Bioinformatics, 31(12), 2015, 1920–1928
doi: 10.1093/bioinformatics/btv071
Advance Access Publication Date: 2 February 2015
Original Paper

OXFORD

Sequence analysis

Reference-based compression of short-read sequences using path encoding

Carl Kingsford[1,*] and Rob Patro[2]

[1]Department of Computational Biology, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA and [2]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA

*To whom correspondence should be addressed.

Associate Editor: Ivo Hofacker

Received on July 23, 2014; revised on January 16, 2015; accepted on January 29, 2015

Abstract

Motivation: Storing, transmitting and archiving data produced by next-generation sequencing is a significant computational burden. New compression techniques tailored to short-read sequence data are needed.

Results: We present here an approach to compression that reduces the difficulty of managing large-scale sequencing data. Our novel approach sits between pure reference-based compression and reference-free compression and combines much of the benefit of reference-based approaches with the flexibility of *de novo* encoding. Our method, called path encoding, draws a connection between storing paths in de Bruijn graphs and context-dependent arithmetic coding. Supporting this method is a system to compactly store sets of kmers that is of independent interest. We are able to encode RNA-seq reads using 3–11% of the space of the sequence in raw FASTA files, which is on average more than 34% smaller than competing approaches. We also show that even if the reference is very poorly matched to the reads that are being encoded, good compression can still be achieved.

Availability and implementation: Source code and binaries freely available for download at http://www.cs.cmu.edu/~ckingsf/software/pathenc/, implemented in Go and supported on Linux and Mac OS X.

Contact: carlk@cs.cmu.edu.

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

1 Introduction

The size of short-read sequence collections is often a stumbling block to rapid analysis. Central repositories such as the NIH Sequence Read Archive (SRA; http://www.ncbi.nlm.nih.gov/sra) are enormous and rapidly growing. The SRA now contains 2.5 petabases of DNA and RNA sequence information, and due to its size, it cannot be downloaded in its entirety by anyone except those with enormous resources. When select experiments are downloaded, the local storage burden can be high, limiting large-scale analysis to those with large computing resources available. Use of cloud computing also suffers from the data size problem: often transmitting the data to the cloud cluster represents a significant fraction of the cost. Data sizes also hamper collaboration between researchers at different institutions, where shipping hard disks is still a reasonable mode of transmission. Local storage costs inhibit preservation of source data necessary for reproducibility of published results.

Compression techniques that are specialized to short-read sequence data can help to ameliorate some of these difficulties. If data sizes can be made smaller without loss of information, transmission and storage costs will correspondingly decrease. While general compression is a long-studied field, biological sequence compression—though studied somewhat before short-read sequencing (e.g. Cherniavsky and Ladner, 2004; Matsumoto *et al.*, 2000)—is still a young field that has become more crucial as data sizes have outpaced increases in storage capacities. In order to achieve compression beyond what standard compressors can achieve, a compression

approach must be tailored to the specific data type, and it is likely that different compression approaches are warranted even for different short-read experimental settings such as metagenomic, RNA-seq or genome assembly applications.

Here, we present a new compression algorithm for collections of RNA-seq reads that outperforms existing compression schemes. RNA-seq experiments are extremely common, and because they are repeated for many different conditions, the number of future experiments is nearly unbounded. Among the SRA's 2500 terabases, there are over 72 304 experiments labeled 'RNA-seq' that contain short-read sequences of expressed transcripts. While the compression technique we describe here was motivated by, and optimized for, RNA-seq data, it will work for any type of short read data.

Existing short-read compression approaches generally fall into categories: reference-based schemes (Campagne *et al.*, 2013; Fritz *et al.*, 2011; Li *et al.*, 2014) attempt to compress reads by aligning them to one or more known reference sequences and recording edits between the read and its mapped location in the reference. *De novo* compression schemes (Adjeroh *et al.*, 2002; Bhola *et al.*, 2011; Bonfield and Mahoney, 2013; Brandon *et al.*, 2009; Burriesci *et al.*, 2012; Cox *et al.*, 2012; Deorowicz and Grabowski, 2011; Hach *et al.*, 2012; Jones *et al.*, 2012; Kozanitis *et al.*, 2011; Popitsch and von Haeseler, 2013; Rajarajeswari and Apparao, 2011; Tembe *et al.*, 2010) attempt to compress without appeal to a reference. SCALCE (Hach *et al.*, 2012) is one of the most effective *de novo* compressors. It works by reordering reads within the FASTA file to boost the compression of general purpose compressors.

Reference-based schemes require a shared reference to be transmitted between all parties who want to decode the data. Most reference-based schemes (e.g. Campagne *et al.*, 2013; Fritz *et al.*, 2011; Jones *et al.*, 2012; Li *et al.*, 2014) focus on compressing alignments between the reads and a set of reference sequences. As such, they work by compressing BAM files, which are the result of alignment tools such as Bowtie (Langmead *et al.*, 2009). The most-used such compression tool is CRAM (Fritz *et al.*, 2011). The limitation of these approaches is that they must encode information in the BAM file that can be recreated by re-running the alignment tool. In fact, such BAM compressors may increase the raw size of the data since all the alignment information must be preserved. Another reference-based compressor, fastqz (Bonfield and Mahoney, 2013), attempts to compress sequences directly using its own alignment scheme without first creating a BAM file.

We present a scheme that lies somewhat in the middle of these two extremes: we exploit a shared reference—a compressed transcriptome—but we do no aligning. The reference serves only to generate a statistical, generative model of reads that is then employed in a fixed-order context, adaptive arithmetic coder. The coder is adaptive in the sense that as reads are encoded, the model is updated in such a way that the decoder can reconstruct the updates without any additional information beyond the initial compressed transcriptome. In this way, if the read set differs significantly from the reference transcriptome, the statistical model will eventually converge on this new distribution, resulting in improved compression. We present a scheme that updates the model in a way that is robust to sequencing errors, which are a common source of poor compression. By sitting between pure reference-based compression and *de novo* compression, the path encoding scheme gains flexibility and generality: the same scheme works reasonably well even when the provided reference is a poor match for the sample but is significantly improved with better shared data.

The arithmetic coder uses a fixed-length context to select a conditional distribution for the following base. This scheme is efficient but has the drawback that at the start of each read, there is insufficient context to apply the model. We solve this problem with a new approach. We encode the starts of all the reads in a single, compact data structure called a bit tree. The bit tree is a general scheme for storing sets of small, fixed length (say, <30 nt) sequences. It is a simplification of other serial encoding schemes such as S-trees (de Jonge *et al.*, 1994) and sequence multiset encoding (Steinruecken, 2014).

Taken together, the bit tree for encoding the read starts and the adaptive, context-aware arithmetic coding for the remainder of the read, produce files that are on average $<66\%$ of the size of those produced by a current state-of-the-art *de novo* encoder, SCALCE (Hach *et al.*, 2012). The size reduction of these files is often larger than the space needed to transmit the reference, and thus the overhead of transmitting the reference is recovered immediately. Our approach also produces smaller files compared with reference-based schemes. Its files are on average 33% the size of those produced by CRAM (Fritz *et al.*, 2011) and on average 59% the size of those produced by fastqz (Bonfield and Mahoney, 2013). These are very large improvements in compression, a field where improvements of several percent are often difficult to achieve.

We call the resulting approach *path encoding* because, in the methods below, we draw a parallel between the design of our arithmetic coder and the problem of efficiently encoding paths in directed graphs, which is a problem that arises in genome assembly (Pevzner *et al.*, 2001) and metagenomic analyses (Iqbal *et al.*, 2012). The bit tree scheme for storing sets of short sequences (kmers) is of independent interest as the need to transmit and store collections of kmers is also increasingly common in de Bruijn-graph-based genome assembly, metagenomic classification (Wood and Salzberg, 2014) and other analyses (Patro *et al.*, 2014).

2 Algorithm

2.1 Overview

Our compression approach is composed of several different encoding techniques that are applied to the input reads as a set. First, the reads are reverse complemented based on a heuristic to determine which orientation matches the initial reference better (Section 2.6). The initial k letters of each read are stored in a bit tree data structure along with the counts of their occurrences (Section 2.3). These initial k letters of each read are called the read *head*. The reads are then reordered to place reads with the same heads next to one another. Finally, the remainder of each read (called the read *tail*) is encoded using an adaptive arithmetic coding scheme (Section 2.4) inspired by the path encoding problem (Section 2.2).

2.2 The path encoding problem

We can capture much of the information in a reference transcriptome using a graph G that has a node for every kmer that occurs in a transcript and an edge (u, v) between any two kmers u and v if v follows u, overlapping it by $k-1$ characters, in some transcript. This is a de Bruijn graph, except it is derived from several strings rather than a single string. A read r, if its sequence occurs in the transcriptome, corresponds to a path in G, and conversely there is only one path in G that spells out r. Therefore, r can be encoded by specifying a path in G by listing a sequence of nodes. This leads to a very general problem:

Problem (Path encoding) *Given a directed graph G, encode a collection of paths $P_1, P_2, \ldots, P_n$, each given as an ordered sequence of nodes of G, using as little space as possible.*

Our compression scheme uses one system for encoding the first node of each read path P_i (the read head) and another system for encoding the remaining nodes in the path (the read tails). We describe each below.

2.3 Encoding the starts of the reads with a bit tree

Let T be the kmer trie defined as follows. T has a root node that has four children, and each edge from the root node to a child is labeled by a different nucleotide in $\{A, C, G, T\}$. Each of these children themselves has four similar children, with edges for each of the nucleotides. This continues until every path from the root to a leaf node has exactly k edges on it. In this way, a complete, 4-ary tree of depth k is constructed such that any path from the root to a leaf spells out a unique kmer, and every possible kmer corresponds to some such path in T. The set of kmers K that appear at the start of some read corresponds to a subset of those possible paths, and we can construct a subtree $T_{|K}$ from T by removing all edges that are not used when spelling out any kmer in K. Knowing $T_{|K}$ allows us to reconstruct K precisely: K is those kmers spelled out by some path from the root to a leaf in $T_{|K}$.

$T_{|K}$ can be encoded compactly by performing a depth-first search starting at the root, visiting each child of every node in a fixed order (say A then C then T then G) and emitting a 1 bit whenever an edge is traversed for the first time and a 0 bit if we attempt to go to a child that does not exist. This bit stream is then compressed using a general purpose compressor, gzip (Gailly, J. and Adler, M. http://www.gzip.org). $T_{|K}$ can be reconstructed from this stream of 0s and 1s by performing a depth-first search on T traversing an edge whenever a 1 bit encountered but pruning subtrees whenever a 0 bit is read. K is then reconstructed as the set of kmers corresponding to the leaves that we encountered.

The trie T never need be actually built to perform the encoding or the decoding. Rather, a sorted list of the kmers is sufficient for simulating the traversal of the trie to encode, and decoding only ever needs to implicitly construct the part of the trie that is on the current depth-first search path. In practice, encoding and decoding of very large collections of kmers takes very little time or memory.

The same kmer may start many reads, but the encoding of $T_{|K}$ only records which kmers were used, not the number of times each was used. To store this, we write out a separate file called the count file with the count of each kmer in $T_{|K}$ in the order that the kmers will be visited during the decoding of $T_{|K}$. This file stores counts as space-separated ASCII integers, and the entire file is compressed using the gzip algorithm. $T_{|K}$ also does not record the order in which the kmers were used as read heads, so we reorder the read set to put reads with the same head adjacent to one another in the same order as their starts will be encountered during the decoding of $T_{|K}$.

This data structure is essentially the same as an S-tree (de Jonge *et al.*, 1994) specialized to kmer tries, except that no data is stored at the leaves, and because the length of every sequence is a known constant k we need not store any information about the (always nonexistent) children of nodes at depth k. It is a simplification of Steinrucken (Steinruecken, 2014) since counts are stored only for the leaves.

2.4 Arithmetic coding of read tails

Arithmetic coding (Moffat *et al.*,1998; Rissanen and Langdon, 1979; Witten *et al.*, 1987) compresses a message by encoding it as a single, high-precision number between 0 and 1. During the encoding, an interval $[a, b]$ is maintained. At the start, this interval is [0, 1], and at each step of the encoding, it is reduced to a subinterval of the current interval. At the end, a real number within the final interval is chosen to represent entire message. The interval is updated based on the probability of observing each symbol in a particular context. For path encoding, we store a probability distribution $p_u(\cdot)$ associated with each node u in G on its outgoing edges such that $p_u(v)$ gives an estimate for the probability that edge (u, v) will be the one used by a path leaving u. We also give the outgoing edges of u an arbitrary, fixed order $\langle v_1, \ldots, v_{d(u)}\rangle$, where $d(u)$ is the out-degree of u (when encoding DNA or RNA, $d(u)$ is always 4). Using this ordering, we can compute the cumulative distribution $p'_u(v_j) = \sum_{i<j} p_u(v_i)$. The probability distributions $p_u(\cdot)$ for each node in G represent the statistical generative model that encodes the information about which sequences are more or less likely.

Let $P_i = \langle u_1, \ldots, u_\ell\rangle$ be a sequence of nodes of the path P_i that we are encoding. The first node u_1 is encoded using the bit tree approach described above. Suppose we have encoded $u_1, \ldots, u_{j-1}$ and the current interval is $[a, b]$. To encode u_j, we update the interval to:

$$\left[a + (b-a)p'_{u_{j-1}}(u_j),\ a + (b-a)\left(p'_{u_{j-1}}(u_j) + p_{u_{j-1}}(u_j)\right)\right]. \quad (1)$$

This chooses a subinterval of $[a, b]$ that corresponds to the interval for u_j in $p'_{u_{j-1}}$. The intuition for why this approach achieves compression is that it requires fewer bits to specify a number that falls in a high probability (large) interval than in a low probability (small) interval. If we choose the distributions $p_u(\cdot)$ well so that common edges are given high probability, we will use few bits to encode frequently occurring symbols.

In practice, Equation (1) is not used directly because it would require infinite precision, real arithmetic which is not available on digital computers. Rather, an approach (Moffat *et al.*, 1998) that uses only finite, small precision, integer arithmetic and that rescales the current interval when necessary is used. This practical arithmetic coding has achieved state-of-the-art compression in many applications.

2.5 Initializing and updating the sequence generative model

The probability distributions $p_u(\cdot)$ for each node u specify what we consider to be a high-probability sequence (which is equivalent to a high probability path). It is here that we can use shared, prior information to influence the encoding. These distributions need not be constant—so long as the decoder can reconstruct any changes made to the distributions, we can adapt them to observed data as we see it.

We derive $p_u(v)$ using counts $c_u(v)$, which are set according to:

$$c_u(v) = \begin{cases} 10(n_{uv} + 2) & \text{if } (u, v) \text{ occurs in the reference} \\ 10n_{uv} & \text{if } n_{uv} \geq 2 \text{ and } (u, v) \notin \text{ the reference} \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

where n_{uv} is the number of times edge (u, v) was observed in the read paths that have been encoded so far. This expression for $c_u(v)$ requires an edge (u, v) to either occur in the reference or be used at least twice in a read path before it is given the larger weight. This is to reduce the impact of sequencing errors that are frequent, but unlikely to occur twice (the +2 makes kmers in the reference start as if we had seen them twice, and this is the only place where the reference is used). The 1 in the third case of Equation (2) acts as a pseudocount for edges that have not yet been observed, and the 10 in the first two cases sets the relative weight of observations versus this

pseudocount (changing this weight within a reasonable range has little effect on the compression—see Supplementary Table 6). We compute $p_u(v) = c_u(v)/\sum_w c_u(w)$.

During the encoding of reads, it is possible that we encounter a kmer that we have never seen before. In this case, we encode the base following this kmer using a default probability distribution derived from a distinct count distribution $c_0(b)$ that gives the number of times we encoded base b using this default distribution. After the first time we see a new kmer, we add it to the graph G and on subsequent observations, we treat it using Equation (2).

An alternative way to view the arithmetic coding scheme above is that the probability distribution is provided by a fixed-order (k) Markov chain for which the transition probabilities are updated as edges are traversed. The k preceding bases provide a context for estimating the probability of the next base. The read heads provide the initial context for the Markov chain. This view also motivates the need to handle sequence errors and (less common) sequence variants effectively, because an error in a read will produce an incorrect context for k bases, resulting in decreased compression.

2.6 Other considerations

The reference model only includes the forward strand of the transcript but, in an unstranded RNA-seq protocol, reads may come from either strand. We implement a heuristic for selecting whether or not to reverse complement the input read. To do this, we estimate whether the forward read r or its reverse complement $rc(r)$ will produce a smaller encoding by counting the number of times adjacent kmers in the read are both present in the reference for both r and $rc(r)$. If $rc(r)$ has a higher number of observed transitions in the statistical model, we reverse complement the read before encoding. The decision to reverse complement reads is made for all reads at the beginning of compression before any reads are encoded and before the read starts are encoded. We often do not need to store whether a read was flipped or not since in an unstranded protocol the strand that was sequenced and recorded in the file was arbitrary to begin with. However, if it is important to store the string in the direction it was originally specified, a single bit per read is recorded indicating whether the read was reverse complemented.

Due to biases in RNA-seq, and due to pooling of technical replicates, it is often the case that the exact same read sequence is listed more than once in a read file. To more compactly encode this situation, we check whether the set of reads that start with a given kmer m consists entirely of d duplicates of the same sequence. In this case, we record the number of reads associated with m in the count file as $-d$ rather than d, and we only store one of the tails in the path file.

To simplify the statistical model, any Ns that appear in the input file are translated to As upon initial input. This is a strategy taken by other compressors (Hach *et al.*, 2012) because the lowest quality value always indicates an N and all Ns must have the lowest quality value. If quality scores are not stored with the sequences and the locations of the Ns are needed, a separate compressed file is output with their locations.

Because reads are reordered, the two ends of a mate pair cannot be encoded separately if pairing information is to be preserved. Instead, when dealing with paired-ended RNA-seq, we merge the ends of the mate pair into a longer read, encoding this 'read' as described above. If the library was constructed with the mate pairs from opposite strands, one strand is reverse complemented before merging so that the entire sequence comes from the same strand in order to better match the generative model described above. This transformation can be undone when the reads are decoded.

2.7 Implementation

Software, called kpath, implementing the path encoding and decoding method was written in the Go programming language, using a translation of the arithmetic coding functions of Moffat *et al.* (1998). The software is parallelized and can use several threads to complete various steps of the encoding and decoding algorithms simultaneously. Parallelization is used for the input, output and preprocessing steps, but arithmetic coding itself is not easily parallelizable because information from all previous reads may be needed to decode the current read. To limit memory usage, the counts $c_u(\cdot)$ described above are stored in 8-bit fields with a mechanism to hold the few kmers with counts $\geq$255. This results in a small loss of compression effectiveness compared with 32-bit fields but large improvements in running times for the larger files.

2.8 Comparison with other methods

SCALCE (Hach *et al.*, 2012) version 2.7 was run with its default parameters, using `-r` for paired-end read sets. The file sizes reported are the sizes of the `.scalcer` files it produces, which encode the sequence data (except the positions of the Ns). The program fqzcomp (Bonfield and Mahoney, 2013) version 4.6 was run using the recommended parameters for Illumina data (`-n2 -s7+ -b -q3`). The file sizes used were the sizes of only the portion of its output file that encodes for the sequences, as printed by fqzcomp. Running fqzcomp with `-s8` instead of `-s7+` produced files that were still larger than SCALCE. For paired-end reads, fqzcomp often achieved better compression if it was provided with a FASTQ file that contained both ends merged into a single read, and so sizes for compressing these files (the same as provided to kpath) were used. Despite this, fqzcomp always produced files that were larger than SCALCE, and so only the SCALCE numbers are reported. Both SCALCE and fqzcomp are *de novo* compressors. Experiments with the *de novo* version of fastqz always produced larger files than fqzcomp and so its results are not reported here. The reference-based version of fastqz (Bonfield and Mahoney, 2013) version 1.5 was provided the same reference as used with path encoding (a multi-fasta file with transcripts), processed with the `fapack` program. The file sizes reported for fastqz are the sum of the sizes of its output files `.fxb.zpaq` and `.fxa.zpaq` that encode the sequences (except the Ns).

CRAM (Fritz *et al.*, 2011) is designed for compressing BAM files. To adapt it to compress sequences, read files were aligned with Bowtie (Langmead *et al.*, 2009) using `–best -q -y –sam` to an index built from the same transcriptome as used for path encoding. Quality values, sequence names and sequence descriptions were stripped from the file (fields 1 and 11), MAPQ values were all set to 255, and the RNEXT and PNEXT fields were set to '*' and '0' respectively. The resulting simplified SAM file was converted to a sorted BAM file using samtools (Li *et al.*, 2009). This file was then encoded using CRAM, and the reported file size is that of the resulting `.cram` file. (Leaving the MAPQ, RNEXT and PNEXT fields unchanged resulted in compressed files of nearly identical size.)

3 Results

3.1 Path encoding effectively compresses RNA-seq reads

We selected seven short-read, RNA-seq datasets of various read lengths and number of reads. Both single- and paired-end protocols

are represented among the sets. One dataset, SRR037452, was chosen because it is a benchmark dataset for comparing RNA-seq abundance estimation algorithms (e.g. Patro *et al.*, 2014; Roberts and Pachter, 2013). Three sets related to human embryo development were chosen as a representative set of related experiments that one might consider when investigating a particular biological question. A fifth set represents a larger collection of single-end reads of a human cell line. Finally, to assess the effect of using a human transcriptome as a reference when encoding other species, RNA-seq experiments from *Mus musculus* (SRR689233) and the bacterium *Pseudomonas aeruginosa* (SRR519063) were included. The read sets are a mix of paired- and single-end protocols, with read lengths ranging from 35 to 90 bp. Taken together, these are representative set of RNA-seq read sets.

Path encoding with $k = 16$ is able to reduce these files to 12–42% of the size that would be achieved if the file was naïvely encoded by representing each base with 2 bits (Table 1). The 2-bit encoding is approximately one-fourth the size of the sequence data represented as ASCII. (It is approximate because the ASCII encoding includes newline characters separating the reads.) Thus, path encoding reduces files to 3–10.5% of the original, raw ASCII encoding. For the human datasets, this is on average 34% smaller than the encoding produced by the SCALCE compression scheme (Hach *et al.*, 2012), a recent, highly effective *de novo* compression approach. This is also smaller than the *de novo* compressor fqzcomp (Bonfield and Mahoney, 2013), which produces files that are larger than those produced by SCALCE.

When encoding these files, a human reference transcriptome derived from the hg19 build of human genome was used to prime the statistical model. This transcriptome contains 214 294 transcripts and occupies 96 446 089 bytes as a gzipped FASTA file. This reference file is required to decompress any files that were compressed using it, but because the same reference transcriptome can be used by many RNA-seq experiments, the cost of transmitting the reference can be amortized over the many files encoded with it. The cost of transmitting or storing the 92 Mb of the reference can be recovered after <1 to 6.12 transmissions of a compressed file (Table 1, last column). Often the size of the reference transcriptome plus the encoded file is less than the size required by previous pure *de novo* compression schemes—this means that path encoding can also be seen as a very effective *de novo* encoder if the reference is always transmitted with the file, with the option to become a reference-based compressor if several files share the same reference.

Even though this reference contains only human transcripts, it is still effective when encoding RNA-seq experiments from other organisms. For mouse data (SRR689233), for example, the compression is on par with that achieved for the human datasets. For the very different bacterium *P. aeruginosa* (SRR519063), the compression gain over *de novo* encoding is still substantial (Table 1, last row). Thus, a single reference can provide enough information to effectively encode RNA-seq reads from many organisms, further allowing its size to be amortized across collections of read sets.

When compared against a recent reference-based encoding scheme, fastqz (Bonfield and Mahoney, 2013), path encoding fares well, consistently producing smaller files than the mapping approach taken by fastqz. In contrast to path encoding, using a mismatched reference for fastqz results in files that are larger than if no reference were used at all (Table 1, last 2 rows). This is because nearly no reads map sufficiently well to the reference. This shows that the non-mapping-based reference scheme implemented by path encoding is both more effective and more robust than mapping-based schemes, which require good matches along a read to benefit from the reference and which also spend a lot of their encoding recording the edits between the reference and the mapped read.

Much of the previous work on reference-based compression has focused on compressing alignment BAM files. The archetypical example of this is CRAM (Fritz *et al.*, 2011). BAM files contain more than sequences. They normally include quality values, sequence descriptions, etc. and may contain multiple alignments for each sequence. To fairly compare sequence compression schemes, we generated BAM files with a single, best alignment for each read to the reference transcriptome, and then stripped extraneous fields (including quality values and sequence names) from the resulting BAM file by setting them to the appropriate 'empty' value. These streamlined BAM files were then compressed with CRAM (Table 1). In all cases, path encoding produced a much smaller file than CRAM. This is not entirely fair to CRAM, since it attempts to preserve all alignment information in the BAM files, and it also allows for random access to records in compressed file, which path encoding does not. However, for raw compression and transmission, path encoding sequences directly is much more effective than compressing a BAM file. Again, when the reference is mismatched to the sequence (Table 1, last two rows), compression of the CRAM mapping-based approach is reduced substantially.

3.2 Encoding of the read tails represents the bulk of the compressed file

A path encoded file consists of several parts (Fig. 1). The bulk of the space is used to encode the ends of reads using a context-dependent arithmetic coding scheme (see Section 2). The first few characters (here 16) of each read are encoded via a bit tree—a data structure that encodes a set of kmers—along with counts for how many reads begin with each kmer ('read head counts' in Fig. 1). Together,

Table 1. Compressed sizes (in bytes) using various methods

Read set	Org.[a]	S/P[b]	2-Bit	SCALCE	fastqz	CRAM	PathEnc	No. Trans.[c]
SRR037452 (Bullard *et al.*, 2010)	*H.s.*	S	102 487 744	66 630 706	80 465 928	156 554 323	**43 105 624**	4.10
SRR445718 (Yan *et al.*, 2013)	*H.s.*	S	823 591 625	252 989 168	238 180 853	375 901 891	**154 960 810**	0.98
SRR490961 (Yan *et al.*, 2013)	*H.s.*	S	1 228 191 700	300 176 711	316 478 709	518 183 711	**170 613 303**	0.74
SRR635193 (Kim *et al.*, 2012)	*H.s.*	P	736 178 787	294 524 283	272 862 515	366 789 369	**187 256 974**	0.90
SRR1294122 (Friedli *et al.*, 2014)	*H.s.*	S	1 001 574 429	299 329 267	285 710 714	369 774 561	**187 808 066**	0.86
SRR689233 (Xue *et al.*, 2013)	*M.m*	P	738 357 525	233 812 737	266 126 542	929 644 204	**167 659 551**	1.46
SRR519063 (Winsor *et al.*, 2011)	*P.a*	P	688 509 129	100 403 786	183 275 880	714 193 963	**84 642 682**	6.12

[a]Organism (H.s., human; M.m., mouse; P.a., *Pseudomonas aeruginosa*).
[b]P indicates paired-end reads; S indicates single-end reads.
[c]Number of transmissions before size of the reference is recovered.

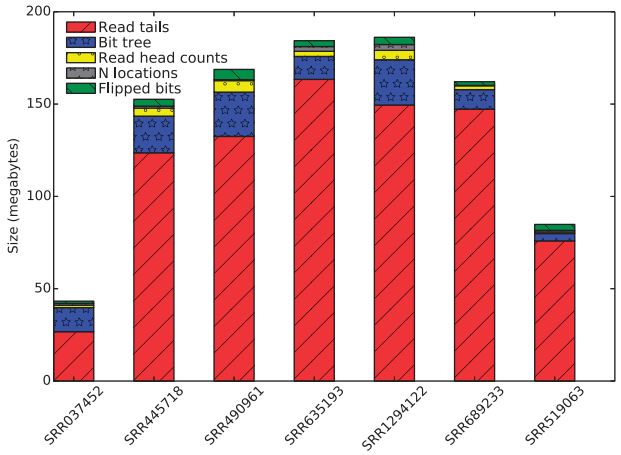

Fig. 1. Sizes of the various components of the compressed files. 'Read tails' are the portion of the reads encoded using arithmetic encoding. 'Bit tree' gives the storage used by the bit tree for encoding the read starts (the first $k = 16$ letters of each read). 'Read head counts' is the space taken to store the number of reads with each start. 'N locations' is the space to store the location of input Ns that were changed to As upon encoding. 'Flipped bits' gives the space needed to record (in a compressed format) a single bit for each read indicating whether the read was reverse complemented

the read tail encoding, the bit tree, and the counts represent the information needed to reconstruct the original reads if we do not care about recording the locations of 'N' characters or the orientation of the reads. Since SCALCE and 2-bit encoding also do not record the location of 'N's and read orientation is often arbitrary, the sum of the sizes of these three parts are what is reported in Table 1.

N locations and the original read orientations can optionally be recovered using the 'N locations' and 'Flipped bits' parts of the compressed output. The sizes of these later two parts are a tiny fraction of the overall size and so the compression effectiveness does not qualitatively change if their sizes are included (Fig. 1). While smarter encoding schemes may reduce the size of these parts of the path encoded file [for example by performing a bit-level Burrows-Wheeler transformation (Burrows and Wheeler, 1994) of the bit vector], they do not represent a large fraction of the output and so improvements to them will likely have a small effect.

3.3 Priming the statistical model results in improved compression

The availability of the reference typically results in a 15–25% reduction in file size for human read sets, a non-trivial gain in compression (Fig. 2A). For example, for a file with 3.8 gigabases of sequence (SRR1294122), path encoding with the reference produces an encoded file of 0.17 GB, while starting with a uniform, empty statistical model produces a file of 0.22 GB. For non-human data, the gain of using a human reference is naturally smaller. For mouse reads, the reference yields only a $\approx 2\%$ gain in compression—still a non-trivial size reduction in the context of large files, but much smaller than with a well-matched reference. For the bacterial data, the reference provides little help, but does not hurt compression, unlike the similar situation with mapping-based approaches.

Although the reference provides a starting point for the statistical model, the arithmetic coding we use is adaptive in the sense that read patterns observed frequently during encoding will become more efficiently encoded as they are observed. By disabling these

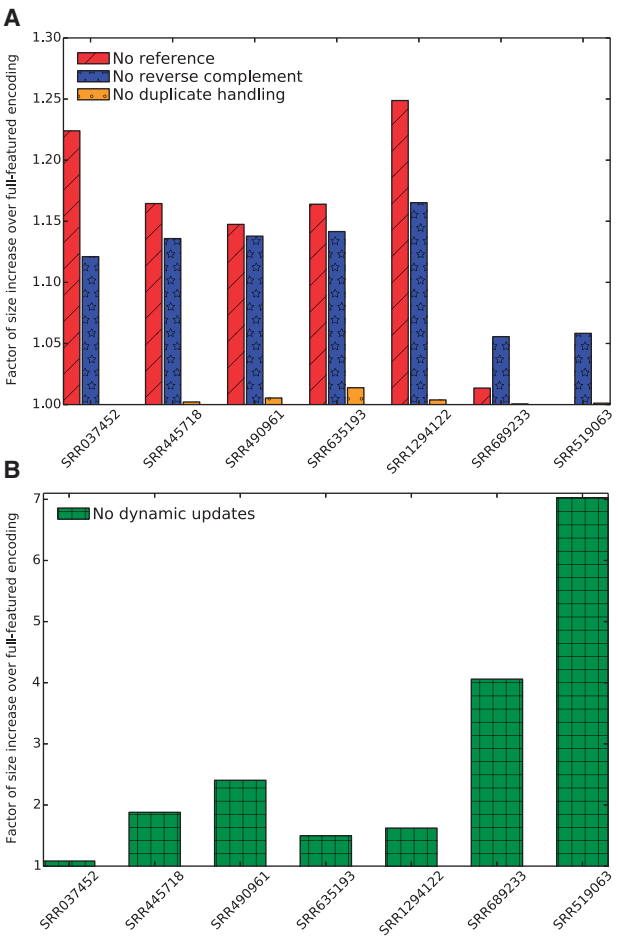

Fig. 2. Performance when several features of the path encoding scheme are disabled. All values are given as percentage over the encoding size for the encoding that uses all the features. (**A**) 'No reference' starts with an empty transcriptome reference. 'No reverse complement' disables the reverse complementation of the reads. 'No duplicate handling' disables the recognition and special encoding of exact duplicate reads. (**B**) 'No dynamic updates' gives the compression when the probabilities of the statistical model are not updated as reads are encoded

dynamic updates, we can quantify their benefit (Fig. 2B), which is substantial. The dynamic updating for the larger human files results in a encodings that are 42–92% the size of those produced by the non-dynamic model. For non-human data, where the initial model is likely to be most wrong, the adaptive coding is essential for good compression, resulting in a file that is 4.1 (mouse) or 7.0 (*P. aeruginosa*) times smaller. Thus, even with a poor initial model, a good model can be constructed on the fly by adapting the probabilities to the reads as they are processed.

These results show that path encoding provides a unified framework for good compression: when a good reference is available, it can be exploited to gain substantially in compression. When a reference is mismatched to the reads being encoded, the initial model is poor but can be improved via adaptive updates.

3.4 Effect of heuristics for reverse complementation and encoding duplicate reads, and of the choice of context size

Reverse complementing reads also provides a significant gain, particularly for reads that match the reference (Fig. 2A). This is

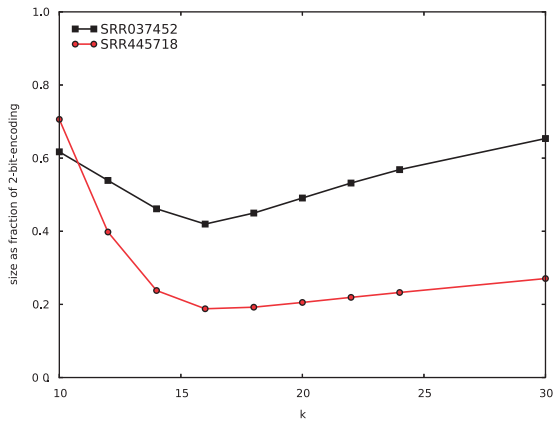

Fig. 3. Effective of kmer length. File size, represented as a fraction of the 2-bit encoding size, using various kmer lengths *k*.

because the reverse complementation allows the read to agree more with the statistical model. Recognizing some duplicate reads also leads to a modest improvement in encoding size (Fig. 2A). The improvement based on handling duplicate reads is small both because there are relatively few exact duplicate reads and because—in the interest of speed—we only tag a read as a duplicate if every read with the same first 16 bases is identical. It is possible that, in more redundant read sets, better handling of duplicate reads could result in a bigger gain.

Path encoding has one major parameter: the kmer length k used to construct the nodes of the context graph (see Section 2). A bigger k uses more of the preceding string as context to set the probability distribution for the next base, but at the same time bigger ks make the effect of sequencing errors last longer since the sequence error affects the context for k bases. In addition, a larger k requires more memory resources to encode and decode. We find that $k = 16$ is the point at which encoding is most effective (Fig. 3). This is also the point at which a kmer can fit in a single 32-bit computer word, leading to an efficient use of memory. While longer k does reduce the size of the encoding of the read ends (both because the read ends are shorter and because a longer context is used), the size of the bit tree encoding the read starts grows more quickly than the savings gained.

3.5 Encoding and decoding path-encoded files is fast

Running times and memory usage for all the tools discussed here are reported in Supplementary Tables 1–4. Path encoding the entire dataset here takes 2.45 h, including all read preprocessing. This is nearly identical to the running time for running bowtie and CRAM on the same dataset (2.44 h; Scramble (Bonfield, 2014) is a faster implementation of CRAM that may improve this running time, although the slowest step in compressing with CRAM is the read alignment). Decoding with path encoding is generally faster, taking 2.10 h for the files in Table 1. When a larger amount of memory is available, the prototype implementation provides a `-bigmem` option that reduces decoding time by 40% (Supplementary Table 5). While these running times are practical, an important direction for future work is improving the speed and memory usage of the technique.

4 Discussion

We have provided a novel encoding scheme for short read sequence data that is effective at compressing sequences to 12–42% of the uncompressed, 2-bit encoded size. To do this, we introduced the novel approach of encoding paths in a de Bruijn graph using an adaptive arithmetic encoder combined with a bit tree data structure to encode start nodes (see Section 2 for a description). These two computational approaches are of interest in other settings as well. Path encoding achieves better compression than both *de novo* schemes and mapping-based reference schemes. Because the reference for the human transcriptome is small (92 Mb) compared with the size of the compressed files, the overhead of transmitting the reference is recovered after only a few transmissions. In addition, although it would be possible to shrink the reference using a custom format, in our current implementation the reference is intentionally chosen to be merely a gzipped version of the transcriptome—a file that most researchers would have stored anyway.

Path encoding is more general than reference-based schemes because we have more flexibility in choosing how to initialize the statistical model with the reference sequence. For example, the reference could be reduced to simple context-specific estimates of GC content. This will naturally lead to worse compression but will also eliminate most of the need to transmit a reference. Technology-specific error models could also be incorporated to augment the reference to better deal with sequencing errors. In addition, single nucleotide polymorphism (SNP) data from a resource such as the HapMap project could be included in the reference to better deal with genomic variation. Framing the problem using a statistical generative model as we have done here opens the door to more sophisticated models being developed and incorporated.

Another source of flexibility is the possibility of lossy sequence compression. Path encoding naturally handles with errors since they will have low probability because they are typically seen only once (or a few times) in contrast to correct kmers that are more frequently seen. While encoding, a base that has low probability in a particular context could be converted to a higher probability base under the assumption that the low probability base is a sequencing error. Implementation of this technique does indeed reduce file sizes substantially, but of course at the loss of being able to reconstruct the input sequence. While lossy compression may be appropriate for some analyses (such as isoform expression estimation) and error correction can be viewed as a type of lossy encoding, because we are interested in lossless compression, we do not explore this idea more here.

An interesting direction for future research is to explore the use of the reference to improve the encoding of the read heads, using for example, a Huffman encoding-like scheme. Another important direction for future work is to reduce the time and memory requirements of the implementation of path encoding. Part of the reason for the higher computational demands is use of dynamic arithmetic encoding, which is needed because the probabilities of kmers need not be stationary. For example, in a file with many As in the first reads but many Cs in the later reads, the dynamic AC will adapt to this non-stationary distribution, leading to improved compression. Another direction for future work is to apply similar ideas to genomic reads, where the reference is much larger.

A recent line of work (e.g. Daniels *et al.*, 2013; Janin *et al.*, 2014; Loh *et al.*, 2012) aims at producing searchable, compressed representations of sequence information. Allowing sequence search limits the type and amount of compression that can be applied and requires some type of random access into the encoded sequences. Arithmetic encoding does not generally allow such random access decoding because the constructed interval for a given symbol depends on all previously observed symbols. However, decompression with our path encoding scheme can be performed in a streaming

manner: the encoded file is read once from start to finish, and the decoder produces reads as they are decoded. This would allow reads to be decoded as they are being downloaded from a central repository.

The other dimension of compressing short-read data is storing the quality values that typically accompany the reads. Path encoding does not attempt to store these quality values as there are other, more appropriate approaches for this problem (Cánovas *et al.*, 2014; Hach *et al.*, 2012; Ochoa *et al.*, 2013; Yu *et al.*, 2014). Path encoding can be coupled with one of these approaches to store both sequence and quality values. In fact, in many cases, the quality values are unnecessary and many genomic tools such as BWA (Li and Durbin, 2009) and Sailfish (Patro *et al.*, 2014) now routinely ignore them. Yu *et al.* (2014) showed that quality values can be aggressively discarded and without loss of ability to distinguish sequencing errors from novel SNPs. Thus, the problem of compression of quality values is both very different and less important than that of recording the sequence reads.

Our main contribution is the design of a high-performing compression scheme. We hope that our compression results will spur further reference-based read compression work in the new direction that we propose here: mapping-free, statistical compression with accompanying supporting pre-processing.

Acknowledgements

We would like to thank Darya Filippova, Emre Sefer, and Hao Wang for useful discussions relating to this work and for comments on the manuscript. This work was primarily completed while R.P. was at the Lane Center at Carnegie Mellon University.

Funding

This work is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4554 to Carl Kingsford. It has been partially funded by the US National Science Foundation [grant numbers CCF-1256087, CCF-1319998]; and US National Institutes of Health [R21HG006913, R01HG007104]. C.K. received support as an Alfred P. Sloan Research Fellow.

Conflict of Interest: none declared.

References

Adjeroh,D. *et al.* (2002) DNA sequence compression using the Burrows-Wheeler transform. In: *Procceeding IEEE Computer Society Bioinformatics Conference*. **Vol. 1**, IEEE Computer Society, Washington, DC, pp. 303–313.

Bhola,V. *et al.* (2011) No-reference compression of genomic data stored in FASTQ format. In: *IEEE International Conference on Bioinformatics and Biomedicine*. IEEE Computer Society, Washington, DC, pp. 147–150

Bonfield,J.K. (2014) The Scramble conversion tool. *Bioinformatics*, **30**, 2818–2819.

Bonfield,J.K. and Mahoney,M.V. (2013) Compression of FASTQ and SAM format sequencing data. *PLoS One*, **8**(3), e59190.

Brandon,M.C. *et al.* (2009) Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, **25**, 1731–1738.

Bullard,J. *et al.* (2010) Evaluation of statistical methods for normalization and differential expression in mRNA-Seq experiments. *BMC Bioinformatics*, **11**, 94.

Burriesci,M.S. *et al.* (2012) Fulcrum: condensing redundant reads from high-throughput sequencing studies. *Bioinformatics*, **28**, 1324–1327.

Burrows,M. and Wheeler,D.J. (1994) A block sorting lossless data compression algorithm. *Technical Report 124*. Digital Equipment Corporation.

Campagne,F. *et al.* (2013) Compression of structured high-throughput sequencing data. *PLoS One*, **8**, e79871.

Cánovas,R. *et al.* (2014) Lossy compression of quality scores in genomic data. *Bioinformatics*, **30**, 2130–2136.

Cherniavsky,N. and Ladner,R. (2004) Grammar-based compression of DNA sequences. *Technical Report* 2007-05-02. University of Washington CSE.

Cox, A.J. *et al.* (2012) Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, **28**, 1415–9.

Daniels,N.M. *et al.* (2013) Compressive genomics for protein databases. *Bioinformatics*, **29**, i283–i290.

de Jonge,W. *et al.* (1994) S+-trees: an efficient structure for the representation of large pictures. *CVGIP: Imag. Understan.*, **59**, 265–280.

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

Friedli,M. *et al.* (2014). Loss of transcriptional control over endogenous retroelements during reprogramming to pluripotency. *Genome Res*, **24**, 1251–1259.

Fritz,M.H.-Y. *et al.* (2011) Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, **21**, 734–740.

Hach,F. *et al.* (2012) SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, **28**, 3051–3057.

Iqbal,Z. *et al.* (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, **44**, 226–232.

Janin,L. *et al.* (2014) BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics*, **30**, 2796–2801.

Jones,D.C. *et al.* (2012) Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res.*, **40**, e171.

Kim,J. *et al.* (2012) Transcriptome landscape of the human placenta. *BMC Genomics*, **13**, 115.

Kozanitis,C. *et al.* (2011) Compressing genomic sequence fragments using SlimGene. *J. Comput. Biol.*, **18**, 401–413.

Langmead,B. (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Li,H. *et al.* (2009) The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, **25**, 2078–9.

Li,P. *et al.* (2014) HUGO: hierarchical mUlti-reference Genome cOmpression for aligned reads. *J. Am. Med. Inform. Assoc.*, **21**, 363–373.

Loh,P.-R. *et al.* (2012) Compressive genomics. *Nat. Biotechnol.*, **30**, 627–630.

Matsumoto,T. *et al.* (2000) Biological sequence compression algorithms. *Genome Inform. Ser. Workshop Genome Inform.*, **11**, 43–52.

Moffat,A. *et al.* (1998) Arithmetic coding revisited. *ACM Trans. Inform. Syst.*, **16**, 256–294.

Ochoa,I. *et al.* (2013) QualComp: a new lossy compressor for quality scores based on rate distortion theory. *BMC Bioinformatics*, **14**, 187.

Patro,R. *et al.* (2014) Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nat. Biotechnol.*, **32**, 462–464.

Pevzner,P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA*, **98**, 9748—9753.

Popitsch,N. and von Haeseler,A. (2013) NGC: lossless and lossy compression of aligned high-throughput sequencing data. *Nucleic Acids Res.*, **41**, e27.

Rajarajeswari,P. and Apparao,A. (2011) DNABIT Compress—genome compression algorithm. *Bioinformation*, **5**, 350–360.

Rissanen,J. and Langdon, G.Jr. (1979) Arithmetic coding. *IBM J. Res. Dev.*, **23**, 149–162.

Roberts,A. and Pachter,L. (2013) Streaming fragment assignment for real-time analysis of sequencing experiments. *Nat. Methods*, **10**, 71–73.

Steinruecken,C. (2014) Compressing sets and multisets of sequences. arXiv:1401.6410 [cs.IT].

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

Winsor,G.L. *et al.* (2011) *Pseudomonas* Genome Database: improved comparative analysis and population genomics capability for *Pseudomonas* genomes. *Nucleic Acids Res.*, **39**(Database issue), D596–D600.

Witten,I.H. *et al*.. (1987) Arithmetic coding for data compression. *Comm. ACM*, **30**, 520–540.

Wood,D.E. and Salzberg,S.L. (2014) Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biol.*, **15**, R46.

Xue,Z. *et al*. (2013) Genetic programs in human and mouse early embryos revealed by single-cell RNA sequencing. *Nature*, **500**, 593–597.

Yan,L. *et al*. (2013) Single-cell RNA-Seq profiling of human preimplantation embryos and embryonic stem cells. *Nat. Struct. Mol. Biol.*, **20**, 1131–1139.

Yu,Y.W. *et al*. (2014) Traversing the k-mer landscape of NGS read datasets for quality score sparsification. In: Sharan,R. (ed.) *Research in Computational Molecular Biology* (Lecture Notes in Computer Science). Switzerland: Springer International Publishing. pp. 385–399.