an offset relative to the previous read's POS; plus a length; plus a set of instructions from which the read can be reconstructed. To achieve the third component, each read is decomposed into alternating "copy" and "replace" counts, illustrated in the lower part of Figure 5. Because the exceptions are only required when a base differs from the one stored in the corresponding position in the PRS, it is beneficial to further split the stream of exceptions into four parts, denoted in the figure as "not A", "not C", and so on. For example, in Figure 5 the exception in Read 3 consists of the two bases AA. The PRS contains TG at the corresponding positions, so the first A is stored in the "not T" subsequence, and the second in the "not G" subsequence.

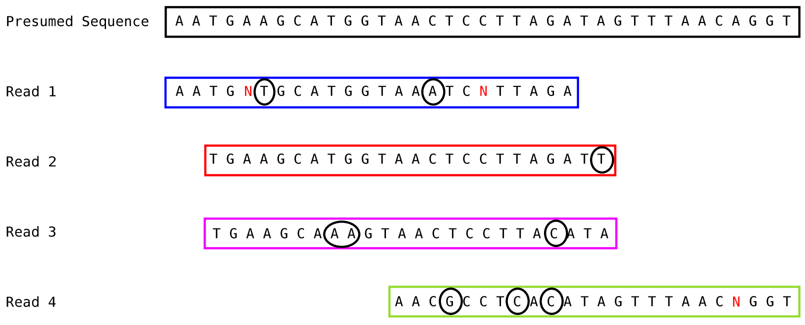
These nine elemental components are striped across nine arrays that collectively allow the set of reads to be reconstructed, provided that the PRS is also available. Each of the arrays can be thought of as being a set of integers with a specialized purpose and localized distribution pattern. For example, the values in the "Replace" array will typically be much smaller than the values in the "Copy" array, and should be stored using a different encoding.

The decision to avoid N values in the PRS is a consequence of their low frequency in the SEQ sequence. If N symbols were allowed in the PRS, the alphabet used to represent it has $\sigma = 5$ symbols. On the other hand, if the PRS is restricted to the standard $\sigma = 4$ bases, each can be represented directly using two bits. Moreover, because the PRS is an internally-stored aid to compression rather than an expected output of the process, it can, if it simplifies processing or saves space, be approximated. Hence, explicitly preventing N values does not damage the correctness of the arrangement, since whatever is in that position in the overlapping reads can be coded as an exception to the base that is arbitrarily used to replace the N.

In addition, because N is such a rare symbol, it is also helpful to code it differently when it appears in the four "not" sequences. Where an N appears in any of the overlapping reads, it is replaced temporarily by whatever symbol appears in that position in the PRS, and represented as a simple copy (or as part of a longer copy). To undo this deliberate simplification, an overall list of locations in the reads that *are* N is also maintained. This list is consulted as the final step in the decoding process, and any occurrences of N within the designated range of positions are reinstated into the output SEQ sequence before it is written. This approach implies a slight redundancy. But N symbols are relatively rare, and the cost of doing it this way is far less than the overhead cost of working with $\sigma = 5$ when coding the PRS, and of working with $\sigma = 4$ when coding the four "not" sequences.


Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
Read 2
T G A A G C A T G G T A A C T C C T T A G A T T
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
Read 4
A A C G C C T C A C A T A G T T T A A C N G G T

(a) Constructing the presumed reference sequence by majority vote.


Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
Read 2
T G A A G C A T G G T A A C T C C T T A G A T T
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
Read 4
A A C G C C T C A C A T A G T T T A A C N G G T

(b) Identifying exceptions to the presumed reference sequence.

Figure 4: Construction of a presumed reference sequence by taking the majority opinion of the overlapping reads at each position: (a) a set of overlapping reads, with N symbols considered to be non-voting; and (b) the locations in those reads at which discrepancies occur, again ignoring any Ns.

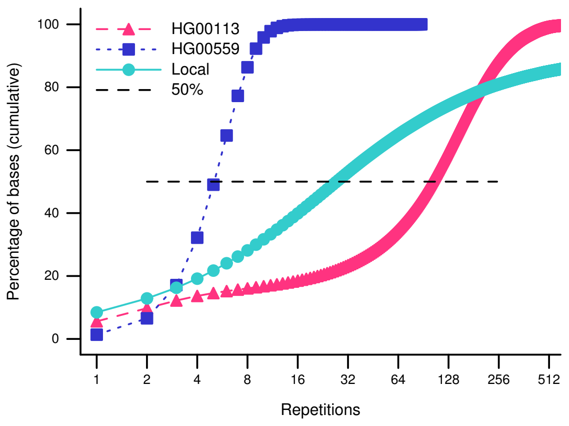


Figure 6: Cumulative plot of the fraction of bases in the SAM file as a function of the number of overlapping reads each base belongs to.

Figure 6 shows the extent to which bases overlap. To generate the three curves, the total set of bases in each file was ordered according to the number of other bases that were coincident with this one. For example, less than a quarter of the total number of bases in each of the three SAM files appeared as the only one aligned with that particular position in the RNAME sequence. More importantly, 50% of the bases share their position with 5 or more other bases in `HG00559`, with 15 or more other bases in `Local`, and with more than 100 other bases in `HG00113`. The number of overlaps arising from the multiplicity of reads is substantial.

Table 6 brings the various components together. Each of the data types comprising the compressed SEQ stream is shown, together with the number of instances of that type of object. The cost of storing each component using a suitable static code is also shown. For example, to code the "not" sequences, each of which consists of symbols over an alphabet of size $\sigma = 3$, the three binary codewords `0`, `10`, and `11` are used, with an average cost of not more than 1.67 bits per symbol, provided only the most frequent of the three alternatives is assigned the one-bit codeword. Similarly, a range of binary codes and Elias γ codes (see Moffat and Turpin (2002) for details) are used for the other components. The critical change that has been achieved compared to the `gzip` approach is that none of the values in Table 6 are based on adaptive (or even semi-static) models or codes.

As was already noted in connection with Table 5, the third of the data files, `Local`, contain a significant fraction of reads that are not associated with an identified reference sequence. These read alignments are coded as if they were non-overlapping, that is, as bases over an alphabet of size $\sigma = 4$ symbols, with the `N` symbols reinstated subsequently, and no use made of a PRS. Those costs are shown in the bottom part of the table.

Summed over the various components, Table 6 shows that the deconstructed SEQ stream can be represented in space that is always at least a little less than is required by `gzip` equivalent (note that the representations in Table 6 also absorb the separate POS field, stored as the offset), and on file `Local` is about half of that space. More importantly, the proposed approach is structured in a manner that has considerably more flexibility than `gzip` in terms of access options, because it is based entirely around static models and codes. Access operations on SAM-format data are discussed shortly, in Section 5.

The QUAL field

Genomic data is discrete rather than sampled-continuous, and hence, at face value, not amenable to lossy compres-

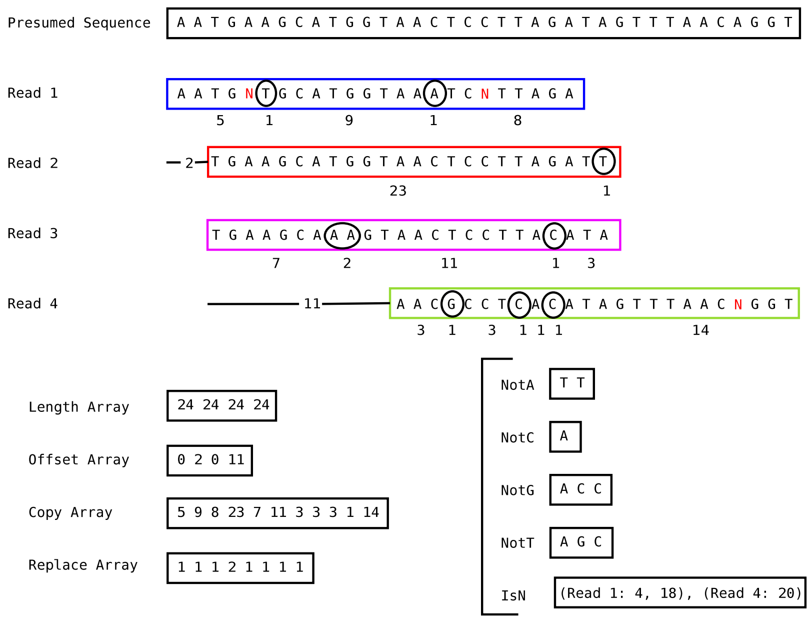

Presumed Sequence
A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T
Read 1
A A T G N T G C A T G G T A A A T C N T T A G A
5 1 9 1 8
Read 2
2
T G A A G C A T G G T A A C T C C T T A G A T T
23 1
Read 3
T G A A G C A A A G T A A C T C C T T A C A T A
7 2 11 1 3
Read 4
11
A A C G C C T C A C A T A G T T T A A C N G G T
3 1 3 1 1 1 14
Length Array
24 24 24 24
Offset Array
0 2 0 11
Copy Array
5 9 8 23 7 11 3 3 3 1 14
Replace Array
1 1 1 2 1 1 1 1
NotA
T T
NotC
A
NotG
A C C
NotT
A G C
IsN
(Read 1: 4, 18), (Read 4: 20)

Figure 5: Representing a group of SEQ fields as independent components relative to a presumed reference sequence.

Component	Code	HG00113		HG00559		Local	
		Number	Size (MB)	Number	Size (MB)	Number	Size (MB)
Reads with an RNAME *field supplied*							
PRS	binary(4)	52,641,432	12.55	58,432,811	13.93	106,253,672	25.33
Length array	constant	5,630,221	0.00	2,515,117	0.00	8,095,516	0.00
Offsets	Golomb	5,630,221	2.01	2,515,117	1.80	6,651,885	3.17
Copies	Elias γ	18,926,297	11.54	14,678,161	6.87	24,642,246	16.88
Replacements	Elias γ	14,358,719	5.09	12,882,985	4.25	19,814,961	5.66
not A	binary(3)	10,649,437	2.12	9,472,568	1.88	11,341,438	2.25
not C	binary(3)	11,813,272	2.35	8,686,985	1.73	12,796,833	2.54
not G	binary(3)	11,266,210	2.24	8,697,083	1.73	12,407,771	2.47
not T	binary(3)	10,850,150	2.16	9,432,063	1.87	11,332,794	2.25
is N	binary(2^{30})	6,446	0.01	226,187	0.26	98,348	0.31
Reads without an RNAME *field supplied*							
Bases	binary(4)	0	0.00	0	0.00	144,363,100	34.42
is N	binary(2^{30})	0	0.00	0	0.00	788,351	0.93
Total			40.06		34.32		96.22

Table 6: Costs for SEQ components when stored as shown in Figure 5. Direct application of gzip to the original SEQ and POS fields in a striped approach results in corresponding costs of 46.43 MB, 39.91 MB, and 232.54 MB respectively.

sion. But some components of the SAM-format data have elements of sampling associated with them, most notably the QUAL field (described in Section 3). Moreover, as is illustrated in Table 4, the SEQ components and the dominant space requirement when SAM data is compressed. As a tangible reminder of this, combining the data presented in Tables 3 and 6 implies that, summed over all of the SEQ values in HG00113, each base can be stored in an average of 0.58 bits. But the corresponding gzip'ed representation of QUAL elements requires an average of 3.97 bits each. This imbalance makes the QUAL fields costly indeed to store.

There are two key characteristics that contribute to QUAL sequences being harder to compress than SEQ components. First, they are over a larger alphabet. The ASCII-33 mapping that is used to convert probabilities into letters typically spans between ten and twenty values in a typical SAM file. And second, it is not possible to exploit the RNAME-based overlaps when compressing QUAL fields in the way that was possible with the SEQ fields, because different reads that cover the same base position are uncorrelated – the QUAL value is influenced by a wide range of factors other than the actual offset at which it occurs.

However, the QUAL values represent quantized values over a numeric domain, and so in some situations it may be appropriate to quantize them more coarsely than via the ASCII-33 representation described in Table 1. If a tolerance p is stipulated, and limited flexibility of values introduced, with the proviso that no QUAL score may be varied by more than p units form its original quantized value, then a spectrum of lossy representations can be introduced. When $p = 0$, the representation is lossless.

To exploit this possibility, we represent the QUAL sequence as a list of tuples, consisting of a value followed by a repeat count. This run-length encoded approach will then naturally exploit repeated values, if they can be created via the flexibility introduced by the lossy representation. To encode a QUAL sequence for a given parameter p, consecutive values from the QUAL are added to a growing run while the difference between the maximum and

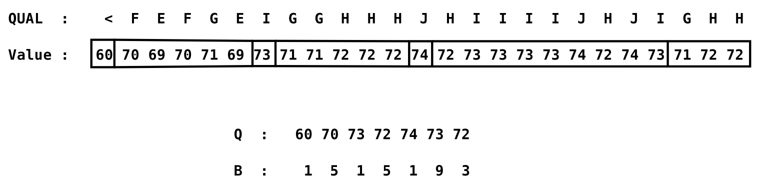

QUAL : < F E F G E I G G H H H J H I I I I J H J I G H H
Value : 60 70 69 70 71 69 73 71 71 72 72 72 74 72 73 73 73 73 74 72 74 73 71 72 72
Q : 60 70 73 72 74 73 72
B : 1 5 1 5 1 9 3

Figure 7: Lossy representation of QUAL fields. In this example $p = 1$, and each original value is represented by a mapped value that differs by at most one from the original.

Component	Code	HG00113 Size (MB)	HG00559 Size (MB)	Local Size (MB)
Fidelity parameter $p = 0$ (lossless)				
Run-length sequence	Elias γ	61.48	32.83	92.16
Byte sequence	ASCII	414.88	204.98	506.48
Byte sequence	binary (global)	311.16	153.74	379.86
Byte sequence	binary (local)	263.83	143.61	347.26
Fidelity parameter $p = 1$				
Run-lengths	Elias γ	66.89	32.98	80.60
Byte sequence	ASCII	300.98	131.70	311.51
Byte sequence	binary (global)	225.73	98.77	233.63
Byte sequence	binary (local)	195.10	94.86	214.30
Fidelity parameter $p = 2$				
Run-lengths	Elias γ	65.01	29.11	71.99
Byte sequence	ASCII	230.62	90.09	229.33
Byte sequence	binary (global)	172.96	67.57	172.00
Byte sequence	binary (local)	151.71	66.74	160.43
Fidelity parameter $p = 3$				
Run-lengths	Elias γ	60.34	24.88	64.28
Byte sequence	ASCII	177.62	64.53	176.24
Byte sequence	binary (global)	133.21	48.40	132.18
Byte sequence	binary (local)	118.93	49.23	126.26

Table 7: Lossy compression of QUAL fields. Each QUAL value is replaced by one that is at most p different from its true value. Direct application of `gzip` to the same SEQ data results in corresponding files of size 239.60 MB, 141.68 MB, and 349.83 MB respectively.

minimum of the values in the run is less than $2p$. Once a trigger item is encountered that would cause the difference to be greater than $2p$, a tuple is emitted comprising the current run length, and the mid-value that represents it. The trigger item is then the first value in the next run. Figure 7 gives and example of this process, with $p = 1$. In this example the 25 QUAL values are reduced to a total of 7 tuples, including one that includes 9 QUAL values in the range 72 to 74, all represented by the mid-value 73. With $p = 2$, the same sequence would be further reduced to just three runs, with mid-values of 60, 71, and 72 respectively.

To store the runs, we again seek to make use of static codes. Table 7 shows how this might be done, for a lossless representation with $p = 0$, and for three different lossy options with $p > 0$. In this set of measurements, the length of each run is assumed to be stored using the Elias γ code (see Moffat and Turpin (2002)); and three different approaches to representing each of the corresponding QUAL values are examined:

- as a plain ASCII bytes, as is used in the uncompressed SAM file;
- as a binary value using whole-of-file global parameters, using the number of bits indicated by the range of QUAL values stored in the SAM file; and
- as a binary value using per-alignment read parameters, with two additional bytes stored per alignment to indicate the upper and lower bounds of the local binary code.

	HG00113	HG00559	Local
$p = 0$	1.16	1.26	1.52
$p = 1$	1.61	1.97	2.48
$p = 2$	2.10	2.88	3.37
$p = 3$	2.72	4.01	4.38

Table 8: Average number of bases per run of QUAL values for three files and four different values of the fidelity parameter p.

The latter is superior in all cases, even allowing for the overhead caused by the two extra bytes. When $p = 0$ the combined cost of the runlengths and QUAL values is (unsurprisingly) greater than the cost of applying `gzip` to the same data. But as p is increased, and lossy compression is introduced, the cost decreases.

Table 8 lists the average length of the runs that are formed. As anticipated, increasing p results in increased run lengths, and hence better compression. On the other hand, lossy representations are always a risk, since future uses of data might require a fidelity of representation that seems unnecessary now. One option would thus be to store the QUAL values in lossy form using a relatively large value of p, plus store a difference list (also compressed) as a separate resource that would then allow exact reproduction of the original QUAL sequence, should it be required.

5 Querying SAM files

The key data extraction operation applied to SAM files is to isolate and present a window of the reads that it contains, identified by an RNAME and a set of offset positions within it. For example, if a particular trait is known to be encoded in some section of the chromosome, a researcher may interrogate the SAM-format data that has been generated for an individual, to see where and by how much that individual differs from the reference in regard to the identified range of bases. Because of the small but persistent possibility of error, all of the read alignments associated with that window of bases are extracted from the SAM file and displayed to the researcher.

Supporting localized extraction options via complete decoding and linear scan is expensive, even for uncompressed data. Tools for working with BAM format data are similarly costly, and involved decompression of non-trivial blocks of data, followed by sequential scanning over the read alignments, looking for overlaps.

The storage structure described in Section 4 emphasized simple static codes for two purposes:

- first and foremost, to avoid all obstacles to random-access decoding, so that given a set of pointers into the various streams of data, decoding can be commenced immediately from that index location; and
- second, to allow for faster decoding than is typically possible if adaptive models, or inverse Burrows-Wheeler transformations, or similar, need to be consulted for each character generated.

The information preserving reordering of the SAM file lines assists with these goals, grouping them first by RNAME, and then by offset relative to the start of it. To extract any/all reads that relate to a specified set of bases, the set of reads that is required is identified by seeking within the compressed representation, looking for the first read alignment whose last base overlaps with the search interval, and for the first read alignment thereafter whose first base is to the right of the search interval.

To allow such seek operations to take place, the set of read alignments will be sampled at regular intervals, and an index built that maps offset values into bit pointers into the compressed data stream (and into each of the distinct streams of bits that must be combined in order to decode). The sampling interval will control the tradeoff between speed of access and space required, with frequent samples allowing fast access, but requiring increased space.

6 Related Work

There has been a range of previous work that examines the problem of efficiently representing DNA data (the SEQ string that is a component of SAM-format files), including early discussions such as that provided by Grumbach and Tahi (1993), who identify the need to locate exact matches, palindromic matches, and complement matches at separations much greater than is the usual case in typical text compression applications. Ten years later, Manzini and Rastero (2004) describe an enhanced scheme that uses a finger-printing techniques to identify three kinds of long repetitions (exact, reverse-complement, and approximate, and possibly far apart) in DNA sequences, and uses a range of methods to code descriptions of the repetitions so identified. Their method is both fast and effective compared to other SEQ-specific approaches, but relies on an adaptive model (namely, the part of the sequence already encoded, which is used as a dictionary of long phrases), and hence is not suited to random-access decoding.

Cao et al. (2007) describe a compression approach based on multiple "experts", each of which forms a probability estimation for each symbol in the genome. The opinions of the experts are then weighted and combined, and an arithmetic coder used to convert the final overall probability distribution into an output bitstream. While this type of approach is interesting from an "exactly how much compression can be attained" point of view, it is at odds with our intention to make use of simple state-less codes that allow indexed random-access decompression.

Kuruppu et al. (2012) describe a DNA compression regime they call COMRAD, which builds an explicit dictionary of 16-base sequences, and then uses it iteratively to form longer recurring phrases, assigning a new identifier to each such extended phrase. It is an example of DNA-tailored grammar-based compression; and when applied to large sets of related genomes, is able to infer and exploit very long cross-genome repetitions. That is, the more closely related the set of sequences that is being processed, the better the more effective the compression. Kuruppu et al. also explored the scalability of their approach, by simulating the generation of extended sets of related genomes, and testing the performance of COMRAD against them.

Deorowicz and Grabowski (2011) consider genomic data stored in FASTQ-format, which, like SAM-format, maintains a QUAL string for each SEQ string, and creates files that can contain millions of short read alignments. They use an adaptive dictionary-based approach, and consider repetitions of 36 bases or more; one of the determining factors as to whether any given phrase is retained in the dictionary is the associated quality scores, working on the principle that low-quality phrases are less likely to recur than high-quality ones. As is proposed in Section 4, Deorowicz and Grabowski also make use of runlength information when storing the QUAL fields. They give results that show that their system DSRC achieves excellent compression with typical FASTQ files in the GB range being reduced to 20% or less of their original size.

Matos et al. (2012) also consider the question of multi-sequence alignment compression. They describe an approach similar to that summarized in Section 4, and derive a sequence that they call the "estimated ancestor". At the core of their mechanism is a two-dimensional context predictor (similar to the type of predictor used for bi-level image compression) that when coupled with blended probability estimates and an arithmetic coder is able to represent a set of related SEQ components in under one bit per base.

Yanovsky (2011) presents a compression implementation for multi-alignment SEQ values called ReCoil, which is designed to handle large files of genomic data stored on disk (rather than in main memory) and where repetitions might be widely separated. In this approach, reads that share common subsequences of 15 or more bases are identified, and a common substitution made at all locations, thereby saving space. The main contribution of the paper is showing how the required steps can be mapped onto sequential scanning and sorting processes that are efficient when the data is held on secondary storage.

Cox et al. (2012) apply the well-known Burrows-Wheeler transform to multi-alignment short read genomic data. But unlike the general-purpose BWT-based compression program bzip2, which uses blocks of just 900 kB, here very large numbers of reads can be accommodated through the suffix-sorting process that generates the BWT. The transformed string is then coded using a context-based estimator, and arithmetic coding. Excellent compression outcomes are achieved, because all of the like subsequences are brought together by the large-scale BWT process, and hence the probability estimates that are generated are relatively highly skewed, and the emitted arithmetic tend to be very short. It is not clear whether the same techniques can be applied to the QUAL fields that dominate SAM-format files.

One potential problem with multi-alignment compression is the need for the RNAME and POS fields to be supplied. While the methods presented in Section 4 include a

PRS in the compressed package, rather than simply referring to an external reference sequence, they nevertheless require the reads to have been aligned at the time they were generated. When read alignments have not been provided with RNAME and POS fields, we have coded them as unreferenced components, and used less effective techniques, as shown in the bottom rows of Table 6. To address this problem, Jones et al. (2012) include a *de novo assembly* component in their `Quip` software, that seeks out possible overlaps of reads seeded using overlapping 12-grams, and uses a probabilistic Bloom filter to reduce the amount of memory space required while this is taking place. The rest of `Quip` makes use of an order-12 (on bases) context-based predictor, and arithmetic coding to convert those predictions into a bitstream.

In work that is closely related to the proposal presented here, Daily et al. (2010) focus on simple static codes such as Golomb codes, Rice codes, and Elias codes, and have created a tool called `GenCompress` that handles multi-alignment files with reference to an externally-stored RNAME sequence. While we do not wish to make use of explicit external reference sequences, there are techniques in their work that may also be applicable when we construct our own implementation.

In a similar vein, Wan et al. (2012) extend generic SEQ compression to consider how best to handle collections of related reads. They carry out a detailed study of the QUAL field that is part of SAM- and FASTQ-format files, and consider mapping transformations – including lossy ones – that improve the compressibility of this data. A range of codes are considered for representing the mapped values, including binary and other static representations. Wan et al. conclude that general purpose compressors such as `gzip` and `bzip2` are less effective for QUAL values than are simple codes, and that compression effectiveness can be traded off against representational fidelity; in this regard, the preliminary results presented above can be regarded as a partial verification of their observations.

A wide range of other techniques have been proposed: Tembe et al. (2010) represent all possible distinct pairs of base and quality value using Huffman codes; Christley et al. (2009) store only the variations between sequences, coding relative to a reference sequence; and Kozanitis et al. (2010) divide reads into fragments of a chosen size, and note that neighboring quality values are correlated and can be handled using a Markov model.

7 Summary

We have described structures and techniques suitable for representing SAM-format files containing genomic data. The next significant step in this project is to implement the proposed combination of mechanism as an integrated compression tool, and verify that it is as effective as is indicated by the results obtained during this feasibility study. We will also implement the required random access operations, and measure their efficiency; beyond that we plan to seek ways of supporting that capability using modern succinct data structures so that the cost of the additional index information is minimized (or indeed, free). Our overall objective is to provide fast random interval-based access and compact storage requirements in a single package. The work presented here lays the foundations for such a development, and gives clear guidance as to the future path of this project.

References

Ansorge, W. (2009), ‘Next-generation DNA sequencing techniques’, *New Biotechnology* **25**(4), 195–203.

Bell, T. C., Cleary, J. G. and Witten, I. H. (1990), *Text Compression*, Prentice Hall, Englewood Cliffs, NJ.

Cao, M. D., Dix, T. I., Allison, L. and Mears, C. (2007), A simple statistical algorithm for biological sequence compression, *in* ‘Proc. IEEE Data Compression Conference’, pp. 43–52.

Christley, S., Lu, Y., Li, C. and Xie, X. (2009), ‘Human genomes as email attachments’, *Bioinformatics* **25**(2), 274–275.

Cox, A. J., Bauer, M. J., Jakobi, T. and Rosone, G. (2012), ‘Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform’, *Bioinformatics* **28**(11), 1415–1419.

Daily, K., Rigor, P., Christley, S., Xie, X. and Baldi, P. (2010), ‘Data structures and compression algorithms for high-throughput sequencing technologies’, *BMC Bioinformatics* **11**, 514.

Deorowicz, S. and Grabowski, S. (2011), ‘Compression of DNA sequence reads in FASTQ format’, *Bioinformatics* **27**(6), 860–862.

Grumbach, S. and Tahi, F. (1993), Compression of DNA sequences, *in* ‘Proc. IEEE Data Compression Conference’, pp. 340–350.

Jones, D. C., Ruzzo, W. L., Peng, X. and Katze, M. G. (2012), ‘Compression of next-generation sequencing reads aided by highly efficient de novo assembly’, *Nucleic Acid Research* pp. 1–9.

Kozanitis, C., Saunders, C., Kruglyak, S., Bafna, V. and Varghese, G. (2010), Compressing genomic sequence fragments using SLIMGENE, *in* ‘Proc. 14th Ann. Int. Conf. Research in Computational Molecular Biology’, RECOMB’10, pp. 310–324.

Kuruppu, S., Beresford-Smith, B., Conway, T. C. and Zobel, J. (2012), ‘Iterative dictionary construction for compression of large DNA data sets’, *IEEE/ACM Trans. Comput. Biology Bioinform.* **9**(1), 137–149.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G. and Durbin, R. (2009), ‘The sequence alignment/map format and SAMtools’, *Bioinformatics* **25**(16), 2078–9.

Manzini, G. and Rastero, M. (2004), ‘A simple and fast DNA compressor’, *Softw., Pract. Exper.* **34**(14), 1397–1411.

Matos, L., Pratas, D. and Pinho, A. (2012), Compression of whole genome alignments using a mixture of finite-context models, *in* ‘Image Analysis and Recognition - 9th International Conference (ICIAR)’, pp. 359–366.

Moffat, A. and Turpin, A. (2002), *Compression and Coding Algorithms*, Kluwer Academic, Boston, MA.

Navarro, G. and Mäkinen, V. (2007), ‘Compressed full-text indexes’, *ACM Comput. Surv.* **39**(1).

Tembe, W., Lowey, J. and Suh, E. (2010), ‘G-SQZ: compact encoding of genomic sequence and quality data’, *Bioinformatics* **26**(17), 2192–2194.

Wan, R., Anh, V. N. and Asai, K. (2012), ‘Transformations for the compression of FASTQ quality scores of next-generation sequencing data’, *Bioinformatics* **28**(5), 628–635.

Yanovsky, V. (2011), ‘ReCoil: An algorithm for compression of extremely large datasets of DNA data’, *Algorithms for Molecular Biology* **6**(23).